18.337 PROJECT REPORT: AUGMENTED NEURAL ODES

KERRI LU

Abstract. Neural Ordinary Differential Equations are a class of deep neural network models that learn supervised mappings from inputs to outputs by solving an ODE initial value problem. However, NODEs have limited expressivity, as they preserve topology of the input space. This motivates the introduction of Augmented Neural ODEs (ANODEs), which augment the space on which the ODE is solved to a higher dimension, increasing model expressivity and allowing for simpler ODE flow trajectories. In this paper, we review and implement ANODE algorithms for regression and classification, experimenting with several variants including zero-padding augmentation, input-layer augmentation, temporally regularized augmentation, and second-order neural ODEs. Running the algorithms on toy time series and classification datasets as well as image classification tasks, we show that augmentation generally improves model performance and enables faster convergence compared to regular NODEs. All code is available at https://github.com/kerrilu/18337-anode-project.

1. Introduction and Background.

Neural ODEs (NODEs), introduced in [1], are a class of deep neural network models that learn a mapping from input $x \in \mathbb{R}^d$ to an output hidden state $\phi(x) = h(T) \in \mathbb{R}^d$ at end time T by solving the initial value problem

$$\frac{dh(t)}{dt} = f(h(t), t) \text{ with initial condition } h(0) = x$$

where $f : \mathbb{R}^d \to \mathbb{R}^d$ is a differentiable, dimension-preserving neural net function. These models can be viewed as a continuous version of residual networks, with parameters encoded in f. During training with backpropagation, the weights of f are adjusted to fit ground-truth output labels, by optimizing some loss function between the learned features $\phi(x)$ and the true labels. While $\phi(x)$ is dimension preserving, NODEs can be modified to perform supervised regression or classification by adding a linear neural network layer $l : \mathbb{R}^d \to \mathbb{R}$ after the ODE solver output, as shown in Figure 1.

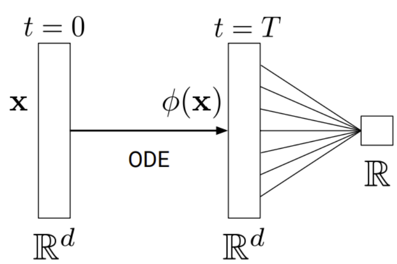

FIG. 1. *Neural ODE architecture with final linear layer mapping to scalar output, from [2].*

1.1. Limitations of NODEs.

NODEs have several practical advantages, such as their use in irregularly sampled time series prediction and density estimation [3], as well as their constant memory cost. However, NODEs also have limited expressivity, as they preserve the topology of the input space. This limitation often makes learning NODE approximations for functions computationally costly [2]. In particular, we define the *flow* of a NODE as the trajectory of the hidden state $h(t)$ over time from $x = h(0)$ to $\phi(x) = h(T)$. Intuitively, since the trajectories corresponding to different initial condition inputs x cannot intersect, there are many classes of functions which cannot be represented by ODE flows.

An illustrative example is the class of one-dimensional functions g satisfying $g(1) = -1$ and $g(-1) = 1$. As shown in Figure 2, any pair of continuous trajectories corresponding to these mappings must intersect each other, and thus cannot be learned by a NODE.

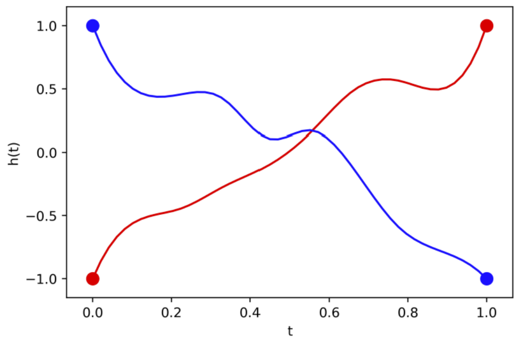

FIG. 2. *NODEs cannot represent trajectories for both $g(1) = -1$ and $g(-1) = 1$, from [2].*

Another canonical example is that of nested spheres. It is shown in [2] that NODEs cannot represent functions $g : \mathbb{R}^d \to \mathbb{R}$ where $g(x) = 1$ for $||x|| \leq r_1$ and $g(x) = -1$ for $r_2 \leq ||x|| \leq r_3$ (for any $0 < r_1 < r_2 < r_3$). As shown in Figure 3, in order for the NODE to linearly separate the two classes of points, the learned flows from the inner and outer spheres would intersect each other.

