Neural ODE for Planar Pushing Dynamics

Miaomiao Dai, Siyuan Li

***Abstract*—We tune the hyperparameter such as intergration step/method and hidden dimensions/layers of Neural ODE–based model for planar pushing dynamics. We use the best hyperparameter to train a single step and mutiple step Neural ODE Model. Compared to the baseline residual neural network, our model achieves a lower validation loss and generates a trajectory that more closely matches the ground truth. Its effectiveness is also demonstrated through closed-loop control with an MPPI controller to push block towards the target pose.**

I. Introduction

Accurate modeling of contact-rich interactions, such as planar push, is crucial for robotic manipulation tasks in cluttered or dynamic environments. These dynamic models are usually hard to derive a analytic form due to the complex relationship and discontinuous dynamics. Learning-based approaches, such as residual neural networks, have been shown to effectively learn dynamic models and provide accurate predictions. However, these models discretize dynamics and implicitly assume that each step corresponds to a fixed time interval jump, which is not true for a continuous-time dynamics. Also, the progression over time cannot be explicitly controlled which makes it lack flexibility. Neural ODEs solve these problems by learning a differential equationwhich operates in continuous time and allows flexible control over the evolution of the dynamics.

In this work, we tune the hyperparameters of Neural ODE-based dynamics model and train a single step dynamic model and multiple step dynamic models based on the best hyperparameters. We compare our Neural ODE Model with the baseline model in different aspects and find that our Neural ODE Model has higher accuracy. Also, we demonstrate its effectiveness through closed-loop control with an MPPI controller.

II. Implementation

A. Planar Pushing Environment

For our planar pushing setup, we will use PyBullet and a custom Gym environment to simulate a Franka Panda robot arm that pushes a block on top of a table. The pushing block, the goal pose and the obstacle are visualized in white, green and brown respectively.

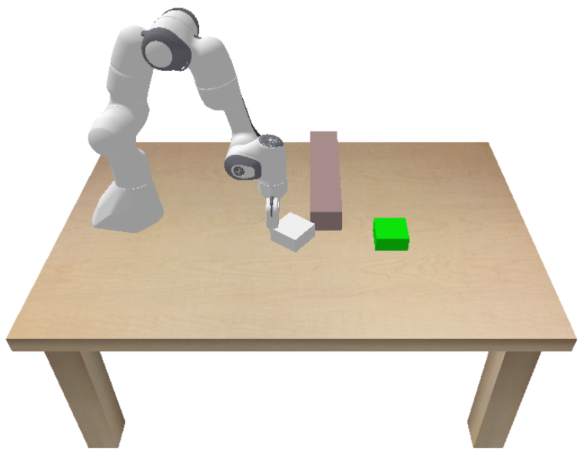

Fig. 1. Planar Pushing Environment

B. Model Architecture

Figure 2 illustrates the core Neural ODE architecture for the planar pushing task:

1) **Input** $[x_t, u_t]$**:**
   - $x_t \in \mathrm{SE}(2)$: the object's current pose (x, y, θ).
   - $u_t \in \mathbb{R}^3$: the push action parameters, consisting of:
     - $p \in [-1, 1]$: pushing location along the lower block edge.
     - $\phi \in [-\frac{\pi}{2}, \frac{\pi}{2}]$ pushing angle.
     - $\ell \in [0, 1]$ pushing length as a fraction of the maximum pushing length. The maximum pushing length is 0.1 m
2) **ODEFunc** $f_\theta(x, u)$**:**
   - A multilayer perceptron that maps the concatenated state and action $[x_t, u_t] \in \mathbb{R}^6$ to the instantaneous state derivative $\dot{X} \in \mathbb{R}^3$.
3) **ODE Solver (`odeint_adjoint`):**
   - Numerically integrates $\dot{x} = f_\theta(x, u)$ over the time interval $[0, \Delta t]$ using a chosen method (e.g., Euler, RK4, Bilinear Optimal Search Heuristic 3, Dormand-Prince 5). The input ac-

