NeuralODE for Planar Pushing Task

Aaron Tran [1], Walter Xu [1]

***Abstract*— This report details the completion of the final project for ROB498: Robot Planning for Learning and Control. We implement NeuralODE [1] to learn the dynamics of a planar pushing task. Data for this task comes from a gym [2] environment with a Franka Emika Panda arm, where the arm pushes a block to a target pose. With appropriate hyperparameter tuning, our NeuralODE absolute and residual dynamics models achieved validation losses of 4.40×10^{-5} and 4.09×10^{-5} respectively. NeuralODE can take significantly longer to train to obtain the same accuracy level as neural network models but is advantageous due to its flexibility with training input and adaptive computation during evaluation.**

I. Introduction

Hand-derived models of physical systems require strong assumptions, especially for complex systems, which often leads to the need for conservative design in safety factors and tolerances. Neural networks get around this issue by modeling the dynamics purely from data, encapsulating all of the nonlinearities and aspects that would be difficult to capture using classical modeling methods.

NeuralODE is a neural network that directly predicts the next state by encoding the underlying differential equations of a system and passing it through a blackbox ODE solver. Compared to neural network models, NeuralODE provides better memory efficiency, adaptive computation, and flexibility.

This network achieves better memory efficiency by computing loss gradients without backpropagation through the ODE solver's operations. During training, NeuralODE does not store intermediate states, giving it constant memory cost during training, though memory is still a function of model depth.

NeuralODE can be tuned for runtime, accuracy, and power needs after training. Since NeuralODE simply learns a vector field that gets passed through a blackbox ODE solver, the performance during evaluation can be changed by changing the ODE solver's parameters. This provides a lot of flexibility, as the model can be trained with high accuracy needs and be used for both low runtime, high accuracy, or low power as necessary.

Another important instance of NeuralODE's flexibility is its ability to allow for continuous time models. NeuralODE can handle irregularly-sampled data by using a continuous-time, generative approach to time-series data. This removes the need to discretize and regularly sample observations to then learn similarly discrete dynamics from. Accordingly, NeuralODE overcomes the challenge of missing data and ill-defined latent variables which other models may struggle with.

We investigate these advantages over pure neural network absolute and residual dynamics models using the torchdiffeq [3] library as our ODE solver.

*Aaron Tran and Walter Xu are with the Department of Robotics, University of Michigan, Ann Arbor, MI 48109, USA. E-mail: aartran@umich.edu.

II. Implementation

The overarching problem that we wish to solve in this project can be formulated as follows:

Given the pose $\mathbf{x}_t = [x_t, y_t, \theta_t] \in \text{SE}(2)$ of the white block shown in Fig 2 and robot arm action $u_t = [p_t, \phi_t, \ell_t] \in \mathbb{R}^3$, find $\mathbf{x}_{t+1}$.

NeuralODE solves this problem by modelling the dynamics using a learned ordinary differential equation, represented by a neural network f:

$$\frac{d\mathbf{x}}{dt} = f(\mathbf{x}) \tag{1}$$

Thus, $\mathbf{x}_{t+1}$ can be calculated as:

$$\mathbf{x}_{t+1} = \int_0^{\Delta t} f(\mathbf{x}_t) \tag{2}$$

A. Network Architecture

First, the state derivative function f is represented as a fully connected network with two hidden layers. Then, the derivative network output is integrated by an ODE solver as in equation 2 to obtain $\mathbf{x}_{t+1}$. The derivative network and the *torchdiffeq* ODE solver form the ODE block of our system, which is then embedded into the absolute and residual dynamics architecture. A full representation of our absolute model network architecture is shown in Fig 1.

The choice of having two hidden layers is based on the relative simplicity in dynamics of this experimental setup. Furthermore, having two layers is consistent with the neural network models we implemented during the course, which will serve as the baseline for comparisons.