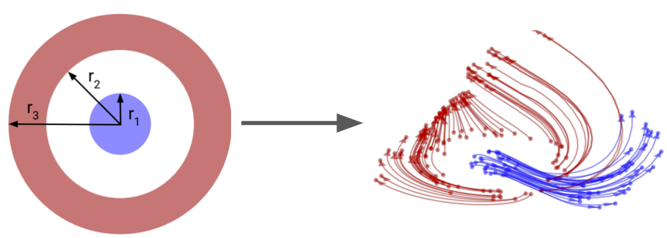

FIG. 3. *NODEs cannot easily represent classification of nested spheres (left) due to intersecting flows (right), adapted from [2].*

1.2. Augmented Neural ODEs. The limitations of NODEs motivate the introduction of Augmented Neural ODEs (ANODEs), which augment the input space from $\mathbb{R}^d$ to $\mathbb{R}^{d+p}$ so that ODE flows are "lifted" to higher dimensions so that they don't intersect. Formally, ANODEs pad input data points x with p zeros, introduce the augmented points $a(t) \in \mathbb{R}^p$, and solve the following modified initial value problem for $h(T)$.

$$\frac{d}{dt}\begin{bmatrix} h(t) \\ a(t) \end{bmatrix} = f\left(\begin{bmatrix} h(t) \\ a(t) \end{bmatrix}, t\right) \text{ with initial condition } h(0) = \begin{bmatrix} x \\ \mathbf{0} \end{bmatrix}.$$

ANODEs can thus learn a broader class of functions and tend to result in smoother and simpler flows. They are "empirically more stable, generalize better and have a lower computational cost" than NODEs, and fewer function evaluations are required to solve the augmented ODE [2].

1.3. Structure of Paper. In this paper, we review and implement variants of ANODE algorithms, and experimentally evaluate their performance. The rest of the paper is structured as follows. In Section 2, we give a brief overview of related work on NODEs and ANODEs. In Section 3, we discuss algorithmic methods and potential ANODE performance improvements such as input-layer augmentation, temporally regularized augmentation, and second-order neural ODEs. In Section 4, we discuss our experiments and evaluate results of using NODEs and ANODEs to learn toy time series and classification datasets, as well as image classification on MNIST. In Section 5, we summarize the conclusions of our work, and in Section 6, we discuss potential future research directions. All code is available at https://github.com/kerrilu/18337-anode-project.

2. Related Work. In addition to the standard *zero-padding augmentation* proposed by [2], there are several potential performance optimizations for ANODEs in the literature. One alternative is *input-layer augmentation*, where the augmented initial condition is $h(0) = g(x)$ for some input network $g : \mathbb{R}^d \to \mathbb{R}^{d+p}$, which allows the model more freedom (improving model capacity) and empirically decreases the number of function evaluations [5]. To further accelerate training time, [3] demonstrates regularization by randomly sampling the end time T of the ODE. Additionally, using second-order (or higher) ODEs can improve parameter efficiency; [6] considers second-order NODEs as a special case of ANODEs with constraints on the neural network structure, and shows that even first-order ANODEs can use augmented dimensions to learn higher-order dynamics.

The Julia SciML ecosystem includes several packages with high-performance implementations of differential equation solvers, such as DifferentialEquations.jl [8]. In particular, the DiffEqFlux.jl library [7] combines neural networks and differential equations, providing a framework for training NODEs as well as related models such as Neural Stochastic Differential Equations. Though less directly relevant, the NeuralPDE.jl library [9] implements physics-informed neural networks to efficiently approximate high-dimensional PDE solutions.

3. Methods. For our implementation of ANODEs, we utilize the DiffEqFlux.jl library to efficiently construct neural networks (using Flux) and train neural ODEs. The implementation of zero-padding augmentation as described in Section 1.2 is straightforward. Using the notation from that section, we simply pad inputs with p zeros, solve the resulting neural ODE by optimizing a loss function of our choice, and remove the augmented points $a(t) \in \mathbb{R}^p$ from the output. We can visualize the resulting ODE flow trajectories, as well as test accuracies and losses over iterations, to evaluate model performance. We experiment with several values for the augmentation dimension p to improve performance.