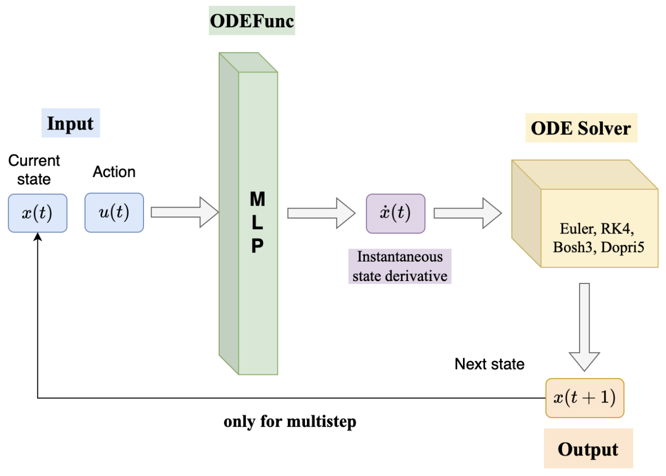

Fig. 2. Neural ODE architecture for planar pushing.

tion u is kept fixed because, in a Gym environment, the action remains constant throughout each step.

- Returns the evolved state at $t = \Delta t$, which is our prediction x_{t+1}.

4) **Output** x_{t+1}**:**

- The solver's final state, representing the model's prediction of the object's next pose.

During training, the adjoint sensitivity method is used automatically for backpropagation, keeping memory usage constant with respect to the number of integration steps and allowing flexible control over numerical precision and network complexity.

C. *ODEFunc* $f_{\theta}(x,u)$

We define

$$f_\theta : \mathbb{R}^6 \to \mathbb{R}^3 , \quad \dot{x} = f_\theta(x_t, u_t)$$

via a multilayer perceptron:

$$\begin{aligned} h_1 &= \mathrm{ReLU}(W_1[x_t; u_t] + b_1) , \\ h_i &= \mathrm{ReLU}(W_i h_{i-1} + b_i) \quad (i = 2 \dots L) , \\ \dot{x} &= W_{L+1} h_L + b_{L+1} . \end{aligned}$$

Note that the input of ODEFunc actually only needs $x_t \in \mathbb{R}^3$ since it will be concatenated with a fixed action during Δt. This network directly outputs the instantaneous velocity $\dot{x}$.

D. *Training*

We collect dataset by sampling trajectories in Panda Pushing Environment. For single step dataloader, the input are current state and action and the target is the next state. For mutiple step dataloader, the input are current state and following actions, and the target are the following states. We use Adam Optimizer with fixed learning rate as the optimizer. For hyperparameters, we use the method of controlling variables to tune them one by one in order to find the optimal set of hyperparameters.

To train the dynamics model, we minimize a weighted quadratic loss between the predicted and ground-truth next state, which we call SE2PoseLoss. For predicted next state $\mathbf{q}_p = \begin{bmatrix} x_p & y_p & \theta_p \end{bmatrix}^\top$ and ground-truth next state $\mathbf{q}_g = \begin{bmatrix} x_g & y_g & \theta_g \end{bmatrix}^\top$, the loss function is

$$\mathcal{L}(\mathbf{q}_p, \mathbf{q}_g) = \mathrm{MSE}(x_p, x_g) + \mathrm{MSE}(y_p, y_g) + r\mathrm{MSE}(\theta_p, \theta_g)$$

$$r = \sqrt{\frac{w^2 + l^2}{12}}$$

The multiple step loss is defined as follow

$$\mathcal{L} = \sum_{i=0}^{i=H} \gamma^i SE2PoseLoss(q_{pi}, q_{gi})$$

where $\gamma \in (0,1)$ is the discount factor.

