Learning Pushing Dynamics with Neural Ordinary Differential Equations

Junhao TU
Mechanical Engineering Department
University of Michigan
Ann Arbor, USA
junhaotu@umich.edu

Chenghao Lin
Robotics Department
University of Michigan
Ann Arbor, USA
lchengh@umich.edu

***Abstract*—Accurately modeling system dynamics is critical for robotics applications such as manipulation, locomotion, and planning. Traditional physics-based models often struggle to capture complex interactions like friction or contact phenomena. Recently, learning-based approaches have emerged as a promising alternative. In this project, we study two different paradigms for learning the dynamics of a planar pushing task: Residual Dynamics Models, which predict discrete changes between states, and Neural Ordinary Differential Equations (NeuralODEs), which model system evolution as continuous-time flows. We evaluate these models both in terms of predictive accuracy and their effectiveness in downstream planning using a Model Predictive Path Integral (MPPI) controller. Experiments in both free-space and obstacle-rich environments show that NeuralODEs consistently outperform residual models, achieving better generalization and control performance. Code and implementation details are available at https://github.com/linden713/Dynamic_Neural_ode.**

I. Introduction

Modeling system dynamics accurately is crucial for robotics, especially in tasks like manipulation, locomotion, and planning. In classical approaches, engineers derive analytical models based on physics. However, these models often miss complicated effects like friction, contact dynamics, or unmodeled interactions. To overcome these limitations, learning dynamics directly from data has become an important trend in modern robotics and machine learning.

One common approach is to use multilayer perceptrons (MLPs) [1] to directly predict the next state from the current state and action. MLPs are powerful function approximators and have been widely used for learning discrete-time dynamics models . An extension of this idea is the Residual Dynamics Model [2], where instead of predicting the next state directly, the network predicts the change from the current state. This residual formulation often helps learning by focusing on smaller differences, which can lead to faster convergence and better generalization.

Recently, work such as "Neural Ordinary Differential Equations" by [3] introduced a new way to model time evolution using learned continuous-time dynamics. Instead of predicting discrete next states, NeuralODEs predict the derivative of the state and use integration to simulate system evolution. This continuous approach better matches how physical systems behave, where changes are smooth and time-dependent.

In this project, we focus on learning the dynamics of the planar pushing task using both a Residual Dynamics Model and a NeuralODE model. The Residual model directly predicts the next state as an additive change, while the NeuralODE models the system as a continuous flow. We compare the performance of these two methods in terms of prediction accuracy and downstream control success using a Model Predictive Path Integral (MPPI) controller [4].

II. Background Information

A. Multi-Layer Perceptrons (MLPs) and Residual Models

Multi-Layer Perceptrons (MLPs) are fundamental building blocks in many machine learning models, including dynamics learning. An MLP consists of an input layer, several hidden layers, and an output layer, where each layer performs a linear transformation followed by a non-linear activation function such as ReLU [5].

Formally, an MLP defines a mapping $f_\theta : \mathbb{R}^n \to \mathbb{R}^m$ as a composition of layers:

$$h^{(l+1)} = \sigma(W^{(l)}h^{(l)} + b^{(l)}), \tag{1}$$

where $h^{(l)}$ denotes the output of the l-th layer, $W^{(l)}$ and $b^{(l)}$ are learnable parameters, and $\sigma(\cdot)$ is a non-linear activation.

In the context of learning system dynamics, MLPs are often used to model *residual dynamics*. Instead of predicting the next state directly, the model learns the change (or residual) between the current and next states:

$$\Delta s = f_\theta(s_t, a_t), \tag{2}$$

$$s_{t+1} = s_t + \Delta s, \tag{3}$$

where s_t is the current state, a_t is the control action, and Δs is the learned state increment. This residual formulation tends to improve learning stability and sample efficiency, especially when the true system dynamics are close to identity or only slightly perturbed.

B. Neural Ordinary Differential Equations (NeuralODEs)

Neural Ordinary Differential Equations (NeuralODEs) define a continuous-time dynamic system where the hidden state evolves according to a learned differential equation:

$$\frac{dh(t)}{dt} = g_\theta(h(t), t) \tag{4}$$

where $h(t)$ is the hidden state at time t, and g_θ is a neural network parameterized by weights θ.

