System Identification with Neural ODEs

Evan Donovan
Boston University
donovan@bu.edu

Parth Ghayal
Boston University
parthg@bu.edu

William Harrington
Boston University
harriw5@bu.edu

Mark Tracy
Boston University
mrktracy@bu.edu

Abstract

Liquid time constant (LTC) networks are a relatively new class of neural ordinary differential equations (ODEs) that have previously shown great promise in time series prediction and system identification. Compared to existing methods, LTC networks are reported to be particularly adept at out-of-distribution testing, and they show impressive data efficiency. Our basic task is to compare the accuracy, adaptability, and computational efficiency between a base case neural ODE and an LTC network that is intended to increase adaptability and expressiveness of the network. We undertake this in the context of system identification by simulating the three-body problem from Newtonian mechanics and attempting to use both a vanilla neural ODE and an LTC network to capture the system dynamics. Results of the present study indicate little or no advantage to LTC networks in system identification or adaptability, contrary to previous findings, while also demonstrating increased computational costs of using LTC networks compared to vanilla neural ODEs. Limitations of the present study are discussed.

1. Introduction

We perform a preliminary exploration of the adaptability of LTC networks compared to a vanilla neural ODE on a system identification task. Before presenting a mathematical formulation of the problem at hand, some background is warranted. We begin by introducing the topic of system identification and providing an overview of the particular system that is the focus of the present study. We then introduce the topics of neural ODEs and LTC networks in particular.

1.1. System Identification

System identification is the process of building models of dynamic systems using data. Practically, system identification focuses on creating simulations of real-world phenomena in a data-driven manner. The focus of this study is on the development of computational models through implementation of neural ODE networks.

The system we focus on is a gravitational model from Newtonian mechanics known as the three-body problem. The system comprises three masses mutually exerting attractive forces on each other according to equations 1 and 2 below. The behavior of the system depends on initial conditions, as well as the value of the gravitational constant G. The system is chaotic in general, meaning that a slight deviation in initial conditions may lead to extreme divergence in eventual system behavior.

$$\mathbf{F}_{i,j} = \frac{G m_i m_j}{||\mathbf{x}_j - \mathbf{x}_i||^3}(\mathbf{x}_j - \mathbf{x}_i) \quad \forall i, j \in \{1, 2, 3\} \tag{1}$$

$$\ddot{\mathbf{x}}_i = G \sum_{j=1, j\neq i}^{3} \frac{m_j}{||\mathbf{x}_j - \mathbf{x}_i||^3}(\mathbf{x}_j - \mathbf{x}_i) \quad \forall i \in \{1, 2, 3\} \tag{2}$$

1.2. Neural ODEs

The paper "Neural Ordinary Differential Equations" (Chen et al., 2018) [6] introduced the notion of using a deep neural network to learn the derivative of a hidden state. This is in essence a continuous limit of residual networks, popularized in [9], which can be seen as a discretization or Euler approximation to neural ODE dynamics.

The dynamics of the hidden state of a neural ordinary differential equation (ODE) system, represented as $\mathbf{x}(t)$ in a D-dimensional space, i.e. $\mathbf{x}(t) \in \mathbb{R}^D$, is described by the equation

$$\frac{d\mathbf{x}(t)}{dt} = f\left(\mathbf{x}(t), \mathbf{I}(t), t, \theta\right), \tag{3}$$

Here, $\mathbf{I}(t)$ is an input function, and the function $f : \mathbb{R}^D \mapsto \mathbb{R}^D$ is implemented as a neural network parameterized by θ, allowing it to capture complex system dynamics. To compute the system's state, numerical ODE solvers are employed, and the network is trained using the adjoint method in reverse-mode automatic differentiation, which is developed in Pontryagin et al. (1962) [1] and explicated by Chen et al. (2018) [6], or using backpropagation through time as explicated by Hasani et al. (2020) [8].

This approach enables constant memory cost, adaptive computation, and parameter efficiency. It offers a novel

framework that embraces continuous-time modeling, facilitates the integration of data arriving at arbitrary intervals, and leads to the development of scalable and invertible normalizing flows.

1.3. LTC Networks

Liquid Time-Constant (LTC) networks are a particular class of neural ODE architecture introduced by Hasani et al. (2021) [8] and expounded upon in [4] and [10]. In LTC networks, the derivative of the hidden state $\mathbf{x}(t) \in \mathbb{R}^D$ at time t is represented as

$$\begin{aligned}\frac{d\mathbf{x}(t)}{dt} = &- \left[\frac{1}{\tau} + f\left(\mathbf{x}(t), \mathbf{I}(t), t, \theta\right)\right] \odot \mathbf{x}(t) \\ &+ f\left(\mathbf{x}(t), \mathbf{I}(t), t, \theta\right) \odot A\end{aligned} \tag{4}$$

In this equation, $f : \mathbb{R}^D \mapsto \mathbb{R}^D$ is a deep neural network parameterized by θ, $A \in \mathbb{R}^D$ is a vector of learnable parameters, and $\mathbf{I}(t)$ represent external inputs to the system, and τ is a hyperparameter of the system. The symbol $\odot$ represents the Hadamard product, i.e. element-wise multiplication.