E. *Control*

To demonstrate the precision of our Neural ODE Model in real world situation. We use our Neural ODE Model as the dynaic model and MPPI as the controller to direct the robot arm to push block towards the target pose. The cost function is defined as follow:

$$\mathrm{Cost}(\mathbf{x}_1, ..., \mathbf{x}_T) = \sum_{t=1}^{T} \left((\mathbf{x}_t - \mathbf{x}_{goal})^T Q (\mathbf{x}_t - \mathbf{x}_{goal}) \right. \\ \left. + 100 \texttt{in\_collision}(\mathbf{x}_t) \right)$$

where the weighting matrix Q is given by

$$Q = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0.1 \end{bmatrix} .$$

III. Results and Discussion

A. *Integration Step*

When dealing with non-adaptive integration methods, the time interval Δt needs to be divided into many small steps so that the numerical methods approximate the solution step by step moving forward in small increments. We investigate the influence of the number of integration steps on the validation loss. We use Euler as the integration method since it's much faster than RK4 and easier to do experiments. We use 0.005 as the learning rate to accelerate training, 128 as the hidden dimension and 2 as the hidden layers. Fig. 3 displays validation loss with different number of integration steps.

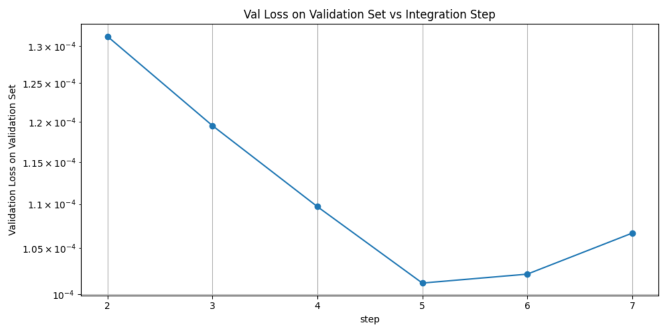

Fig. 3. Validation Loss vs Number of Integration Steps

From step=2 to step=5, the validation loss decreases. This is because more steps lead to smaller errors per step and the ode solver can have a better prediction of the state at the end of the time interval. And the validation loss begins to increase slightly starting from Step = 5. This may because the model now needs to be extremely precises over very short time intervals, and give accurate estimation of dynamic derivative for inputs that don't differ much. In this case, the model becomes to sensitive to small errors and thus the performance drops. Overall, the best integration step is 5.

B. Network Expressivity

Hidden dimensions and the number of hidden layers are the most important factors that influence the network expressivity. We study the effect on validation loss by changing the number of hidden layers [2,3] and varying the hidden dimension among [16,64,128,256]. We use Euler method and the best number of integration of steps obtained from section A for all architectures. Learning rate is set to 0.005 to accelerate training. Fig. 4 displays the validation loss using different hidden dimensions and number of hidden layers.

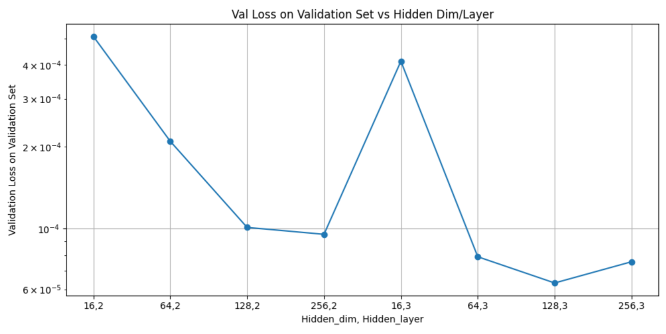

Fig. 4. Validation Loss vs Hidden Dimensions/Layers

When hidden layer=2, the validation loss decreases as the hidden dimension increases. This is because larger hidden dimension can capture more complex relationship and has stronger network expressivity. When hidden layer=3, the validation loss decreases as the hidden dimension increases from 16 to 128. However, the validation loss increases as the hidden dimension increases from 128 to 256. Our explanation for this is when that the number of parameters of the neural network architecture reaches a certain amount, the model begins to overfit. The best hidden dimension is 128 and the best number of hidden layers is 3.