We also implement a few extensions and performance improvements for ANODEs, described in more detail below.

3.1. Input-Layer Augmentation. In Input-Layer Augmented NODEs (IL-ANODEs) [5], the initial condition $h(0)$ is computed as the output of an input neural network $g(x) : \mathbb{R}^d \to \mathbb{R}^{d+p}$ mapping inputs to a higher dimensional space. This can be seen as a generalization of ANODEs (which correspond to the case $g(x) = (x, \mathbf{0})$) that allows for richer representations and greater expressivity compared to the simple zero-padding operation. Empirically, this has been shown to allow for faster convergence and fewer function evaluations. In our implementation, we add a linear layer before the Neural ODE layer (using Flux.Chain) and train the entire network together. Note that the added input layer slightly increases the total parameter complexity of the system.

3.2. Temporal Regularization. Another potential performance improvement is the temporal regularization method proposed by [3], in which the end time parameter T of the NODE or ANODE integration limits is randomly perturbed. This added stochasticity during training is empirically shown to simplify model dynamics, reduce computational cost, and lead to faster convergence during training. Formally, in our implementation, the hidden state at time t becomes

$$h(t) = h(0) + \int_0^T f(h(t), t)dt = \text{ODESolve}(h(0), f, 0, T)$$

where the end time T is sampled uniformly at random from the interval $(t-b, t+b)$ for some scalar parameter b. (Without regularization, we would simply have $T = t$.) This method effectively enforces convergence at time $t - b$ rather than time t, as illustrated in Figure 4.

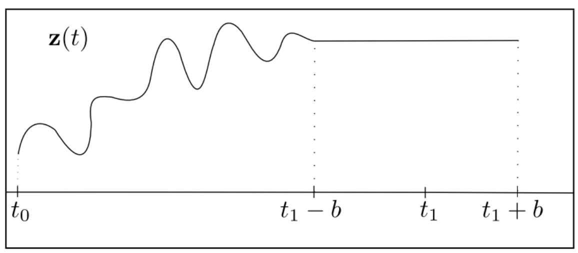

FIG. 4. *Behavior of NODEs after temporal regularization with parameter b, from [3].*

3.3. Higher-Order NODEs and ANODEs. NODEs and ANODEs can be generalized to higher dimensions, as discussed in [5]. An nth order NODE can be described by concatenating vectors $h_i(t) \in \mathbb{R}^{d/n}$ so that the hidden state is $h(t) = [h_1(t), h_2(t), \ldots, h_n(t)]$. The system of coupled first-order equations is

$$\frac{d}{dt}h_i(t) = h_{i+1}(t) \text{ for } i < n,$$
$$\text{and } \frac{d}{dt}h_n(t) = f(h(t), t)$$

where $f : \mathbb{R}^d \to \mathbb{R}^{d/n}$ is a neural network whose output dimension is n times smaller than its input dimension. This results in increased parameter efficiency compared to first-order NODEs and ANODEs where f is dimension-preserving.

Note that equivalently, we can write and solve the nth order NODE equation as

$$\frac{d^n}{dt^n}h_1(t) = f\left(h_1(t), \frac{d}{dt}h_1(t), \ldots, \frac{d^{n-1}}{dt^{n-1}}h_1(t), t\right).$$

For image classification, we implemented a second-order Neural ODE, using DiffEqFlux's built-in function for solving second-order ODE problems. As observed in [6], second-order NODEs can be viewed as a constrained type of ANODE where the augmenting vector $a(t)$ is the derivative of $h(t)$, which also constrains the structure of f. Unlike first-order NODEs, second-order and higher NODEs (even without augmentation) do not necessarily preserve topology of the input space and can thus represent a wider range of functions.

4. Experiments and Results. In this section, we discuss experiments and results from applying ANODEs to a few different settings: toy time-series prediction with sinusoidal functions, toy classification of nested spheres, and MNIST image classification. For the first two toy examples, we use simple zero-padding ANODEs and experiment with augmentation dimensions. For the more complex image classification task, we also experiment with the ANODE variants and performance improvements described in the previous section.