In our experiments, we consider several numerical methods for integrating ordinary differential equations (ODEs):

- **Euler method**: A first-order method that approximates the next state by a simple linear extrapolation from the current derivative. It is computationally cheap but less accurate, especially for stiff or rapidly changing systems.
- **Runge-Kutta 4 (RK4)** [6]: A widely used fourth-order method that evaluates the derivative at multiple points within each time step. RK4 achieves significantly higher accuracy compared to Euler, at the cost of more computations per step.
- **Dormand-Prince 5 (dopri5)** [7]: An adaptive step-size Runge-Kutta method of order 5(4), which automatically adjusts the step size to maintain a desired error tolerance. This makes it particularly efficient and reliable for integrating complex or stiff dynamics.

Training NeuralODEs requires computing gradients with respect to model parameters. However, backpropagating through the entire solver process would be memory-intensive. To address this, NeuralODEs leverage the adjoint sensitivity method, which treats the backward pass as another ODE. Specifically, an adjoint state $a(t) = \frac{\partial \mathcal{L}}{\partial h(t)}$ is defined, and its dynamics are given by:

$$\frac{da(t)}{dt} = -a(t)^\top \frac{\partial g_\theta(h(t), t)}{\partial h} \tag{5}$$

By solving this adjoint ODE backward in time, gradients can be computed efficiently without storing the full forward trajectory.

III. Implement

A. Simulation Environment

We use the PyBullet simulator to model a planar pushing task, where a Franka Panda robot arm pushes a block across a flat table surface. The block's goal pose is indicated visually, and the robot must manipulate the block to reach this target.

We consider two variants of the environment:

- **Free Pushing**: The block is pushed across an unobstructed flat surface toward the goal.
- **Obstacle Pushing**: The block must be pushed toward the goal while avoiding a fixed obstacle placed on the table.

Examples of the two environments are shown in Fig. 1 and Fig. 2.

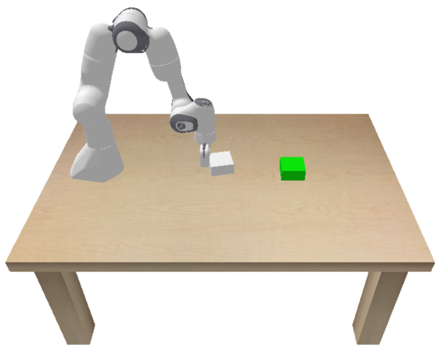


Fig. 1: Free pushing environment without obstacles.

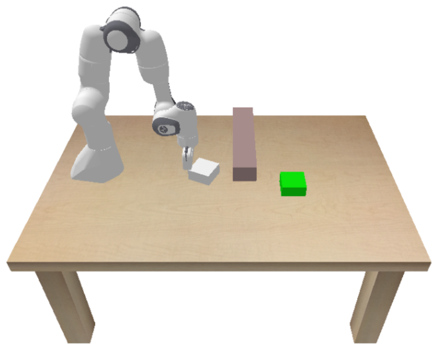


Fig. 2: Obstacle pushing environment with a fixed obstacle.

1) State Space: The system state $\mathbf{x} \in \mathrm{SE}(2)$ is the pose of the block on the table, consisting of its 2D position and orientation:

$$\mathbf{x} = \begin{bmatrix} x & y & \theta \end{bmatrix}^\top$$

We assume a *quasi-static* setting, meaning that velocities and inertial effects are negligible. When the robot stops pushing, the block also immediately stops moving.

2) Action Space: The action $\mathbf{u} \in \mathbb{R}^3$ is parameterized by:

$$\mathbf{u} = \begin{bmatrix} p & \phi & \ell \end{bmatrix}^\top$$

where:

- $p \in [-1, 1]$ specifies the pushing location along the lower edge of the block.
- $\phi \in \left[-\frac{\pi}{2}, \frac{\pi}{2}\right]$ specifies the pushing angle.
- $\ell \in [0, 1]$ specifies the pushing distance as a fraction of a maximum pushing length (0.1 meters).