C. Integration Method

We use four different integration methods (Euler, RK4, BOSH3, DoPri5) as the method of ode solver and compare their final validation loss. Euler and RK4 are non-adaptive methods and we use the best number of integration steps obtained from section A. BOSH3 and DoPri5 use adaptive algorithms and so the number of integration steps don't matter. We use 0.005 as the learning rate to accelerate training, and the best hidden dimension and layers obtained from section B for all integration methods. TABLE I shows lowest validation loss using different integration methods.

Integration Method	**Validation Loss**
Euler	1.1812×10^{-4}
RK4	1.1026×10^{-4}
BOSH3	1.1238×10^{-4}
DoPri5	9.7306×10^{-5}

TABLE I
VALIDATION LOSS FOR DIFFERENT INTEGRATION METHODS.

It turns out that DoPri5 has the lowest validation loss, RK4 and BOSH3 have the similar validation loss and Euler has the largest validation loss. The best integration method is DoPri5. However, the performance of different integration methods don't differ much. So we will use Euler method as our chosen model for the following sections to enable faster training.

D. Learning Rate

Different learning rate are investigated to choose the best one. For learning rate in [0.01, 0.005, 0.003, 0.001, 0.0005, 0.0001], we find the corresponding number of epochs to get the lowest validation loss is [500, 800, 1200, 2000, 3000, 8000]. Euler is used as the integration method. We use the best hidden dimensions and the best number of hidden layers obtained from section B for all learning rates. Fig. 5 shows the best validation loss that different learning rates can achieve.

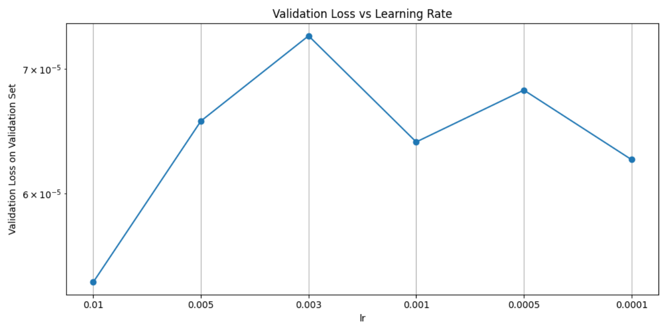

Fig. 5. Validation Loss vs Learning Rate

It turns out that when learning rate is 0.01, it has the lowest validation loss. And from 0.005 to 0.0001, the lowest validation loss don't vary much. Actually, This is inconsistent with our expectations. We had expected that smaller learning rate will contribute to lower validation loss. Our explanation for this is that small learning rates can get stuck in the local minimum. And 0.01 is the right choice since it is able to jump out of local minimum and is not too large to avoid overshooting.

E. Baseline Single Step Model

We use the residual neural network with hidden dimension = 100 and hidden layers = 2 as our baseline model. Learning rates and the number of epochs have been tuned so that the baseline model can achieve its lowest validation loss. They are 0.001 and 1000 respectively.

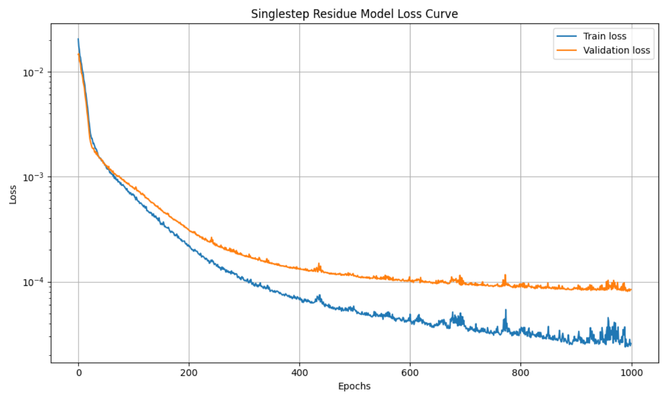

Fig. 6. Loss Curve of Residue Dynamic Mode

The baseline model achieves a validation loss of 9.525×10^{-5}.

F. Best Tuned Single Step Neural ODE Model

The best hyperparameters obtained from previous sections are used to train the single step ode model. They are listed in TABLE II.

