Abstract—Irregular time series data are prevalent in the real world and are challenging to model accurately with a simple recurrent neural network (RNN). As a result, a model that combines the use of ordinary differential equations (ODE) and RNN was proposed (ODE-RNN) to model irregular time series with higher accuracy, but it suffers from high computational costs. In this paper, we propose an improvement in the runtime on ODE-RNNs by using a more efficient batching strategy. Our experiments show that the new models reduce the training time of ODE-RNN significantly ranging from 2 times up to 49 times depending on the irregularity of the data while maintaining comparable accuracy. We also propose different variations of the evolver module which integrates the ODE, and test our model on real-life datasets such as MIMIC-IV. Hence, our model can scale favorably for modeling larger irregular data sets.

Index Terms—irregular time-series, neural ODE

I. INTRODUCTION

Time series models are ubiquitous in numerous applications such as predictive maintenance [11], [12], [14], financial forecasting [18], and next sentence prediction [7], [19]. While statistical time series methods such as ARIMA can outperform machine learning methods for univariate datasets [17], machine learning promises superior performance for large multivariate datasets [2]. Recurrent Neural Networks (RNN) with memory retention such as Gated Recurrent Units (GRU) or Long Short Term Memory (LSTM) have been the gold standard for temporal machine learning models. The hidden state $h_i$ of a traditional RNN cell is only updated every observation, which is not a problem if the data points are equidistant in space or time. However, this poses limitations for applications where observations are irregular or cannot be obtained periodically [10], [23]. For instance, various measurements of patients are often obtained at irregular intervals [24], so there is a need for a model that takes into account the time difference between data points. Hence, modeling irregular time series is extensively studied in recent times, and various RNN-based models have been proposed [25].

To account for irregular sampling, data points are often binned into regular intervals using aggregation. This technique has two problems: 1. The sampling interval chosen has to be larger than the smallest interval between two consecutive data points. 2. Averaging data points destroys valuable information about the correlation between sampling intervals and latent variables [3], [15]. Another solution is to include the time delta as an additional feature to the RNN input, but this will not interpolate the hidden state between observations [22]. Rubanova et al. have developed an ODE model that evolves the latent hidden states using ordinary differential equations (ODE-RNN) [22]. During the training of the model, the sequences of the dataset are randomly separated into mini-batches so that larger mini-batches can decrease training time. An ODE can be solved numerically using an adaptive step approach such as the Runge-Kutta method, or a fixed-step approach such as the Euler method. The ODE solver from their work can be used with both methods [4]. However, their approach requires using the union of time values across a mini-batch (combined time). This can drastically slow down the training time, especially with the presence of multiple non-unique time values.

The remainder of the manuscript is structured as follows: In Section II we introduce the problem statement and review existing methods, with a focus on what we call the combined time ODE-RNN model. Section III introduces our batch-efficient model that scales linearly with the sequence length. In Section IV we demonstrate our model achieves significant speedup in comparison to existing methods while maintaining accuracy. Finally, we discuss our results and implications in Section V.

II. BACKGROUND

a) Problem statement: We consider the auto-regressive problem: Given a time series $i \in \{0, \ldots, N-1\}$ of input features $x_i \in \mathbb{R}^{d_x}$ at times $t_i \in \mathbb{R}$, and given a time $t_N$, to predict $y_N \in \mathbb{R}^{d_y}$. We perform prediction at a single future point without loss of generality. We denote in bold vectors or tensors that vary across the mini-batch, to distinguish them from scalars.

b) RNN and Ordinary Differential Equations: As we reviewed, there are several ways to model an irregular time-series. Using solely RNN, one way they can be modeled is by feeding the time deltas into the RNN update function:

$$h_i = \text{RNNCell}(h_{i-1}, \Delta t, x_i),$$

though this does not define the hidden states between observations, and has been shown to perform poorly in certain tasks.