The derivative network takes in a 6-dimensional input, and outputs a 6-dimensional $\frac{d\mathbf{x}}{dt}$, where the state $\mathbf{x}$ is a concatenation of block pose and robot action. We apply Kaiming Initialization to all fully connected layers to help facilitate more consistent back-propagation. Other network parameters are discussed in Section III

B. Environments

We train all models using data from a gym environment [2] with a Franka Emika Panda arm. Using learned knowledge of the dynamics, the robot pushes the white block in 2 to the target pose, shown as the green block, while navigating around a grey obstacle.

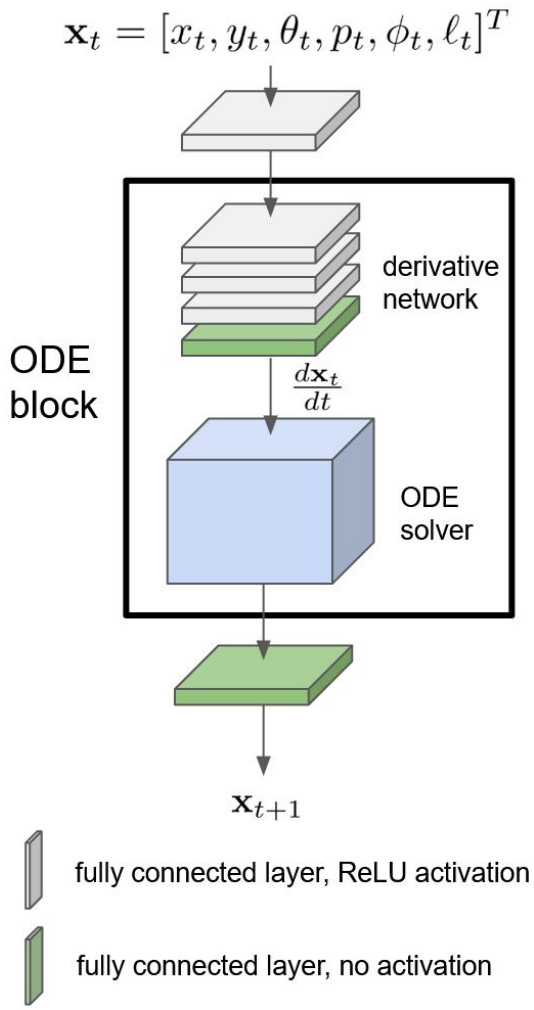

Fig. 1: NeuralODE Network Architecture

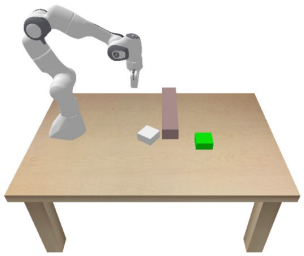

Fig. 2: Planar pushing task environment

C. Controller

The arm uses an MPPI controller, where the next state is predicted using a neural network. The controller uses the cost function in equation (3) to evaluate each trajectory in the rollout. This was the same cost function used in a previous homework and was chosen for its simplicity.

$$\text{Cost}(\mathbf{x}_1, ..., \mathbf{x}_T) = \sum_{t=1}^{T} (\mathbf{x}_t - \mathbf{x}_G)^T Q (\mathbf{x}_t - \mathbf{x}_G) + 100\mathbf{I}(\mathbf{x}_t)) \tag{3}$$

with

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

x_t is the state of the pushed block at time t and

$$\mathtt{I}(\mathbf{x}_t)) = \begin{cases} 1 & \text{if object is in collision} \\ 0 & \text{else} \end{cases}$$

This implementation of MPPI uses a horizon length of 20 and takes 1000 samples when shooting. While these values could be higher and lead to better results, they were left at these values to balance between computational load and performance.

D. Training

The goal for our training process was to match and ideally eclipse the 5×10^{-4} validation loss target given for the neural network dynamics models. This involved tuning many hyperparameters to strike a balance between accuracy and training time. The hyperparameters with the greatest impact on speed are solver types and and solver tolerances. Fixed step solvers were found to be 10-20 times faster than variable step solvers at the sacrifice of accuracy. We mitigated the effects of lower solver accuracy by training for more epochs and increasing decay rate of the Adam optimizer. The extra stabilization and finer weight adjustments towards the end of training were essential for achieving the desired loss. Raising the tolerances of the variable step solvers brought noticeable speed increases; however, we found that the highest acceptable tolerance only brought a 2-3 times speed increase over the default tolerances, which did not justify the variable solvers' use over fixed step solvers.