Integration Step	Method	Hidden Dim	Hidden Layer	LR
5	Euler	128	3	0.01

TABLE II
CHOSEN HYPERPARAMETER

The training and validation loss curves are shown in Fig. 7.

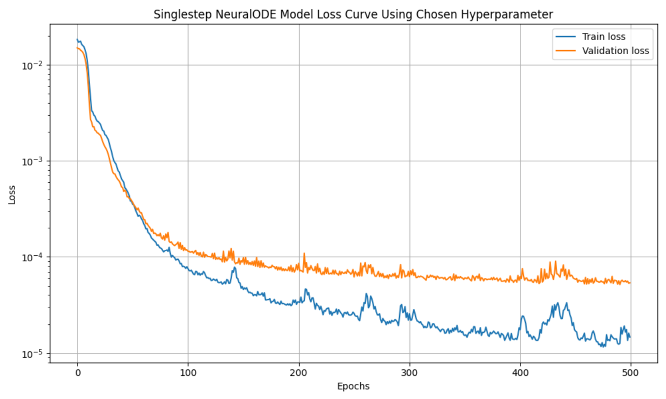

Fig. 7. Loss Curves of Single Step ODE Model

It achieves a validation loss of 5.892×10^{-5}, which is lower than our baseline model's 9.525×10^{-5}.

To evaluate the real-world accuracy of next-state prediction by the Single-Step Neural ODE model, we integrate it as the dynamics model within an MPPI controller and deploy MPPI to push the box toward a target pose in an obstacle PandaPushingEnv environment. It turns out that the goal is reached in all of the ten trials.

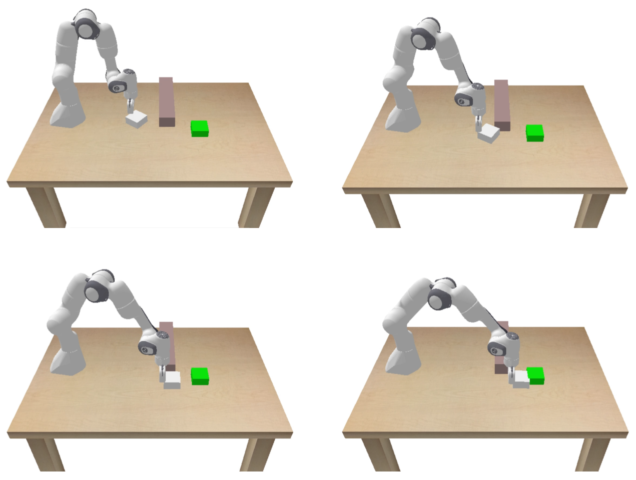

Fig. 8. Pushing Box

G. Mutiple Step Neural ODE Model

The performance of mutiple step Neural ODE Model are evaluated and compared with mutiple step Residue Model. We use the best hyperparameters from TABLE II except for a smaller learning rate to guarantee stable gradient descent since it's mutiple step loss involves more complex relationship. Then we evaluate both Mutistep Neural ODE Model and Mutistep Residue Model on a held out validation set. The results of comparison are shown in Fig. 9. The validation loss uses mutiple step loss.

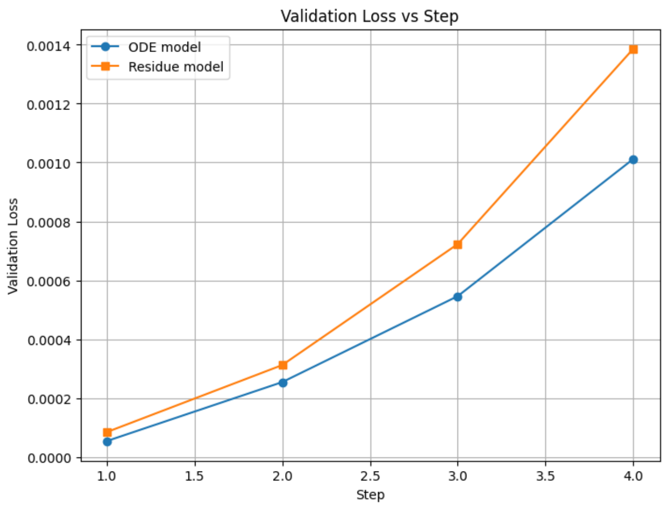

Fig. 9. Compare Mutistep Neural ODE Model and Mutistep Residue Model

At each step in [1,2,3,4], Mutistep Neural ODE Model achieves a lower validation loss than Mutistep Residue Model. This demonstrates the Neural ODE model's ability to perform multi-step predictions.