This architecture has an input-dependent time-constant given by

$$\tau_{sys} = \frac{\tau}{1 + \tau f\left(\mathbf{x}(t), \mathbf{I}(t), t, \theta\right)}$$

The time constant τ_{sys} characterizes the response of the system to a step change in its input at steady state. It is the dependency of the time constant on the neural network $f(\mathbf{x}(t), \mathbf{I}(t), t, \theta)$ that lends liquid time constant networks their name. The term "liquid" refers to the adaptability of the time constant τ_{sys}, which changes in response to the input features and the network's internal state at each moment in time. This adaptability allows LTC networks to emulate a wide range of dynamic systems, making them versatile in modeling complex temporal patterns.

These networks are based on the dynamic causal models (DCMs) developed by Friston et al. (2003). [7] DCMs are physiologically-motivated descriptions for the interconnections of neurons in brain regions. The model suggests that outputs have dependencies on both external and internal stimuli and that these dependencies have dynamic time-variance such that state observations change based on a time constant.

1.4. Mathematical formulation of the problem

Suppose we are given training set of N pairs of initial conditions and subsequent trajectories

$$S = \{(\mathbf{x}_1(t_1^0), y_1), ..., (\mathbf{x}_N(t_N^0), y_N)\},$$

where

$$y_i = \{(t_i^0, \mathbf{x}_i(t_i^0)), ..., (t_i^m, \mathbf{x}_i(t_i^m))\}.$$

In other words, y_i is the i^{th} trajectory, t_i^k is the k^{th} time step in the i^{th} trajectory, and $\mathbf{x}_i(t_i^k) \in \mathbb{R}^D$ is the state of the system at the k^{th} time step in the i^{th} trajectory. Our task is to generate a predicted trajectory $\hat{y_i}$ for an input $\mathbf{x}_i(t_i^0)$.

In our particular setup of the three-body problem, the state vector $\mathbf{x}(t) \in \mathbb{R}^{12}$ is a 12-dimensional vector, with features representing three 2-dimensional positions and three 2-dimensional velocities.

A neural ODE (either a vanilla neural ODE or an LTC) approximates the derivative of the state as such:

$$\frac{d\mathbf{x}(t)}{dt} = f(\mathbf{x}(t), t, \theta),$$

where $f : \mathbb{R}^D \mapsto \mathbb{R}^D$ is a neural network and θ are its parameters. Note that in contrast to equation 3, there is no external input function $\mathbf{I}(t)$.

To solve for a predicted trajectory y, one could solve the definite integral

$$\mathbf{x}(t_i^k) = \int_{t_i^0}^{t_i^k} f(\mathbf{x}(t), t, \theta)dt + \mathbf{x}(t_i^0)$$

for each t_i^k in the trajectory. However, since an analytical solution is not available, we employ a numerical ODE solver. Namely,

$$\mathbf{x}(t_i^k) = \text{ODESolve}(\mathbf{x}(t_i^0), t_i^0, t_i^k, \theta, f)$$

This evaluation for each time point constitutes the forward pass. The loss is calculated using the mean-squared error between the trajectory thus obtained and the known trajectory output by our simulation engine. Finally, we backpropagate to optimize the values of the learnable parameters. Backpropagation is achieved using the adjoint sensitivity method for the vanilla neural ODE (see [6]), and using a custom backpropagation through time methodology as explicated in [8] for the LTC network.

2. Methods

This study was essentially performed in two phases: first, creation of the simulation engine, and second, use of the simulated trajectories to train and test the performance of a vanilla neural ODE and an LTC network.

First, we created a three-body problem simulation engine, which we used to generate a set of trajectories as described in section 1.4. Initial conditions were designed for the trajectories such that the resultant dataset would be information-rich in order that our models could learn efficiently from fewer samples. Numerical issues in the simulation engine necessitated pruning of the trajectories in order to ensure that our models were trained only on realistic trajectories. Details are provided in the first subsection below.

We used the simulation engine to produce two datasets: one with the gravitational constant $G = 1$ and the other with gravitational constant $G = 0.5$. Next, we utilized the trajectories generated by our simulation engine in order to test the adaptability of two models. The first model is a vanilla neural ODE, specifically chosen for its simplicity; the second is an LTC network. Adaptability of each network was tested as follows:

1. Train model on $G = 1$ dataset.
2. Train model on $G = 0.5$ dataset.
3. Take model from step (1) and retrain on $G = 0.5$ dataset.
4. Compare performance and learning curves between step (2) and (3) above.

This results in six total runs across the two models, as summarized in Figure 2.1 below.

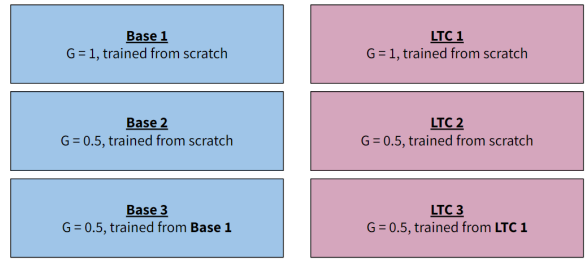

Figure 2.1. Summary of the six runs performed to assess adaptability and performance of each model. The "Base" model is a vanilla neural ODE, while the "LTC" model is an LTC network. In each case, a model instance was trained on our simulated $G = 1$ dataset, another model instance was trained on our simulated $G = 0.5$ dataset, and then the first model instance was re-trained on the $G = 0.5$ dataset, enabling us to compare performance with and without transfer learning.