4.1. Time-Series Prediction with Sinusoidal Functions. For this toy example, we generate 100 datapoints from each of the functions $y(x) = \sin(x)$ and $y(x) = \sin(x) + x$ and inject a small amount of Gaussian noise. Then, our goal is to train ANODEs to learn the functions. Note that these two functions cannot be represented by regular NODEs due to intersecting flow trajectories (e.g. any continuous flows for $\sin(5\pi/6) = 0.5$ and $\sin(\pi/2) = 1$ would have to intersect).

We compare zero-padding ANODE performance for different numbers of augmentation dimensions p, and also compare against the baseline performance of the original NODE algorithm. We use a simple neural net with input and output size $2 + p$ (since each datapoint $(x, y(x))$ is two-dimensional), and a single hidden layer of size 20. We use $(0, 0)$ as the initial condition. The loss function is the sum of squared differences between predicted coordinates and ground-truth data. The ANODE is trained for 1000 iterations using Adam optimizer with learning rate 0.05, then the BFGS optimization algorithm is used for fine-tuning.

The final functions learned by the NODE and ANODEs are shown in Figure 5. In both cases, the NODE prediction (in red) differs significantly from the ground-truth data (in blue). Increasing the number of augmentation dimensions significantly improves the predictions. The ANODE padded with 5 zeros predicts $y(x) = \sin(x)$ fairly well. More augmentation is needed for the more complex $y(x) = \sin(x) + x$ function: an ANODE padded with 10 zeros performs well.

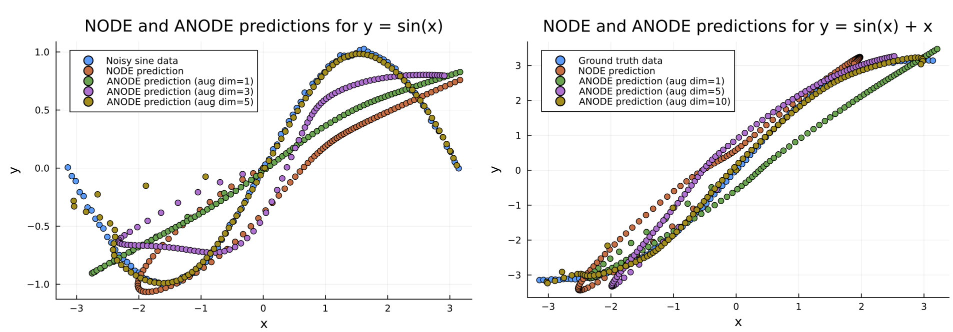

FIG. 5. *NODE and ANODE predicted coordinates for our time-series functions.*

We can also quantify convergence by plotting loss values over training iterations. For example, Figure 6 shows the models' loss for predicting $y(x) = \sin(x) + x$. In general, we see that increasing the number of augmentation dimensions results in lower loss and better convergence in fewer iterations (whereas the regular NODE loss and ANODE loss with augmentation dimension $p = 1$ do not converge at all). This is expected, as lifting the ODE to a higher dimensional space improves the model's expressivity and allows it to learn simpler flows more quickly.

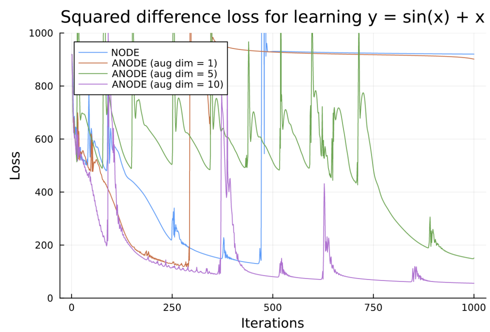

FIG. 6. *NODE and ANODE training loss for predicting $y(x) = \sin(x) + x$. Note that the loss converges in fewest iterations when we use the highest number of augmentation dimensions ($p = 10$).*

4.2. Classification of Nested Spheres. In this section, we train ANODEs to do classification on the canonical example where regular NODEs fail: nested concentric circles. We generate a dataset of 4000 datapoints, in which points in the inner circle (of radius 1) are labeled "1" and points in the outer annulus (between the circles of radius 2 and radius 3) are labeled "-1." The dataset is visualized in Figure 7.