3) Environment Interface: The planar pushing task is wrapped as a standard gymnasium environment, supporting the `reset()` and `step()` APIs. The environment provides the current block pose as the observation and accepts an action vector as input.

B. Data Collection and Preprocessing

We collected simulation trajectories by executing random actions and recording the resulting transitions. Each transition consists of (s_t, a_t, s_{t+1}).

1) Single-Step Data: For baseline training, we constructed a dataset of single-step samples (s_t, a_t, s_{t+1}). The dataset was split into 80% training and 20% validation sets.

2) Multi-Step Data: For multi-step prediction, we built a dataset containing sequences $(s_t, a_t, \ldots, a_{t+k-1}, s_{t+k})$, where k is the number of rollout steps (2 or 5 in our experiments). This structure allows the models to learn long-term state evolution.

C. Dynamics Models

1) Residual Dynamics Model: The residual model learns the difference between consecutive states:

$$\Delta s = f_\theta(s_t, a_t) \tag{6}$$

$$s_{t+1} = s_t + \Delta s \tag{7}$$

where f_θ is a fully connected neural network.

2) NeuralODE Dynamics Model: The NeuralODE dynamics model learns the continuous-time derivative of the state conditioned on a fixed action input. Specifically, we define an augmented vector by concatenating the state and action:

$$x_{\text{aug}} = \begin{bmatrix} s \\ a \end{bmatrix}$$

The derivative with respect to time is modeled as:

$$\frac{d}{dt}\begin{bmatrix} s \\ a \end{bmatrix} = \begin{bmatrix} g_\theta(s, a) \\ 0 \end{bmatrix} \tag{8}$$

where g_θ is a neural network that predicts the state dynamics, and the action is assumed to remain constant during the integration.

The next state is computed by solving the ODE from $t = 0$ to $t = 1$ using a numerical solver:

$$x_{\text{aug}}(t = 1) = \text{ODEsolve}\left(\frac{d}{dt}x_{\text{aug}}, x_{\text{aug}}(t = 0)\right) \tag{9}$$

Finally, we extract the updated state component as the prediction:

$$s_{t+1} = x_{\text{aug}}(t = 1)\big|_{\text{state}} \tag{10}$$

3) Network Architecture: The function g_θ is modeled as a Multi-Layer Perceptron (MLP) with:

- Input: Concatenated state and action
- Two hidden layers with 100 neurons each and Tanh activation
- Output: State derivative $\frac{ds}{dt}$

Similarly, the residual dynamics model uses an MLP f_θ with the same architecture but ReLU activations, predicting the residual Δs between consecutive states.

D. Training Strategy

Models were trained using a discounted multi-step loss over future predicted poses:

$$\mathcal{L} = \sum_{i=1}^{k} \gamma^{i-1} \|\text{pose}(s_{t+i}) - \text{pose}(\hat{s}_{t+i})\|^2 \tag{11}$$

where $\gamma = 0.9$ is the discount factor. Pose errors include translation and orientation components.

Both models were trained for 3000 epochs using the Adam optimizer with a learning rate of 1×10^{-4} and a batch size of 500. The training dataset was split into training and validation sets. Models were trained separately for different prediction horizons $k = 1, 2, 3, 4, 5$.

During training, validation loss was evaluated at every epoch. The model checkpoint achieving the lowest validation loss was saved for later use.

E. Planning and Control Using MPPI

To evaluate the learned dynamics models, we implemented a Model Predictive Path Integral (MPPI) controller.

1) Obstacle-Free Pushing: For pushing without obstacles, the cost function penalizes the squared distance to the goal pose:

$$\text{Cost}(\mathbf{x}_1, \ldots, \mathbf{x}_T) = \sum_{t=1}^{T} (\mathbf{x}_t - \mathbf{x}_{\text{goal}})^\top Q (\mathbf{x}_t - \mathbf{x}_{\text{goal}}) \tag{12}$$

where Q is weighting matrix. This cost encourages reaching the correct (x, y, θ) target configuration.

2) Obstacle-Avoidance Pushing: To support obstacle avoidance during pushing, we first implement a collision detection function. The block and the obstacle are both modeled as rectangles, and the Separating Axis Theorem (SAT) is used to efficiently check for overlaps between them.