2.1. Simulating the System

We created our own synthetic dataset on which to train and test the models. This was accomplished by randomly generating a set of initial positions and velocities, and then simulating how these initial conditions should evolve over time so that the neural networks could be trained to do the same. Synthetic generation allowed us to control every aspect of the experiment, including the gravitational constant, G.

2.1.1 Initial Conditions

We chose to restrict our three-body problem to the 2-dimensional, equal mass case. For one dataset, the gravitational constant, G was set equal to 1, for the other G was set equal to 0.5.

The initial positions are generated such that each initial position was generalizable in terms of scaling, rotation, and mirroring, with the coordinate system placed at the system's barycenter (collective center of mass). Each fully unique set of initial positions is generated according to the scheme outlined in Figure 2.2, which is derived from a similar initialization scheme used by Breen et al. [3]. This scheme allows for the generation of the 2-dimensional positions for all three points from just two random variables (angle and radius of particle 2), resulting in a much richer dataset per generated sample than simply generating three random points.

Additionally, each point was given an initial velocity with a magnitude and direction both pulled from a uniform distribution over [0,1]. Starting each particle off with zero initial velocity would have preserved the generalizability from the positional initialization, but this low-energy system resulted a much higher frequency of very close encounters and thus large computational errors, as explored in Section 2.1.3. Increasing the the maximum velocity magnitude from zero to one increased the post-filter acceptance rate from 5% to 60%, a significant time savings. Adding an initial velocity meant that we had to enforce a net-zero initial linear momentum constraint so that the barycenter remained stationary at the origin. The initial velocity of particle 3 was defined so as to satisfy this constraint.

Finally, with the gravitational constant, the masses, the maximum radius, and the maximum speed all set to the value of one, we ensured that the problem was well-scaled and thus ready to be passed into an ODE solver.

2.1.2 Simulation

In order to create the synthetic dataset, we simulated the resulting trajectories as an initial value problem using an off-the-shelf ODE solver. After some experimentation, the function RK45 was found to best balance speed and accuracy, although future experiments may benefit from using an arbitrary-precision N-body solver such as `brutus` [2] for dataset generation.

From this simulation, the initial particle positions and velocities serve as inputs to the neural network and the collection of particle positions and velocities at all time-steps serve as labels. The simulation time was chosen by hand. For numerical methods, computational errors accumulate over time, so excessively long simulations, especially those with many encounters, will tend to diverge significantly from reality. After some informal experimentation, a non-dimensionalized simulation time of 2 was found to be just long enough for a few 'interesting' encounters. The simulation time step of $dt = 0.01$ was chosen so that we could

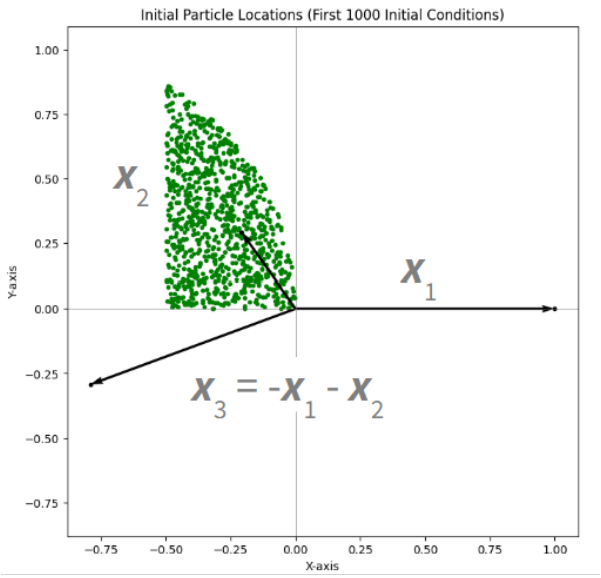

Figure 2.2. A visual depiction of the initial position generation performed as part of the simulation engine. The procedure was as follows: define $\mathbf{x}_1$ as $(1, 0)$, randomly generate $\mathbf{x}_2$ within the green region, and calculate $\mathbf{x}_3$ by vector addition. This is equivalent to randomly generating three points, placing the origin at their barycenter, and defining the x-axis as the direction of the furthest point ($\mathbf{x}_1$) and the y-axis in the direction of the closest point ($\mathbf{x}_2$), and scaling so that $r_1 = 1$.