In the ODE-RNN model [5], the ODE is solved using a Neural ODE, a class of neural networks with parameters $\theta$, 

104
which defines the hidden state as a solution to an ODE initial-value problem. Given the hidden state from the previous time step $h_{t-1}$, the evolution is broken into two parts: First, the hidden state is updated temporally:

$$h'_i = \text{ODESolve}(f_\theta, h_{t-1}, (t_{i-1}, t_i)) \text{ where } f_\theta = dh/dt. \quad (2)$$

A numerical ODE solver is used to evolve the hidden state from $t_{i-1}$ to $t_i$, while learning the parameters $\theta$. Second, this hidden state $h'_i$ is updated using a standard RNN:

$$h_i = \text{RNNCell}(h'_i, x_i) \quad (3)$$

These two steps are repeated for subsequent time steps.

c) Combined time ODE-RNN model: We begin by examining how Rubanova’s [22] combined method works to demonstrate how more irregular datasets lead to increasingly slower model training. The reason for the slowdown is two-fold: The first is that while an RNN requires only a single forward step between each input sample, the ODE portion takes multiple steps that depend on both the size of the time jump $\Delta N+1 = t_{N+1} - t_N$ as well as the curvature ($df_\theta/dt$) of the hidden state. The tuning of hyperparameters allows for some trade-off between training time and accuracy, by changing either the acceptable error tolerance (for adaptive procedures such as dopri5) or by changing the fixed time jump (for fixed procedures such as forward Euler). A second reason for slower training is that it is more difficult to sidestep by tuning hyperparameters since the way this combined time method handles integration time in a mini-batch. If the mini-batch only has a single sequence or if the irregular sequence is identical across the mini-batch, the process is straightforward, as the model updates the hidden state using the RNN of sample $x_i$ and then uses ODE evolve to advance from time $t_i$ to $t_{i+1}$.

The process requires additional steps when the mini-batch has multiple sequences that are not equal in their irregular intervals. Consider the illustration shown in Figure 1(a) for a mini-batch of 3, each with 4 samples at different times as indicated by the horizontal axis. Time is evolved in unison across the mini-batch. The sequence begins with all samples at $t_0 = 0$ (black dotted line), and the RNN with $x_0$ (black circles) can be applied. Then the hidden state is evolved forward in unison across the mini-batch, as all 4 samples are integrated up to $t = 1$ (red dotted arrow). An RNN update is performed only for the first sequence (red circle), while the other two sequences are unchanged, using a mask. Then integration continues to $t = 1.5$ (pink dotted arrow), RNN is applied selectively to the third sequence (pink circle), and the process continues until the last time point is reached. The pseudocode for this is given in Algorithm 1. This process slows training: the loop iterated over the $\{t_i\}$ in the first example unique times across the mini-batch which can be much larger than the length of any single item in the mini-batch ($4$ in this example). More generally, for a batch size $k$, the number of iterations will be $O(\text{unique}_{i \in k} \{t_i\}) \ll O(k N_k)$, and slowdown is more pronounced when using a large mini-batch size $k$, or when the dataset is more irregular, and the number of unique times across the batch grows.

| Algorithm 1: Rubanova's combined time ODE-RNN |
|-----------------------------------------------|
| **Data:** Data points $\{x_i\}_{i=0..N-1}$, corresponding time differences $\{\Delta t_i\}_{i=0..N-1}$ and final jump $\{\Delta t_i\}_{i=N}$ |
| $\{t_i\}_{i=0..N} = \sum_{j=0}^{i} \Delta t_j$; $ct = \text{unique}(t_n)$; $\triangleright$ combined time across mini-batch |
| $h = 0$; |
| for $j = 0.. \text{len}(ct) - 1$ do |
| $t_{\text{cur}} = ct(j)$; $t_{\text{prev}} = ct(j - 1)$; |
| $h' =$ |
| $\begin{cases} 
    \text{ODESolve}(f_\theta, h, (t_{\text{prev}}, t_{\text{cur}})), & t_{\text{cur}} < t_{N+1} \\
    h, & \text{otherwise} 
\end{cases}$ |
| $\triangleright$ Perform ODE if needed |
| $h =$ |
| $\begin{cases} 
    \text{RNN}(h', x(t = t_{\text{cur}})), & t_{\text{cur}} \in t \\
    h', & \text{otherwise} 
\end{cases}$ |
| $\triangleright$ Update hidden state if it exists for $t_{\text{cur}}$ |
| end for |