Based on the collision detection results, we extend the cost function to penalize collisions:

$$\begin{aligned}\text{Cost}(\mathbf{x}_1, \ldots, \mathbf{x}_T) = &\sum_{t=1}^{T} (\mathbf{x}_t - \mathbf{x}_{\text{goal}})^\top Q (\mathbf{x}_t - \mathbf{x}_{\text{goal}}) \\ &+ 100 \times \text{in\_collision}(\mathbf{x}_t)\end{aligned} \tag{13}$$

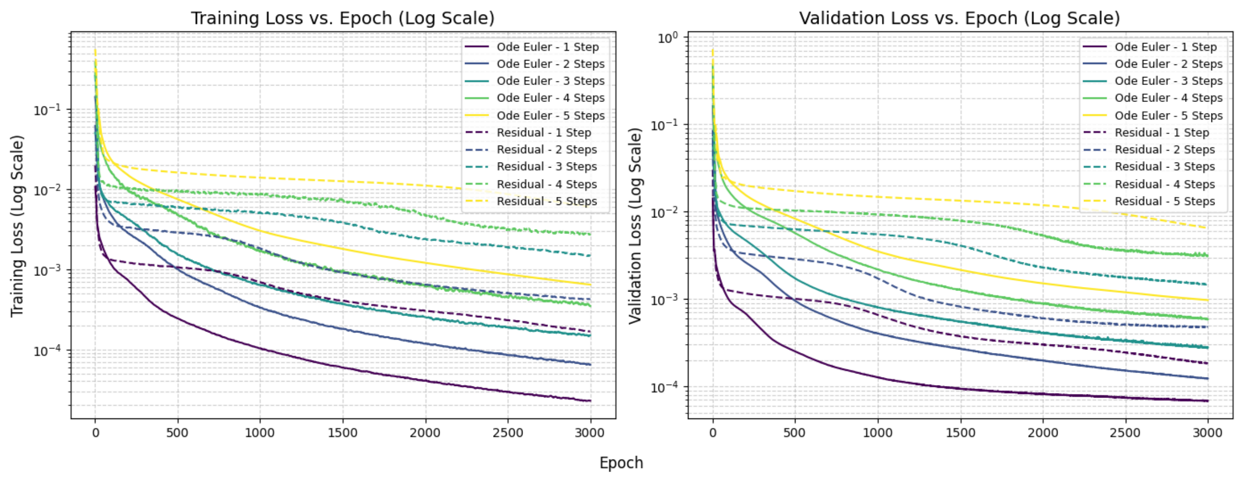
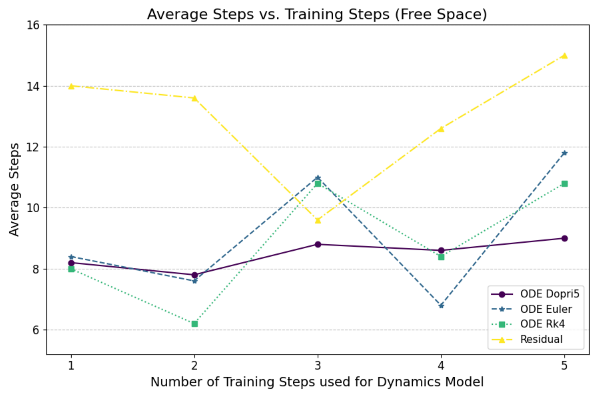
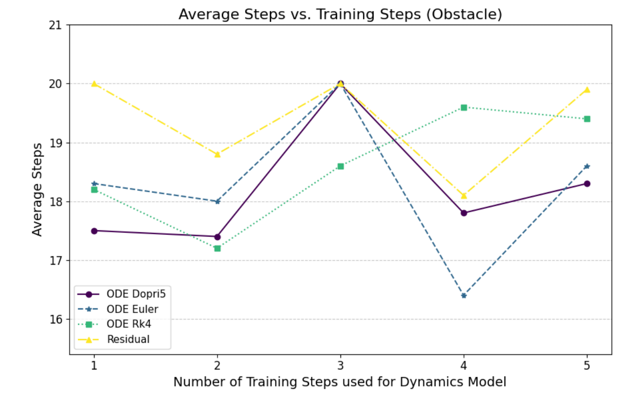


Fig. 3: Training and validation loss comparison for NeuralODE (Euler) and Residual models across different prediction horizons.