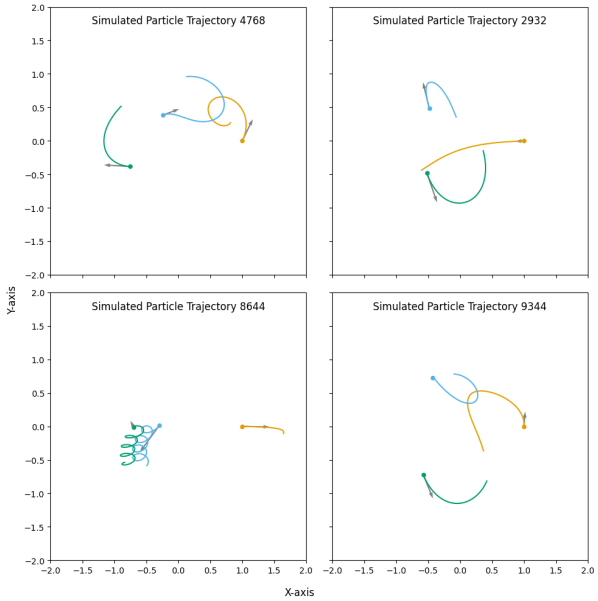

Figure 2.3. Example post-filter trajectories from the G=1 dataset. The dots represent the initial positions and the gray arrows the initial velocity vectors. Note that even after filtering-out many very close encounters, a number of 'interesting' cases—where particles interact closely and repeatedly, such as in the bottom left image—still remain in the dataset.

generate results quickly and iterate on our process. However, the success of Breen et al. with $dt = 10^{-8}$ [3] suggests that our simulation may be excessively coarse.

It should be emphasized that the 3-body system is an inherently chaotic one; small changes in the initial state of the system can result in end states that diverge significantly. Close encounters drive this chaos, with small changes to the encounter time and location as well as in the velocities of the particles involved producing vastly different resulting trajectories [3].

2.1.3 Filtering

The RK45 ODE solver is an adaptive step size function; when particles get very close to each other, the rapid change in acceleration prompts the solver to reduce the step size so that it approaches zero. These computations with extremely small numbers sometimes result in early termination, but in other cases simply result in unrealistic results (Figure 2.4).

A variety of filters were explored to avoid feeding these non-physical cases into the neural networks' training, validation, and testing sets. These filters included a minimum allowable separation distance, a maximum allowable speed, and a maximum allowable energy conservation error. In the end, only the energy filter proved necessary.

The energy error filter consists of a calculation of the total mechanical energy in the system at the initial conditions (Eq. 5). If the total mechanical energy of the system ever deviates from this initial energy by more than a set relative error (Eq. 6), the initial conditions and the simulation are removed from the dataset. The results of the energy filter are summarized in Figure 2.5. This allows us to use a fast but inaccurate ODE solver such as RK45 without having to worry about training our network on unrealistic trajectories. It also serves to filter out any initial conditions which result in trajectories deemed "hard to solve," allowing our networks to focus on the general case. Further experimentation with more accurate ODE solvers would be needed to validate this decision.

$$
\begin{aligned}
E(t) &= \sum_{i=1}^{3} \left(E_i^{\text{kinetic}}(t) + E_i^{\text{potential}}(t) \right) \\
&= \frac{1}{2} m_1 ||\mathbf{v}_1(t)||^2 + \frac{1}{2} m_2 ||\mathbf{v}_2(t)||^2 + \frac{1}{2} m_3 ||\mathbf{v}_3(t)||^2 \\
&\quad - \frac{G m_1 m_2}{||\mathbf{x}_1(t) - \mathbf{x}_2(t)||} - \frac{G m_2 m_3}{||\mathbf{x}_2(t) - \mathbf{x}_3(t)||} \\
&\quad - \frac{G m_3 m_1}{||\mathbf{x}_3(t) - \mathbf{x}_1(t)||}
\end{aligned}
\tag{5}
$$

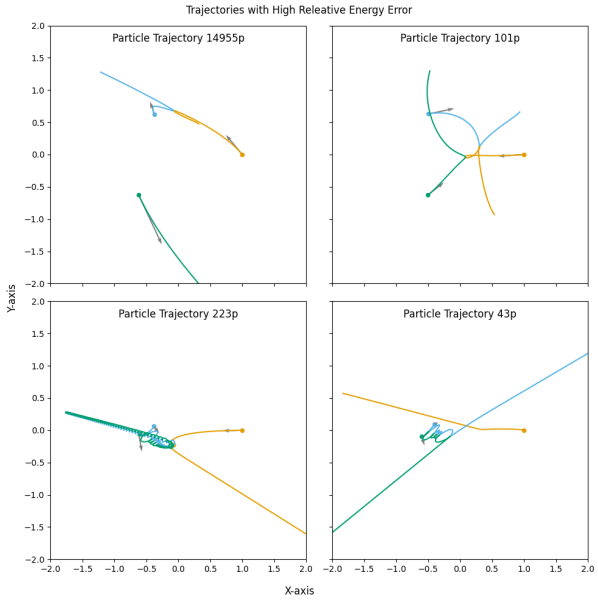

Figure 2.4. Examples of pre-filtered trajectories with high energy-conservation error. Note that in the top-left image, the particles are on a collision course from the beginning and appear to bounce off one anther multiple times after experiencing tight, looping encounters. In the bottom-left, two particles start off close to each other with near-zero velocity and they enter a decaying spiral with many encounters. In the bottom-right, two particles appear to bounce off one another repeatedly until they finally gain a huge amount of energy and fly off on near-linear trajectories.

$$\delta E(t) = \left| \frac{E(t) - E(t_0)}{E(t_0)} \right| \tag{6}$$

2.2. Training the Models

In this section we describe the details of the two neural network models that we trained to perform system identification.

The first model is the base case, or vanilla, neural ODE. The model consists of three fully-connected linear layers, incorporating rectified linear unit (ReLU) activations to capture the nonlinear behavior of the system. The network configuration was chosen for its simplicity in order to be contrasted with the more complex LTC network model. The network configuration is shown in Figure 2.6 below.