**Result:** $h$

d) Other related work: Recent works used various strategies to improve the efficiency and accuracy of ODEs for learning irregular time series data. The second-order neural ODE optimizer (SNOpt) computes the second-order derivative gradients for the backward propagation to improve training efficiency [16]. Likewise, heavy ball ODE (HBODE) combines gradient descent with momentum with ODE, to generate a 2nd order ODE with a damping factor to accelerate training [26]. The Taylor-Lagrange Neural ODE (TL-NODE) model uses the Taylor expansion series with Lagrange form of error approximation of the Taylor series to replace the ODESolve for doing numerical integration, to improve the efficiency of the forward propagation of neural ODE [8]. The model order reduction method uses proper orthogonal decomposition to reduce the dimensions of the weight matrices of the network, and a discrete empirical interpolation Method to replace the activation functions with interpolation operations [13]. Lyapunov theory is control theory to replace the loss function of neural ODE with the Lyapunov Loss to guarantee that the ODE exponentially converge and provide adversarial robustness [21]. Skip DEQ made two improvements to the existing DEQ model by first, adding an explicit layer with some regularizations before the implicit layer to better predict the initial state, and secondly, replace the implicit layer with an ODE that runs to infinite time using adaptive ODE solvers [20]. Finally, the neural flows method does not approximate the solution of an ODE but directly learns the solutions of the ODE, so an ODE solver is not necessary [1].

These methods can be considered as extensions or substitutions of the original ODE-RNN algorithm, while our proposed method is a direct improvement in training time to the ODE-
RNN algorithm itself.

III. MODEL

Our approach is to reduce training time performance by changing how time is evolved across the batch. Instead of using the combined unique times, we allow each sample in the mini-batch to evolve independently, as shown in Figure 1(b). Using the same example input data, all sequences begin at \( t_0 = 0 \) (black dotted line), and the RNN \( (x_0) \) (black circles) is applied. Unlike the combined time approach, each sequence across the mini-batch is evolved to different points in time: \( t_1 = [1, 2, 1.5] \) (red dotted line). The RNN for the observations \( x_1 \) (red circles) can then be applied to all sequences in the batch, each sample is evolved to its next sample in time (blue), and the process continues. Full details are shown in Algorithm 2. This reduces the number of iterations of the loop from \( O(\sum_k N_k) \) in the combined time approach to \( O(N) \). As we will show, this provides significant speedup when the dataset is highly irregular. Additionally, our approach eliminates the overhead of masking the RNN updates, since they occur simultaneously across the mini-batch.

![Fig. 1. Comparison of different batching strategies of a mini-batch of 3 sequences with 4 samples in each sequence, with arrows indicating an evolver loop iteration (a) The combined time method takes the union of all the time steps in a batch and loops every step to a total of 6 iterations. (b) Our model only loops every step regardless of the time values of the time steps, and only requires 3 iterations.](image)

**Algorithm 2: Our batch efficient ODE-RNN**

- **Data**: Data points \( \{x_i\}_{i=0..N-1} \), time differences \( \{\Delta t_i\}_{i=0..N-1} \), and final time-jump \( \{\Delta t_i\}_{i=N} \)
- **Result**: \( h \), \( o \)

The combined time approach with the Torchdiffeq evolver allows for many choices in the integration method: fixed time-step methods such as forward Euler, the midpoint method or Runge-Kutta, and adaptive integrators where a relative error rate is given and the method automatically adjusts the time step to meet the estimated threshold. Keeping our objective of improving model performance, we introduce three different evolver modules, all of which use a form of forward Euler, each with a slightly different method of performing the integration. We decide not to use other adaptive integration methods such as dopri5 which is incompatible with our evolver’s strategies, or higher order methods such as Runge-Kutta which would complicate our code and obfuscate the comparison to the combined times model. Additionally, higher-order ODE methods do not always translate to more accurate models, as we will discuss in our results. Of the three methods we introduce, the first uses a fixed time step, with a varying number of steps across the mini-batch. The second uses a fixed number of steps, with varying time steps across the mini-batch. The third uses a geometrically increasing time step, with a varying number of steps across the mini-batch. Our approach is best suited for large irregular time-series datasets where the irregularity varies across mini-batch samples.