H. Trajectory Comparison

To visualize the distinct performance of Neural ODE Model and Residue Model in predicting next state, we use PandaPushingEnv to sample actions and generate a groud truth trajectory. At each step, we use Mutistep Neural ODE Model and Mutistep Residue Model to predict next state given the current state and sampled action. Their predicted trajectories and the ground truth trajectory are shown in Fig. 10.

For x coordinate, Neural ODE Model has more accurate prediction. For y coordinate, two models have similar prediction accuray. For θ coordinate, the Neural ODE model achieves significantly higher precision. These observations offer insight into why Neural ODE Model can achieve lower validation loss than Residue Model.

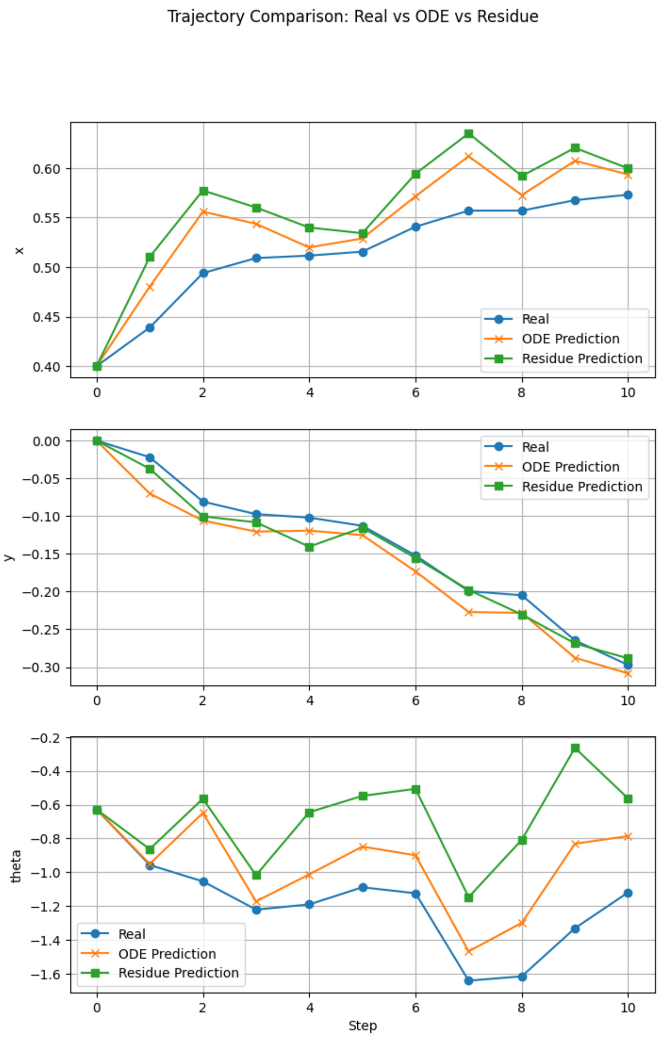

Fig. 10. Compare Trajectory

IV. Conclusion

We use the method of controlling variables to tune hyperparameters one by one in order to find the optimal set. Based on the best hyperparameter, we trained single step and mutiple step Neural ODE Model. It achieves a lower validation loss than the baseline residue neural network model. Its effectiveness as a dynamics model is validated by successfully using an MPPI controller to push the block to the target.

References

[1] R. T. Q. Chen, Y. Rubanova, J. Bettencourt, and D. Duvenaud, "Neural ordinary differential equations," in *Advances in Neural Information Processing Systems*, 2018, pp. 6571–6583.