We used He initialization and optimized parameters using an Adam optimizer with a learning rate set to 0.001 over 100 epochs, minimizing the mean squared error loss between our predicted trajectories and the results from our simulation engine, as described in section 1.4. We utilized the 'odeint' function with 'dopri5' method from the 'torchdiffeq' library. The 'dopri5' method utilizes adaptive step-size Runge-Kutta integration. Backpropagation through the ODE solver is handled automatically by the 'torchdiffeq' library. We utilized learning rate scheduling with exponential decay rate (gamma = 0.98) to enhance training stability.

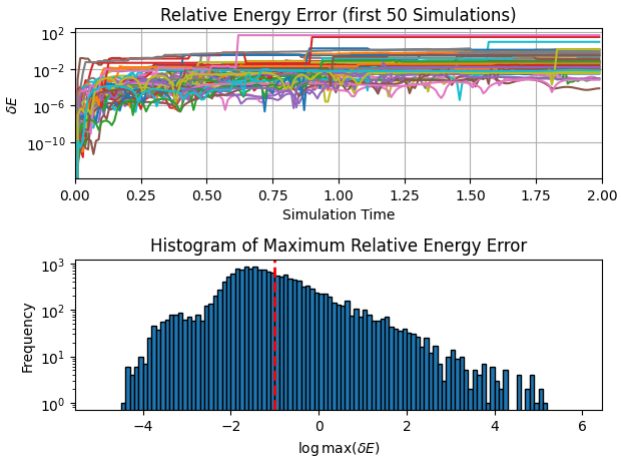

Figure 2.5. Relative energy error vs. Simulation time and and histogram of the maximum relative energy error in each simulation. The sudden spikes in relative error visible in the upper plot correspond with very close encounters. Note the histogram's x-axis is a log scale. The red dashed line represents the 10% relative error used as the limit in filtering these results.

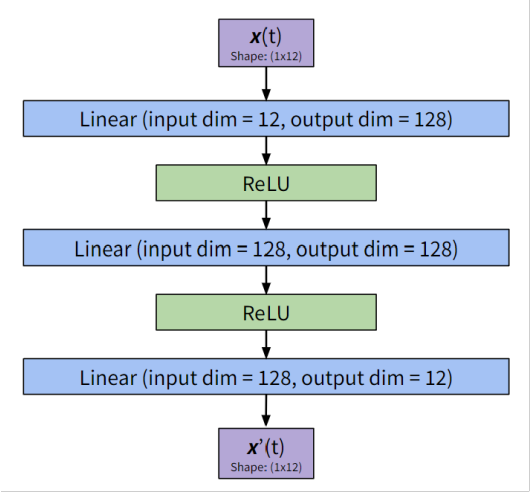

Figure 2.6. Base case, or vanilla, neural ODE model configuration. The model consists of three fully-connected linear layers with ReLU activation functions.

The second model we trained is the liquid time constant (LTC) network model. For this model, we leveraged the 'ncps.torch' library created by the authors of the paper "Liquid Time Constant Networks" [8]. We created a custom dataset and data loader to integrate the appropriate classes and methods into our workflow. We utilized the automatic neural circuit policy ('AutoNCP') function from that library to generate the connection pattern with 128 total units and

12 "motor neurons." This function configures the functional form of the function f in equation 4 by dictating which terms may interact in the system of coupled ODEs. Incorporation of the 'ncps.torch' package required use of Pytorch Lightning. The LTC network model underwent a 400-epoch training regimen with periodic model checkpoints and comprehensive logging mechanisms. Adaptive stopping was used to terminate training if validation loss increased consistently over successive batches. The Adam optimizer was utilized. A learning rate of 0.01 was found to be most effective.

3. Results

First, we present results indicative of the performance of our models on the system identification task. Next, we present and discuss results from the adaptability experiment.

3.1. Comparative Trajectory Results

In order to establish the baseline effectiveness of our models, we plot trajectories generated by our simulation engine against the predictions of our models. The base case achieves a mean-squared validation error of approximately 0.1, whereas the LTC model achieves a mean-squared error of approximately 0.2. Considering that there are 12 components of the state vector, each potentially varying on the range $[-\infty, \infty]$, we consider such errors relatively successful. By visual inspection, the trajectories appear well-approximated, though there is room for improvement with more time and hyperparameter tuning. An example comparison of our simulated trajectories to the predictions of our model is shown in Figure 3.1.