Our first model is the fixed dt method that uses a scalar value \( \Delta t \), which is held constant throughout training, to evolve all hidden states. As shown in Algorithm 3, the loop is set by the largest time to evolve in the batch, and a simple mask is used to prevent updating any hidden state once its respective \( \Delta t_i \) is reached. This ensures the rounding error is independent of the time jumps but can lead to long run-time and very deep networks when the largest time jump is much larger than \( \Delta t \). As a rule of thumb, a decent \( \Delta t \) to start with will be 1/10th of the average time steps in the dataset.

A second method uses a fixed number of iterations in the loop that is independent of the time differences in the mini-batch, shown in Algorithm 4. Here the step size \( s \) is no longer a scalar but a vector with values for each sequence in the mini-batch. This has the advantage of giving a consistent run-time and depth to the forward pass regardless of the time-jumps, though it can lead to larger rounding errors for larger valued \( s \).

The fixed dt and adaptive fixed methods have opposing trade-offs: the first can suffer from very deep networks, while the second may incur large rounding errors. We propose the
Algorithm 3: Evolver module: Fixed dt mode

**Data:** Step size $s$, Hidden state $h$ and corresponding time differences $\Delta t$

$m = \Delta t/s$; ▷ number of steps varies across mini-batch

$M = \max(m)$;

for $j = 0..M - 1$ do
  $\text{mask} = j < m$;
  $s_{\text{cur}} = \min\{s, \Delta t - j \times s\}$; ▷ Prevent overshooting target time if $s$ is too large
  $h' = f_0(h)$; ▷ learning $\frac{dh}{dt}$
  $h = h + s_{\text{cur}} \times \text{mask} \times h'$
end for

**Result:** $h$

Algorithm 4: Evolver module: adaptive fixed mode

**Data:** Number of steps $M$, Hidden state $h$ and corresponding time differences $\Delta t$

$t_i = \sum_{j=0}^{i} \Delta t_j$;

$s = \Delta t/M$; ▷ step size varies across mini-batch

for $j = 0..M - 1$ do
  $\text{mask} = j * s < \Delta t$;
  $h' = f_0(h)$; ▷ FC learning $\frac{dh}{dt}$
  $h = h + s \times \text{mask} \times h'$
end for

**Result:** $h$

third method in an attempt to balance the two. We hypothesize that there is a tradeoff in curvature in the hidden state immediately after a new measurement $t = t_i$, than there is if the state is evolved for a later time $t >> t_i$. The geometric adaptive algorithm starts with a small step size $s_0$, which is then increased using the multiplicative constant $s_i = s_{i-1} \times r$ with the growth factor $r > 1$. If the time step were to overshoot $t_{i+1}$, it is reduced to reach it exactly, and a mask is used to prevent it from being evolved further. Full details are shown in Algorithm 5. For example, using our default values $s_0 = 0.001, r = 1.5$, requires 5 steps to reach $t = 0.01, 15$ to reach $t = 1$, and only 39 steps to reach $t = 10,000$. Although a similar increase in step size for a flatter derivative is possible with the adaptive routines of the combined time method, in practice we found that method to be even slower than using the combined time method with a fixed step size.

IV. EXPERIMENTS

We evaluate our model on synthetic and public real-world datasets including synthetic sine waves, MuJoCo physics simulation, and MIMIC-IV clinical dataset. The task for all of our models is to predict the last time step given all the preceding time steps in the sequence. They are trained to minimize the MSE loss in accordance with the hyperparameters and details listed in the appendix. In the following section, we compare the training time and accuracy of these three models to a simple RNN model as well as the combined time method baseline, using these datasets. We implement the combined time method with a wrapper for torchdiffeq [5], and use Euler as the integration method and dopri5 for comparison.

A. Synthetic dataset

We follow the procedure stated by Rubanova et al. to generate 10,000 synthetic sinusoidal wave sequence data with variable frequencies and starting position, and constant amplitude with a few minor modifications [22] (see appendix). To find out how the irregularity of the data influences the training time and accuracy, we generate random sequences with 50 random time steps in each sequence. The time steps are sampled with rounding of 0.1 and 0.001 units for comparison (Figure 2), the latter yielding significantly more irregular time-series.