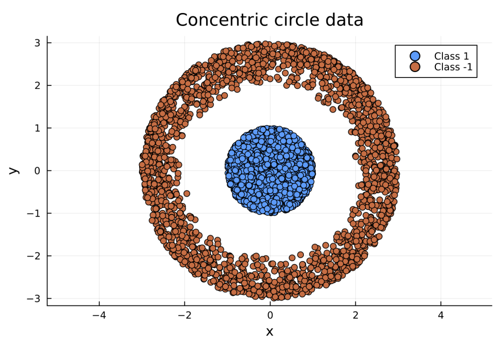

FIG. 7. *Concentric circle data with class labels.*

We use a neural network with two hidden layers of dimension 64, and Adam optimizer with learning rate 0.05. A final linear layer of the network outputs a real scalar value (so technically it computes a regression). We use mean square error loss between the ground-truth and predicted labels. We experiment with 2, 5, and 10 zero-padding augmentation dimensions and train all models for 20 epochs.

The resulting contour maps for predicted regression values at each coordinate are shown in Figure 8. The NODE model does not converge, while the performance of the ANODE models improves as the number of augmentation dimensions increases (for similar reasons as explained in the previous subsection). With 10 augmentation dimensions, the contour map is close to being a series of concentric circles of decreasing label value as radius increases, which is the desired result.

We plot the mean square error loss for each of the models over the training iterations in Figure 9. The loss does not converge for the NODE. For the ANODEs, increasing augmentation dimension leads to faster convergence and lower loss, as before.

Finally, to better understand ANODE behavior, we plot the learned features (the

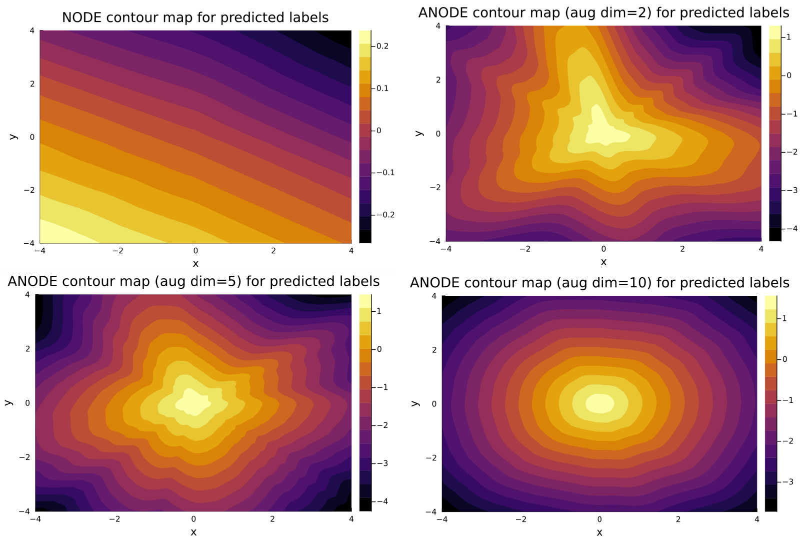

FIG. 8. *Contour maps for predicted labels at each coordinate, for trained NODE and ANODE models.*

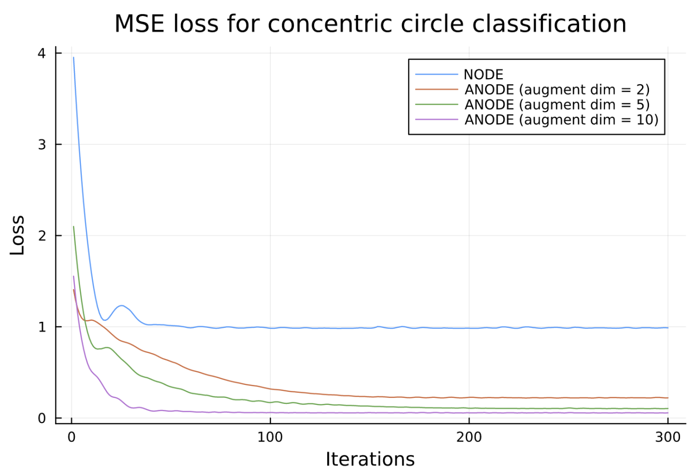

FIG. 9. *Model loss during training for learning concentric circles.*

final flow trajectory values $\phi(x) = h(T)$) for augmentation dimension 10. In Figure 10, we choose a few of the augmented dimensions and plot the final flow trajectory locations on the z-axis in 3D. We also plot the learned features for the regular NODE in 2D for comparison. In the ANODE model, we see that the higher dimensionality allows the inner circle class to “lift out” and become linearly separable from the outer annulus, without having intersecting flows. By contrast, the NODE learned features are only a slightly distorted version of the original data, and the red and blue points do not become linearly separable.