All training was conducted on an i-7 10700 CPU, with 8 cores at 2.9 GHz. We train for as many epochs as possible before training and validation loss difference diverges, which takes around 5 minutes. While we did train on a GPU, there were no noticeable improvements due to the ODE solver's implementation. This was further verified by the developers of *torchdiffeq* on GitHub.

We evaluate the model's SE2 pose loss, described by equations 4 and 5:

$$\mathcal{L} = \text{MSE}(x_1, x_2) + \text{MSE}(y_1, y_2) + r_g \times \text{MSE}(\theta_1, \theta_2) \tag{4}$$

$$r_g = \sqrt{\frac{w^2 + l^2}{12}} \tag{5}$$

where r_g is the radius of gyration of the square face.

We compute single step losses only. Although multi step loss would yield better results, we are more interested in comparing NeuralODE to the neural network dynamics models at a low computational cost. We do not believe there are any particular aspects of NeuralODE that lend it more towards multi step loss training than single step loss training when compared to these baseline models.

We hope to prove that NeuralODE can outperform our baseline models in accuracy. We observed that our baseline

models achieve a minimum loss of 2×10^{-4} on the validation dataset, so we set this as our target.

III. Results

Our best model achieve losses of 4.40×10^{-5} and 4.09×10^{-5} on the validation dataset, outperforming the 2×10^{-4} target. The hyperparameters used to train our best model are shown in table II. Figures 3 and 4 depict the training and validation losses for training Absolute and Residual Neural ODE models.

Hidden Dim	N Steps	Solver	Epochs	LR
128	4	Euler	1000	3×10^{-3}

TABLE I: Best Model Hyperparameters

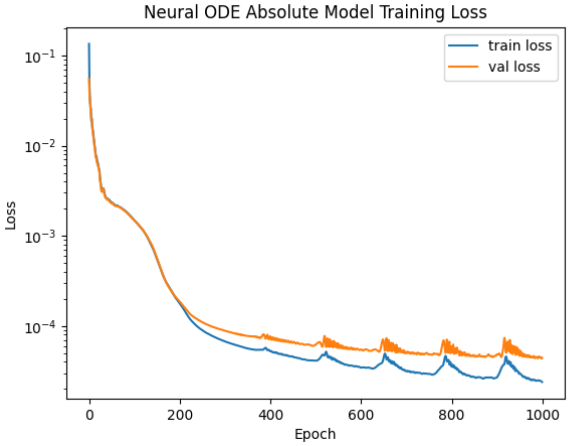

Fig. 3: Neural ODE Absolute Model Loss Curve

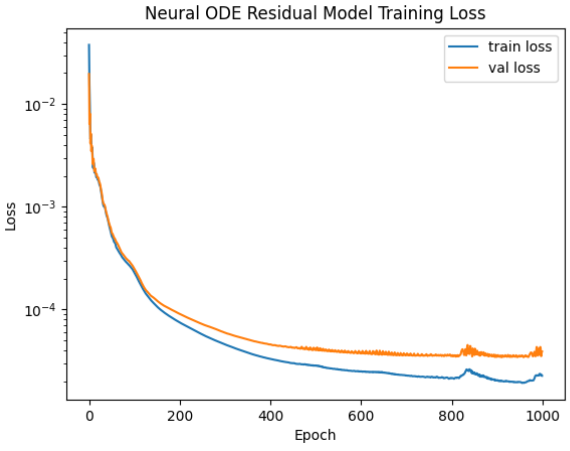

Fig. 4: Neural ODE Residual Model Loss Curve

We further verify and validate our models through the simulation task as described in Section II-B. The absolute NeuralODE model is used in combination with an MPPI controller to complete the task. Figure 5 compares the NeuralODE absolute model with the neural network absolute model, labeled as "baseline." The top plot shows the predicted trajectory given the ground truth state and action at each time step. The bottom plot shows the corresponding orientation of the object at each time step. The initial position is at $(x_0, y_0) = (0.4, 0)$ and the goal position is at $(x_G, y_G) = (0.7, -0.1)$.