(a) Free Space: Average steps to goal



(b) Obstacle: Average steps to goal

Fig. 4: Comparison of control performance (average steps) in free-space and obstacle environments.

where in_collision($\mathbf{x}_t$) is an indicator function that outputs 1 if a collision is detected at time step t and 0 otherwise.

The additional penalty (set to 100) discourages the block from colliding with obstacles, guiding the MPPI controller to plan safer trajectories that navigate around them.

IV. Results

We investigated the performance of the NeuralODE and Residual dynamics models by systematically varying the number of prediction steps, the numerical integration algorithm, and network expressivity. We compared their training behaviors and evaluated their control performance using an MPPI controller in two scenarios: obstacle-free and obstacle-rich environments.

A. *Effect of Number of Prediction Steps*

The influence of the number of prediction steps used during training on control performance was analyzed. Fig. 4a and Fig. 4b show the average number of steps needed to complete the task in free space and obstacle environments, respectively, across different model types and prediction horizons.

1) Obstacle-Free Environment: In the obstacle-free task (Fig. 4a), NeuralODE models (ODE Dopri5, ODE Euler, and ODE RK4) achieved consistently low average step counts when trained with fewer prediction steps (1–2 steps). Among these, ODE Euler exhibited the lowest average steps at 4 prediction steps, indicating a particularly efficient learned dynamics when appropriately regularized by moderate horizon training.

In contrast, Residual models required significantly more steps across all settings, particularly at 1–2 prediction steps. Their average steps remained above 12

TABLE I: Control performance summary (success rates) in Free Space and Obstacle environments across different models, integration methods (for ODE), and number of prediction steps.

Model	Free Space					Obstacle				
	1 Step	2 Steps	3 Steps	4 Steps	5 Steps	1 Step	2 Steps	3 Steps	4 Steps	5 Steps
ODE Dopri5	100%	100%	100%	100%	100%	**20%**	**30%**	**40%**	20%	**20%**
ODE Euler	100%	100%	100%	100%	100%	**20%**	20%	30%	10%	0%
ODE RK4	100%	100%	100%	100%	100%	10%	10%	**40%**	10%	10%
Residual	100%	100%	100%	100%	**80%**	0%	10%	10%	**30%**	10%

in most cases, showing poor extrapolation ability from single-step dynamics to longer-term planning. This highlights the NeuralODE's strength in capturing global system behavior even from short-horizon training data, leveraging continuous-time modeling advantages.

2) Obstacle Environment: In the presence of obstacles (Fig. 4b), the task difficulty increased substantially, as reflected by higher overall average step counts. Nevertheless, NeuralODE-based models still consistently outperformed Residual models. Notably, the best performance among NeuralODEs was achieved when trained with 3 prediction steps: at this setting, ODE Euler and ODE Dopri5 achieved relatively low average steps while maintaining robust performance under complex environmental constraints.

Interestingly, while ODE Euler achieved the lowest steps at 4 prediction steps in free space, its best obstacle performance occurred earlier (at 3 steps), suggesting that overly long prediction horizons during training may lead to model overfitting or instability in more dynamic environments.

Residual models, again, showed fluctuating and overall poorer performance, with larger variance and less predictable trends as prediction steps varied. This suggests that Residual dynamics models are less capable of learning robust policies that generalize under environmental perturbations.

3) Summary: Overall, these results indicate that NeuralODE models benefit from training with a moderate number of prediction steps (2–3 steps), achieving a balance between local accuracy and long-term stability. In contrast, Residual models lack this robustness, especially in environments requiring complex, adaptive behavior.

B. Effect of Integration Algorithm

The choice of numerical integration algorithm in NeuralODE models significantly impacts control performance, particularly under complex environmental dynamics.