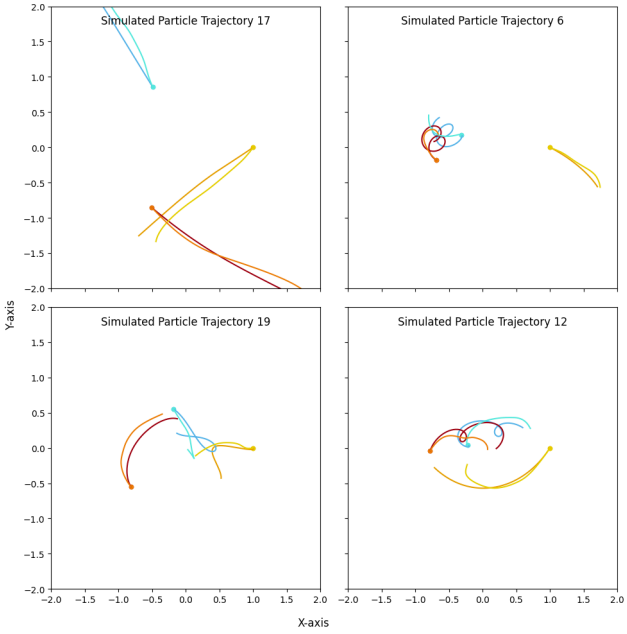

Figure 3.1. An example of simulated trajectories (darker colors) with predicted trajectories (lighter colors) from the vanilla neural ODE model. While there is room for improvement with further hyperparameter tuning, clearly the model is able to approximate the behavior of the system at a level much better than chance.

3.2. Adaptability Results

The results of the adaptability analysis are shown in Figure 3.2. Results are discussed more thoroughly in the Discussion section, but we highlight a few observations. First, in all cases shown the validation loss approximately scales with the training loss, showing that the models do not overfit despite the moderate number of training examples (1185 trajectories). Second, the base case appears to exhibit more efficient transfer of learning from the $G = 1$ case to the $G = 0.5$ case than does the LTC model. With the base case model, the training and validation loss are approximately optimal very shortly after training begins on the $G = 0.5$ case with transfer learning. By contrast, in the LTC network case the transfer of learning is less efficient, and it takes many more epochs for the model to improve performance on the new dataset.

3.3. Sample Efficiency of Transfer Learning

To further compare the adaptability of the base case and LTC network models, we assessed the sample efficiency of transfer learning for both models. To perform this analysis, we followed the following procedure for each of the base case and the LTC network, and for each value of N (we used $N \in \{100, 500, 829\}$):

1. Train the model on the full dataset with $G = 1$.
2. Train the model on N sample trajectories with $G = 0.5$ for 20 epochs.
3. Test the accuracy of the resultant model on trajectories with $G = 0.5$ that were held out from training in step 2.

This procedure, performed using either the base case or LTC network model, generates one point on the corresponding curve shown in figure 3.3.

We see from this procedure that although the LTC network model is less performant than the base case model overall, it may be more data efficient in its transfer learning. Its performance is approximately equivalent when trained on $N = 100$, $N = 500$, or $N = 829$ trajectories, whereas the base case only achieves its best performance when trained on all 829 samples. However, the relatively high loss in all three cases for the LTC network model indicates that 20 epochs may not be sufficient for the LTC model to learn in any of the $N = 100$, $N = 500$, or $N = 829$ cases. Given more time, this experiment could be repeated