As expected, the augmentation allows the model to learn a class of regression functions that would not be learnable by the regular NODE.

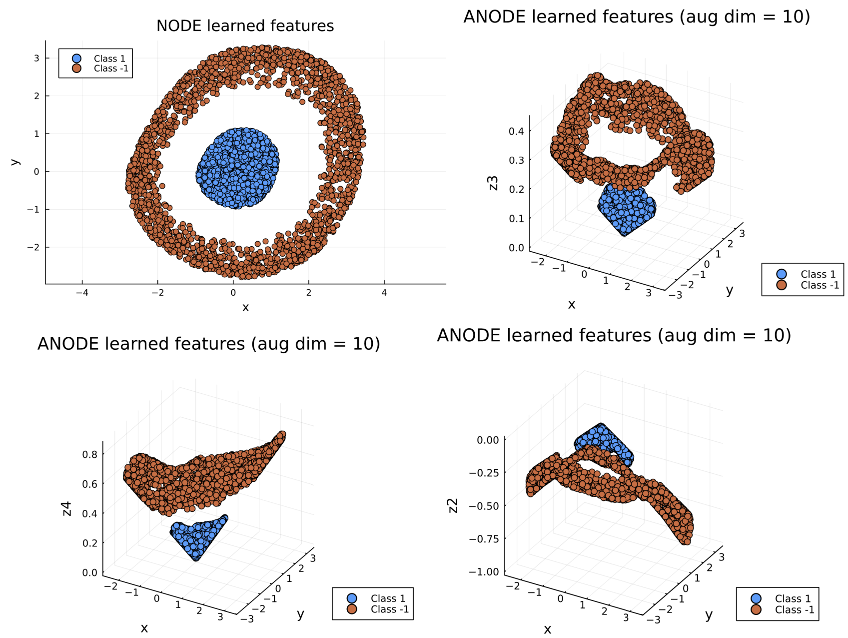

FIG. 10. *Flow trajectory features at final time T for NODE and three of the augmented ANODE dimensions.*

4.3. MNIST Image Classification. Finally, we train NODE and ANODE models to perform classification on the MNIST image dataset, which consists of 28 by 28 pixel images of handwritten digits paired with one-hot ground-truth labels from 0 to 9.

The first layer of the neural network reshapes the images to vectors of length 784. The next layer is a dense downsampling layer with 20 output nodes which serve as the input to the Neural ODE. The NODE has two hidden layers of 10 nodes each, and 20 output nodes. Finally, there is a dense layer that maps the NODE output to a vector of length 10, corresponding to the one-hot encoding of the ten digits. We use logit cross entropy loss, Adam optimizer with learning rate 0.05, and batch size 100. We train for 50 iterations.

For zero-padding augmentation, the only modification to the network architecture is that the input to the NODE is padded with p zeros, for a total length of $20+p$ (and then the NODE output must also have length $20+p$, but it is truncated to the first 20 nodes before being fed into the output layer). The overall network architecture is shown in Figure 11.

We experiment with values $p = 10, 25, 40$. In Figure 12, we plot the NODE and ANODE model classification accuracy and loss on the test set over the training iterations. The ANODE models converge to somewhat higher accuracy and lower loss in fewer iterations than the NODE. The final NODE accuracy of 73% is a few percentage points lower than the final ANODE accuracy of 77% (with augmentation dimension $p = 10$). Further increasing the augmentation dimension also slightly improves final accuracy, as seen in the graph.

We also experiment with the ANODE variants described in Section 3.