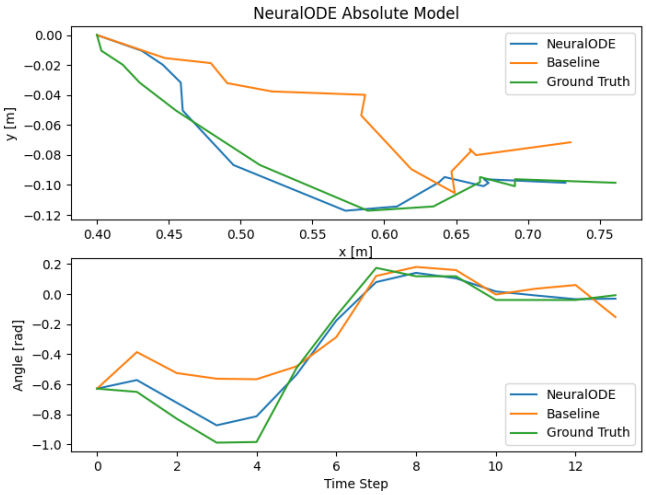

Fig. 5: Simulation Results

The simulation is a rigorous test for model accuracy. As the collision term in the cost function is binary, the controller favours the path that brings the block as close to the obstacle as possible without collision to minimize the distance towards the goal. Although our baseline model has an impressive validation loss of 2×10^{-4}, it collided with the obstacle causing a large deviation from the ground truth trajectory. A potential improvement for more consistent simulations is to augment the cost function with the distance between the block and obstacle.

IV. Discussion

When tuning our model, we performed a hyperparameter sweep to determine potentially optimal hyperparameter values and discover trends between them and the losses we obtain. The sweep revolves around some base parameters, described in Table II. Each of the hyperparameters of interest are varied in each sweep, while the other parameters are held constant. Figures 6, 7, and 8 as well as Table III show the results of our sweep.

In Figure 6, we find that though increasing the hidden dimension size generally decreases loss, it leads to overfit fairly quickly. The loss reduction is expected due to the models increase in expressivity. This same expressivity is likely too much at the higher hidden dimension sizes where the model fits to many more complex features than apparent in training data.

In Figure 7, we vary the number of integration steps that the model performs. For the range that we tested this appeared to have little to no effect on overfit. While more steps are better, there is a diminishing return on accuracy with an increasing training time.

In Figure 8, we find that the model with base hyperparameters has an optimal learning rate. This optimum likely

describes the point at which the learning rate is large enough to step outside of local minima in training but small enough to avoid oscillating too greatly around the optimal weights of the network.

In Table III, we find that DoPri5 achieved the lowest validation loss. DoPri5 is a 5th order Runge-Kutta of Dormand-Prince Shampine, BoSh3 is a 3rd order Runge-Kutta of Bogacki-Shampine, Euler is the Euler method, and Explicit Adams is the Explicit Adams-Bashforth. The first two of these are adaptive step solvers and the second two are fixed step solvers. While fixed step solvers perform worse across the board, they are much faster. Of the two adaptive solvers, DoPri5 achieved higher validation accuracy while having a slower computation time, which is expected as it is a higher order solver.