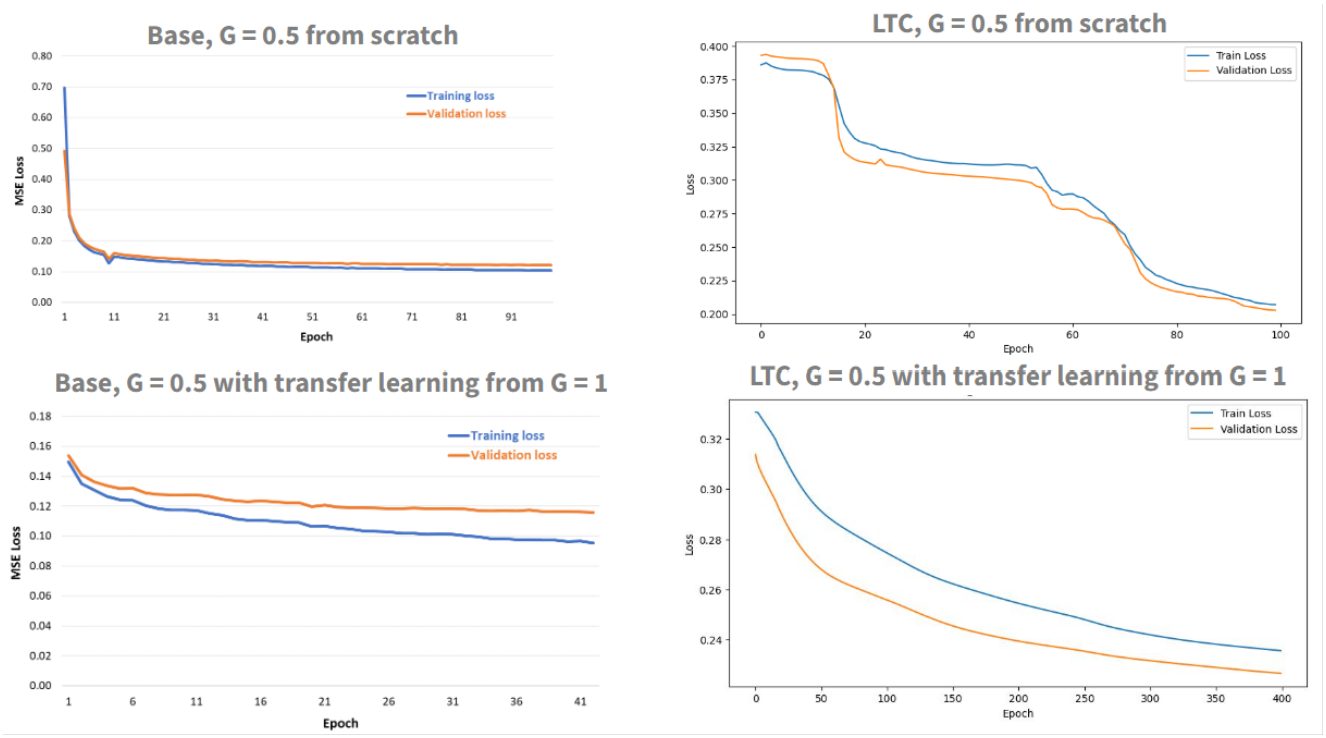

Figure 3.2. Learning curves showing comparative adaptability of the base case and LTC network models. The upper diagram in each column shows the learning curve resulting from training on the $G = 0.5$ dataset from scratch (without transfer learning), while the lower diagram in each column shows the learning curve resulting from training on the $G = 0.5$ dataset from with transfer learning from the $G = 1$ dataset. The base case model exhibits more effective transfer learning from the $G = 1$ dataset to the $G = 0.5$ dataset.

with longer training times to enhance this measure of data efficiency.

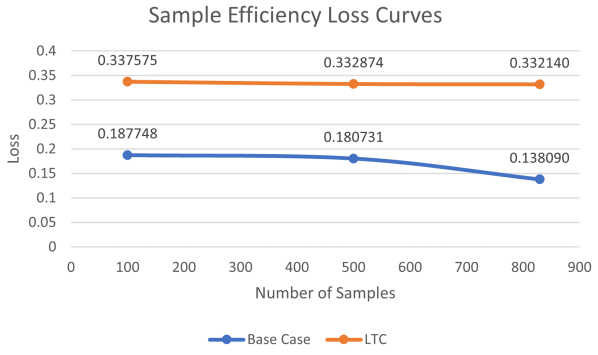

Figure 3.3. Sample efficiency curves for the base case and LTC network case. The base case network requires a full training set to achieve optimal performance, whereas the LTC network achieves consistent performance with smaller sample sizes. However, we note that the lower performance of the LTC compared to the base case calls into question the usefulness of the comparison.

4. Discussion

Contrary to our expectations, our results show that the base case, vanilla neural ODE model exhibited superior transfer learning capabilities when compared to the LTC network. It is crucial to acknowledge that these types of networks are still under active investigation within the research community, thus rendering our results as preliminary and subject to further validation.

Comparing the base case to the LTC model, we see that the base case performance was better than the LTC network's performance in terms of the validation loss, a measure of accuracy of the network's predictions, and in terms of adaptability to out-of-distribution data. It should be noted that network performance is contingent upon a multitude of factors, and achieving optimal functionality with LTC networks may necessitate extensive fine-tuning of the underlying parameters.

The study of LTC networks, as with other ODE-based neural networks, demands significant computational resources. These networks are inherently resource-intensive, primarily due to the sequential progression required for forward trajectory propagation. This characteristic inherently restricts parallelization. We found the LTC network to be

about half as well-performing as the vanilla neural ODE, while also requiring about twice the training time. This computational burden underscores the practical challenges of deploying LTC networks, potentially limiting their application to specific use cases where such computational resources are justified.

The LTC network's nearly stepwise learning trajectory poses a question for future investigations. The model exhibited long periods of stasis followed by punctuated leaps in learning; and when transfer learning was applied from the $G = 1$ case, the trajectory was radically changed, exhibiting steady learning over time. It remains to be seen from further experimentation whether this is a replicable result.

The accelerated time frame of the present study posed a significant challenge. Further experimentation would require more time to be invested in hyperparameter tuning of the models in order to improve performance. Additionally, the development of more nuanced metrics for adaptability could provide deeper insights into the functional dynamics of these networks. Further research should aim to elucidate the conditions under which LTC networks might outperform traditional neural ODE models and establish a clearer understanding of their potential advantages.

A concrete direction for future research, which we developed theoretically but were unable to implement due to time constraints, is presented in Appendix A.

5. Contributions and Code Repository

In the table below, we outline the division of labor for the project. All members collectively agreed on this division of labor and have agreed to help each other as needed. Additionally, anything outside of the points mentioned here was handled by the team as a whole.

Task	**Assigned to**
Dataset generation	Billy
Base case neural ODE	Mark, Parth
LTC network	Mark, Parth
Post processing	Evan
Parameters tuning	Evan
Slides	All
Report	All

Table 5.1. Division of labor for the project.

The GitHub repository associated with this project can be found here: https://github.com/mrktracy/NeuralODEs

References

[1] Richard Bellman, L. S. Pontryagin, V. G. B. Boltyanskii, R. V. Gamkrelidze, and E. F. Mischenko. Mathematical theory of optimal processes. In *Mathmatics of Computation*, 1962.

[2] Tjarda Boekholt and Simon Portegies Zwart. On the reliability of n-body simulations. *Computational Astrophysics and Cosmology*, 2, 2015.

[3] Philip G Breen, Christopher N Foley, Tjarda Boekholt, and Simon Portegies Zwart. Newton versus the machine: solving the chaotic three-body problem using deep neural networks. *Monthly Notices of the Royal Astronomical Society*, 494(2):2465–2470, 04 2020.