Our efficient ODE-RNN models achieve superior performance in terms of run time and accuracy against the baseline combined time method and simple RNN models (Figure 3). Regarding the different modes of our model, the adaptive fixed mode gives the least mean squared error (MSE) for time values rounded to the nearest 0.001 units, while all models result in similar MSEs for rounding of 0.1 units. In the more irregular 0.001 dataset, our model achieves a significant accuracy improvement compared to the combined time method and simple RNN. Moreover, the combined time method also attains significant improvement in accuracy over simple RNN. In addition, we observe that the dopri5 solver offers only minimal improvements in accuracy for the combined time method model as demonstrated in previous research [1], while

1Our code is available on github https://github.com/craterlabs/improved-batching-ode
using greater tolerance parameters to make training feasible (Table I).

Our method achieves a speedup of 49 times against the combined time method using the Euler solver for the 0.001 units dataset, and achieves a marginal speedup for the 0.1 units dataset (Figure 3 and Table II). Since the number of unique time steps in a batch with a rounding of 0.001 units is much greater than for a rounding of 0.1 units, more iterations are required, thus resulting in a tremendous increase in training time per epoch. For the combined time method models, we use a larger learning rate of 0.01 as in Rubanova et al. to reduce the number of epochs necessary, so that the training can be feasibly performed. Using dopri5 solver for the combined time method model results in approximately 4 times longer training runtime than the Euler solver while achieving no significant improvements as demonstrated previously [1] (Table II).

B. MuJoCo physics simulation

MuJoCo is a physics engine containing a module named "Hopper" for generating the physical dynamics of an imaginary frog-like organism. Following the previously reported methods exactly [1], [22], 10,000 sequences of the "hopper" with 100 time steps and 14 features in each step are generated. Subsequently, we randomly sample 50% and 10% of the time steps for performing comparison experiments and use them to train our model.

Our model attains better performance in terms of accuracy against the combined time ODE-RNN and simple RNN baseline models in a similar fashion to the synthetic datasets. Moreover, the different methods of our model perform similarly, and the model achieves maximum performance gain for the 10% sampled dataset, which has higher irregularity than the 50% sampled dataset (Table I). In addition, our implementation results in at least two times faster training runtime than the combined time method (Table II).

C. MIMIC-IV clinical dataset

The MIMIC-IV dataset contains clinical data from more than 40,000 intensive care unit patients in Beth Israel Dea-
Acknowledgements

We thank Seyed-Parsa Hojjat, María C. Rodríguez-Liñán and Khalid Ali for useful discussions and feedback, and Naushen Fatma for contributing to the combined times wrapper code. We also thank David Duvenaud and Yulia Rubanova for the helpful discussion at the onset of our project.

REFERENCES

[1] Marin Biloš, Johanna Sommer, Syama Sundar Rangapuram, Tim Januschowski, and Stephan Günnemann. Neural flows: Efficient alternative to neural ODEs. Advances in Neural Information Processing Systems, 2021.

[2] Vitor Cerqueira, Luis Torgo, and Carlos Soares. Machine learning vs statistical methods for time series forecasting: Size matters. arXiv preprint arXiv:1909.13316, 2019.

[3] Zhengping Che, Sanjay Purushotham, Kyunghyun Cho, David Sontag, and Yan Liu. Recurrent neural networks for multivariate time series with missing values. Scientific reports, 8(1):1–12, 2018.

[4] Ricky T. Q. Chen, Brandon Amos, and Maximilian Nickel. Learning neural event functions for ordinary differential equations. International Conference on Learning Representations, 2021.

[5] Ricky T. Q. Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, Advances in Neural Information Processing Systems, volume 31. Curran Associates, Inc., 2018.

[6] Edward De Brouwer, Jaak Simm, Adam Arany, and Yves Moreau. Gruode-bayes: Continuous modeling of sporadically-observed time series. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett, editors, Advances in Neural Information Processing Systems, volume 32. Curran Associates, Inc., 2019.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805, 2018.

[8] Franck Djeumou, Cyrus Neary, Eric Goubault, Sylvie Putot, and Ufuk Topcu. Taylor-lagrange neural ordinary differential equations: Towards fast training and evaluation of neural odes. arXiv preprint arXiv:2201.05715, 2022.