1) Obstacle-Free Environment: In the obstacle-free setting, all integration methods (Euler, RK4, Dopri5) enabled successful task completion, achieving 100

Dopri5 and **RK4** provided relatively stable and low average steps across different training step horizons, indicating strong numerical stability and precise state propagation even with minimal training supervision.

Euler integration exhibited more variability in performance. Although it maintained high success rates, the average number of steps fluctuated more noticeably with training steps, suggesting that Euler's lower-order approximation introduces compounding errors over rollout, especially with larger horizons.

2) Obstacle Environment: In the presence of obstacles (Table I and Fig. 4b), differences between integration methods became more pronounced:

Dopri5 achieved the highest robustness, maintaining relatively high success rates (up to **40%**) and lower average steps compared to other integrators. Its adaptive time-stepping and higher-order accuracy likely mitigated prediction errors that would otherwise cause collisions or failures in more challenging scenarios.

RK4 showed occasional strong performance (achieving **40%** success at 3 prediction steps) but was generally less reliable than Dopri5, particularly at larger prediction horizons.

Euler suffered the most under obstacle conditions, with success rates declining significantly as prediction steps increased, even reaching **0%** at 5 prediction steps. This highlights the limitations of first-order integration methods in dynamic, constrained environments, where precise future state estimation is critical.

Comparison to Residual Models: Across both environments, NeuralODE models—regardless of integration method—outperformed Residual models in success rates and average steps. Residual models particularly struggled in obstacle scenarios, where integration accuracy and long-term stability become essential.

Summary: These findings suggest that while simple integration methods like Euler can suffice in easy tasks, higher-order methods such as Dopri5 or RK4 are essential for maintaining robustness and accuracy in environments with complex dynamics or strict collision constraints.

C. Network Expressivity

This project evaluates network expressivity from two perspectives: learning performance during training and generalization ability as measured by control success rates.

1) Training Performance: Since the NeuralODE model (with Euler integration) and the Residual model share similar neural network architectures and differ mainly in their treatment of dynamics (explicit vs. residual form), their relative training performance directly reflects differences in model expressivity.

As shown in Fig. 3, NeuralODE models consistently achieve lower training and validation losses compared to Residual models across all numbers of prediction steps. This is particularly notable at larger horizons (4–5 steps), where Residual models' losses plateau significantly higher than NeuralODEs'. The superior fitting performance of NeuralODEs suggests that representing the system dynamics as continuous flows, even with a simple Euler integrator, provides a more expressive and physically coherent model than discrete residual updates. This better captures the smoothness and temporal consistency inherent in the planar pushing task.

2) Control Performance: Control performance comparisons further highlight the differences in expressivity:

- NeuralODE models with Euler integration outperform Residual models across all prediction horizons in both Free Space and Obstacle environments (Table I).
- Among NeuralODEs, models using more accurate solvers (Dopri5, RK4) achieve even better control performance, particularly under obstacle conditions, reflecting the benefit of more faithful trajectory prediction.

Higher expressivity enabled NeuralODE models to generalize better from training to real-time multi-step control, particularly when faced with obstacle avoidance challenges where predictive accuracy over long horizons becomes crucial.

3) Summary: These results demonstrate that formulating dynamics learning as continuous-time modeling (NeuralODE) offers significant advantages over residual-based updates. Even when using similar network architectures, NeuralODEs achieve lower training/validation losses and superior downstream control performance. Furthermore, higher-order ODE solvers (Dopri5, RK4) enhance this expressivity, yielding models that are both more accurate and more robust for planning and control tasks.

References

[1] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[3] R. T. Chen, Y. Rubanova, J. Bettencourt, and D. K. Duvenaud, "Neural ordinary differential equations," *Advances in neural information processing systems*, vol. 31, 2018.

[4] G. Williams, A. Aldrich, and E. Theodorou, "Model predictive path integral control using covariance variable importance sampling," *arXiv preprint arXiv:1509.01149*, 2015.

[5] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.

[6] J. C. Butcher, *Numerical methods for ordinary differential equations*. John Wiley & Sons, 2016.

[7] J. R. Dormand and P. J. Prince, "A family of embedded runge-kutta formulae," *Journal of computational and applied mathematics*, vol. 6, no. 1, pp. 19–26, 1980.