[4] M. Chahine et al. Robust flight navigation out of distribution with liquid neural networks. *Sci. Robot.*, 8(77):eadc8892, Apr 2023.

[5] Mathieu Chalvidal, Matthew Ricci, Rufin VanRullen, and Thomas Serre. Neural optimal control for representation learning. *ArXiv*, abs/2006.09545, 2020.

[6] Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David Kristjanson Duvenaud. Neural ordinary differential equations. In *Neural Information Processing Systems*, 2018.

[7] K. J. Friston, L. Harrison, and W. Penny. Dynamic causal modelling. *NeuroImage*, 19(4):1273–1302, Aug 2003.

[8] R. Hasani, M. Lechner, A. Amini, D. Rus, and R. Grosu. Liquid time-constant networks. *AAAI*, 35(9):7657–7666, May 2021.

[9] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition, Dec 2015.

[10] C. Vorbach, R. Hasani, A. Amini, M. Lechner, and D. Rus. Causal navigation by continuous-time neural networks. *AAAI*, 35(9):7657–7666, May 2021.

[11] Han Zhang, Xi Gao, Jacob Unterman, and Tom Arodz. Approximation capabilities of neural odes and invertible residual networks. In *International Conference on Machine Learning*, 2019.

Appendix A. Direction for Future Research

We introduce in this appendix a direction for future research, which we investigated theoretically but were unable to implement due to time constraints.

The paper "Go with the Flow: Adaptive Control for Neural ODEs" by Chalvidal et al. (2021) [5] introduces a method they call N-CODE for increasing the expressiveness of vanilla neural ODEs, which on their own suffer from known limitations in expressiveness [11]. The idea is to allow the parameters θ in the usual neural ODE setting to vary with time in a manner that allows for both open-loop and closed-loop control of the state vector $\mathbf{x}(t)$. We can make this precise as follows:

In the usual neural ODE setting, we have:

$$\frac{d\mathbf{x}(t)}{dt} = f(\mathbf{x}(t), t, \theta)$$

We create an augmented state $z(t) = [\mathbf{x}(t), \theta(t)]$ with the dynamics

$$\frac{dz(t)}{dt} = h(z(t), t),$$

which may be decomposed into the following coupled system:

$$\begin{aligned} \frac{d\mathbf{x}(t)}{dt} &= f(\mathbf{x}(t), t, \theta(t)) \\ \frac{d\theta(t)}{dt} &= g(\mathbf{x}(t), \theta(t), t) \\ z(0) &= [x(0), \theta(0)] = [x(0), \gamma_\mu(x(0))], \end{aligned}$$

where γ_μ is a learnable function with parameters μ that maps the initial state $x(0)$ for a given trajectory to an initial parameter $\theta(0)$.

The function γ_μ is akin to open-loop control of the system based on the initial condition, whereas the coupling between the evolution of $\theta(t)$ and $\mathbf{x}(t)$ allows for a form of closed-loop control. The authors prove that this augmentation increases the expressiveness of neural ODEs and demonstrate its superior performance on a memorization task that requires quick adaptation to input data.

The original LTC formulation is provided above as equation 4, but is repeated here for convenience, omitting the input term $\mathbf{I}(t)$, since it is not relevant to our system identification problem (there are no external inputs to the system):

$$\frac{d\mathbf{x}(t)}{dt} = -\left[\frac{1}{\tau} + f\left(\mathbf{x}(t), t, \theta\right)\right] \odot \mathbf{x}(t) + f\left(\mathbf{x}(t), t, \theta\right) \odot A$$

Clearly this is a neural ODE of the general form

$$\frac{d\mathbf{x}(t)}{dt} = h(\mathbf{x}(t), t, \theta'),$$

where the parameter $\theta' = [\theta, A]$.

Our proposed approach is to apply the methodology of Chalvidal et al. (2021) [5] in allowing the vector A to evolve with time in a manner that is coupled to the state vector $\mathbf{x}(t)$:

$$\begin{aligned} \frac{d\mathbf{x}(t)}{dt} &= -\left[\frac{1}{\tau} + f(\mathbf{x}(t), t, \theta)\right] \odot \mathbf{x}(t) + f(\mathbf{x}(t), t, \theta) \odot A(t) \\ \frac{dA(t)}{dt} &= g(\mathbf{x}(t), A(t), t) \\ \{x(0), A(0)\} &= \{x(0), \gamma_\mu(x(0))\} \end{aligned}$$

The particular form of LTC networks lends itself well to this form of control, since $A(t)$ appears linearly in the dynamics of $\mathbf{x}(t)$.

There are many potential avenues one can explore with this hybrid formulation, which fell beyond the scope of this project. At a high level, though, having two coupled deep learning models operating on overlapping sets of inputs may

allow for more nuanced control of the system. While the neural network $f(\mathbf{x}(t), t, \theta)$ might capture the primary dynamics, $g(\mathbf{x}(t), A(t), t)$ can introduce corrective or complementary dynamics. This may improve the performance of the model, albeit at a computational cost. One could formally analyze the additional computational cost of this approach, which must be weighed against any advantages observed in its performance. Finally, different functional forms for g and γ_μ could be explored.