[9] Alistair Johnson, Lucas Bulgarelli, Tom Pollard, Steven Horng, Leo Anthony Celi, and Roger Mark. Mimic-iv, 2021.

[10] Alistair EW Johnson, Tom J Pollard, Lu Shen, Li-wei H Lehman, Mengling Feng, Mohammad Ghassemi, Benjamin Moody, Peter Szolovits, Leo Anthony Celi, and Roger G Mark. Mimic-iii, a freely accessible critical care database. Scientific data, 3(1):1–9, 2016.
[11] Ameeth Kanawaday and Aditya Sane. Machine learning for predictive maintenance of industrial machines using iot sensor data. In 2017 8th IEEE International Conference on Software Engineering and Service Science (ICESS), pages 87–90, 2017.

[12] Kahiomba Sonia Kiangala and Zenghui Wang. An effective predictive maintenance framework for conveyor motors using dual time-series imaging and convolutional neural network in an industry 4.0 environment. IEEE Access, 8:121033–121049, 2020.

[13] Mikko Lehtimäki, Lassi Paanonen, and Marja-Leena Linne. Accelerating neural odes using model order reduction. arXiv preprint arXiv:2105.14070, 2021.

[14] Chin-Yi Lin, Yu-Ming Hsieh, Hsien-Cheng Huang, and Muhammad Adnan. Time series prediction algorithm for intelligent predictive maintenance. IEEE Robotics and Automation Letters, 4(3):2807–2814, 2019.

[15] Zachary C Lipton, David Kale, and Randall Wetzel. Directly modeling missing data in sequences with rnns: Improved classification of clinical time series. In Machine learning for healthcare conference, pages 253–270, PMLR, 2016.

[16] Guan-Horng Liu, Tianrong Chen, and Evangelos Theodorou. Second-order neural ode optimizer. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, Advances in Neural Information Processing Systems, volume 34, pages 25267–25279. Curran Associates, Inc., 2021.

[17] Spyros Makridakis, Evangelos Spiliotis, and Vassilios Assimakopoulos. Statistical and machine learning forecasting methods: Concerns and ways forward. PLOS ONE, 13. 03 2018.

[18] Ricardo P Masini, Marcelo C Medeiros, and Eduardo F Mendes. Machine learning advances for time series forecasting. Journal of Economic Surveys, 2021.

[19] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. arXiv preprint arXiv:1301.3781, 2013.

[20] Avik Pal, Alan Edelman, and Christopher Rackauckas. Mixing implicit and explicit deep learning with skip deqs and infinite time neural odes (continuous deqs). arXiv preprint arXiv:2201.12240, 2022.

[21] Ivan Dario Jimenez Rodriguez, Aaron D Ames, and Yisong Yue. Lyapunov framework for training neural odes. arXiv preprint arXiv:2202.02526, 2022.

[22] Yulia Rubanova, Ricky T. Q. Chen, and David K Duvenaud. Latent ordinary differential equations for irregularly-sampled time series. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d’Alché-Buc, E. Fox, and R. Garnett, editors, Advances in Neural Information Processing Systems, volume 32. Curran Associates, Inc., 2019.

[23] Mohammed Saeed, Mauricio Villarroel, Andrew T Reisner, Gari Clifford, Li-Wei Lehman, George Moody, Thomas Heldt, Tin H Kyaw, Benjamin Moody, and Roger G Mark. Multiparameter intelligent monitoring in intensive care ii (mimic-ii): a public-access intensive care unit database. Critical care medicine, 39(5):952, 2011.

[24] Ikaro Silva, George Moody, Daniel J Scott, Leo A Celi, and Roger G Mark. Predicting in-hospital mortality of icu patients: The physionetcomputing in cardiology challenge 2012. Computing in cardiology, 39:245–248, 2012.

[25] Philip B. Weerakody, Kok Wai Wong, Guanjin Wang, and Wendell Ela. A review of irregular time series data handling with gated recurrent neural networks. Neurocomputing, 441:161–178, 2021.