Hidden Dim	N Steps	Solver	Epochs	LR
128	2	Euler	300	10^{-3}

TABLE II: Base Parameters

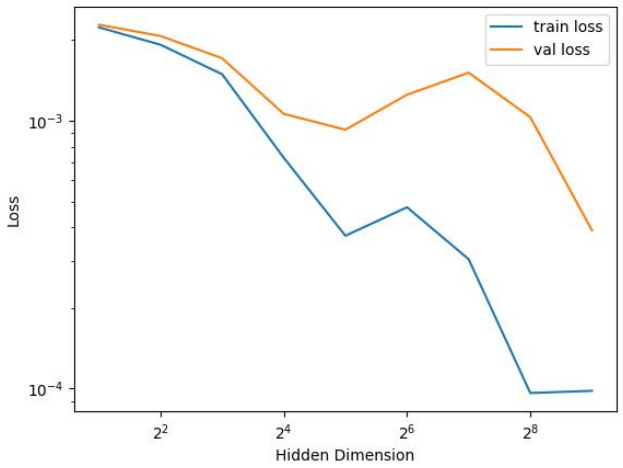

Fig. 6: Hidden Dimension vs Loss

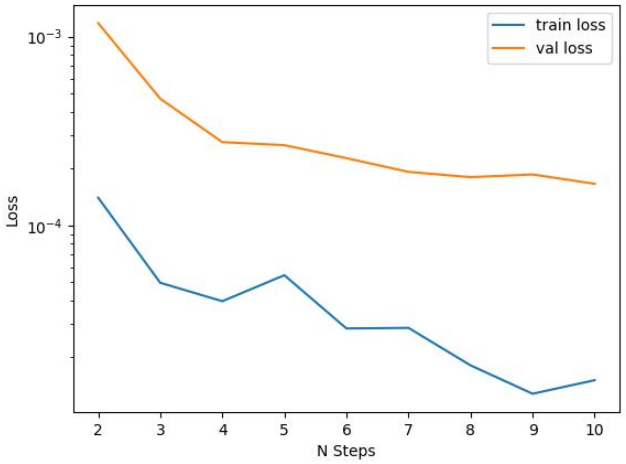

Fig. 7: N Steps vs Loss

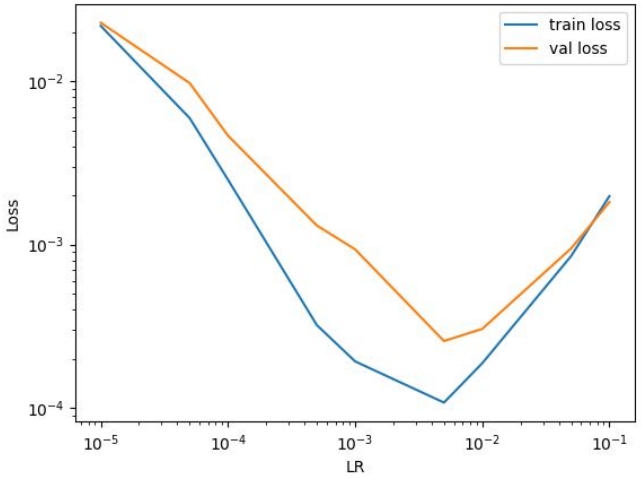

Fig. 8: LR vs Loss

Solver	Train Loss($\times 10^{-4}$)	Val Loss($\times 10^{-4}$)
Euler	2.09	9.90
Explicit Adams	1.84	6.80
DoPri5	1.81	3.68
BoSh3	1.01	4.04

TABLE III: Solver vs Loss

V. Conclusion

We achieved better results with the NeuralODE models than the neural network models. However, these results do not fully showcase the capabilities of NeuralODE, which we list below:

- Handling of continuous time-series models. Data can come in at arbitrary times which eases data collection. Our simulation environment provides perfect data, making it fairly easy to collect data for training.
- Memory efficiency. Our model is relatively small, not exploiting the memory efficiency of NeuralODE
- Runtime configurability. Neural ODE can be configured after training to support different accuracy, speed, and power consumption needs. In our simulations, this too was unnecessary as there is no real limiting design constraint.

In short, to fully utilize the strengths of NeuralODE, much larger complex models with real-time performance constraints are necessary. We look forward to future opportunities that will allow us to explore the potential of Neural ODEs in greater depth and scale, and to develop new techniques that can leverage the power of this innovative approach to learning dynamics.

References

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

[2] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[3] R. T. Q. Chen, "torchdiffeq," 2018.