[26] Hedi Xia, Vai Suliafu, Hangjie Ji, Tan Nguyen, Andrea Bertozzi, Stanley Osher, and Bao Wang. Heavy ball neural ordinary differential equations. In M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan, editors, Advances in Neural Information Processing Systems, volume 34, pages 18846–18859. Curran Associates, Inc., 2021.
APPENDIX

A. Synthetic data

The protocol reported by Rubanova et al. is followed [22]. We generate 10,000 one-dimensional sequences with 50 random time points rounded to the nearest of 0.1 as well as 0.001 units in the interval [0, 5]. We use the sine function with amplitude of 1 and frequency sampled uniformly from the interval of [0.5, 1], and sample the starting point from a normal distribution with a mean of 1 and standard deviation of 0.1.

B. MuJoCo physics simulation

The code published by Bilos et al. and Rubanova et al. is used directly to obtain the data for this experiment [1], [22]. They generate 10,000 sequences of the “hopper” with 100 time steps with 14 features. We use that data without any changes, and sampled 50 or 10 time steps randomly for the experiment.

C. MIMIC-IV clinical dataset

We followed [1] for processing MIMIC-IV dataset without any changes. They also follow and modify the procedure from [6] which pre-processes MIMIC-III to pre-process MIMIC-IV. After running the code, the data contains time series of 17,874 patients with 102 features. We zero-padded the time series sequences to the longest sequence of 919 time steps at the beginning of the sequences. Also, the data was used directly without any additional normalization. Because a lot of the features are not observed in each time step, the data contains a mask dataset of 102 features consisting of 0s and 1s indicating whether that corresponding features are observed or not.

D. General procedure

All the data used in the experiments below are split into train, validation, and test sets with an 80:10:10 ratio. The models are trained with early stopping using the validation set, and the results are reported for the test set. We used a machine with 187 GB of RAM, as well as one or two NVIDIA Quadro RTX 8000 48GB GPUs to train our models for single and multi GPU modes respectively. We find the optimal hyperparameters for each model individually as time and available computation resources allow. We use a fixed random seed for all runs.

E. Packages

Please refer to requirements.txt in our code for the complete list of packages.

- pytorch-lightning: 1.5.10
- pytorch: 1.10.2
- CUDA: 11.3
- torchdiffeq: 0.2.2 [5]
- All experiments: Adam optimizer
- Multi-GPU: False
- Early stopping patience epochs: 10
- Min/max epochs: 50/1000
- Learning Rate: 0.01 (Combined time) 0.001 (All other models)
- Default Mode: Fixed dt
- Number of adaptive steps for adaptive fixed mode: 5
- Dynamic step growth factor for adaptive geometric mode: 1.5
- Combined time method dopri5 tolerances: $10^{-3}$ (rtol) $10^{-4}$ (atol)

F. Synthetic data

- Batch size: 50 (single-GPU), 25 (multi-GPU)
- Hidden size: 10
- Evolver dropout: False
- Step size: 0.1 (ODE-RNN fixed dt and Combined time with euler)

G. MuJoCo physics simulation

- Batch size: 50 (single-GPU), 25 (multi-GPU)
- Hidden size: 20
- Evolver dropout: False
- Step size: 0.01 (ODE-RNN fixed dt and Combined time with euler)

H. MIMIC-IV clinical dataset

- Batch size: 100
- Hidden size: 64
- Evolver dropout: 0.4
- Step size: 0.05 (ODE-RNN only)

We also tested using multiple GPUs in parallel, using synthetic and MuJoCo datasets with standard deviation. Due to the large size of the MIMIC dataset, our instrument does not have enough memory to run the multi-GPU modes. We find that the models do not parallelize well, and multi-GPU offers no real speedup when compared to a single GPU.

| Wall time per epoch (s) | Synthetic (Rounding) | MuJoCo (% points selected) |
|-------------------------|----------------------|-----------------------------|
|                         | 0.001                | 0.1                         |
| Fixed dt                | 26.13 ± 1.19         | 16.55 ± 0.40                |
| Adaptive fixed          | 31.05 ± 1.58         | 32.47 ± 1.25                |
| Adaptive geometric      | 23.75 ± 0.61         | 18.09 ± 0.41                |

TRAINING RUNTIME PER EPOCH AFTER THE FIRST EPOCH FOR MULTI-GPU

111