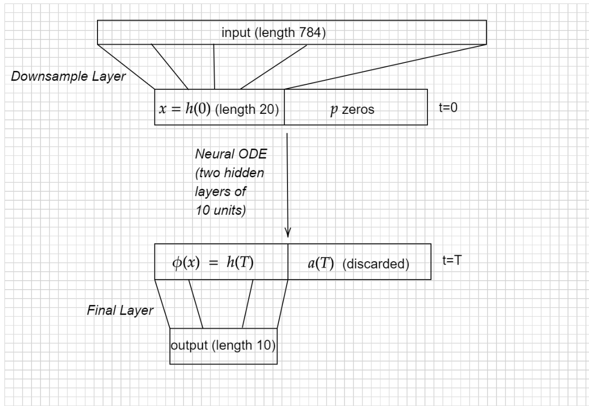

FIG. 11. *Neural network and ANODE architecture for MNIST classification.*

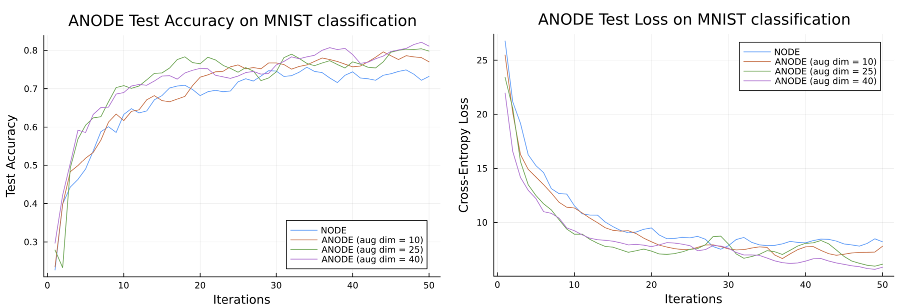

FIG. 12. *Zero-padding ANODE test accuracy and loss over training iterations.*

4.3.1. Input-Layer Augmentation. To implement input-layer augmentation (IL-ANODEs), we change the dense downsampling layer to have $20+p$ output nodes. That is, the network directly learns a mapping from the input vector of length 784 to a vector of length $20+p$ which will serve as the input to the NODE (rather than mapping to a vector of length 20 and then padding with zeros).

As before, we experiment with different augmentation dimensions $p = 10, 25, 40$ and plot the resulting test accuracies and losses in Figure 13. Even with the same number of augmentation dimensions $p = 10$, the IL-ANODE has somewhat faster convergence and has a slightly higher final accuracy of 81% compared to the zero-padding ANODE. This is consistent with the hypothesis that learning an input network increases the freedom and capacity of the model to represent more complex mappings. However, further increasing the IL-ANODE augmentation dimension above $p = 10$ does not seem to result in significantly improved performance.

4.3.2. Temporal Regularization. The original NODE is solved from time span $t = 0$ to $t = 1$. We implement temporal regularization by randomly sampling the end time from the interval $(1-b, 1+b)$ where b is the regularization parameter defined in Section 3.2. We keep the zero-padding augmentation dimension constant at $p = 10$ and experiment with several values of $b = 0.2, 0.3, 0.6$. Plotting the training accuracies and losses in Figure 14, we see that the temporally regularized ANODEs are slower to converge than the regular ANODE. This is expected, as regularization works to prevent overfitting during training (reducing the variance and generally increasing the

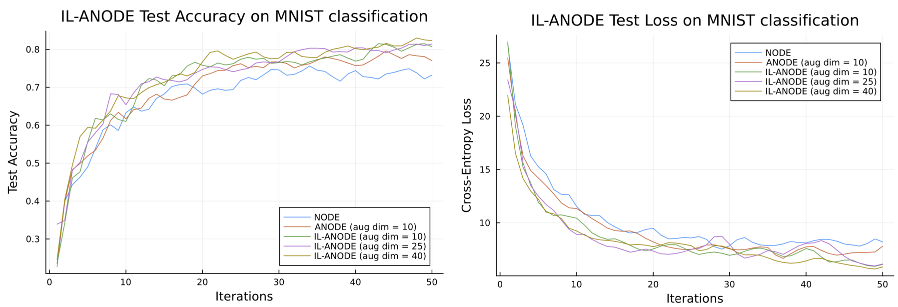

FIG. 13. *Input-layer augmentation ANODE test accuracy and loss over training iterations.*

bias), which results in lower training accuracy.

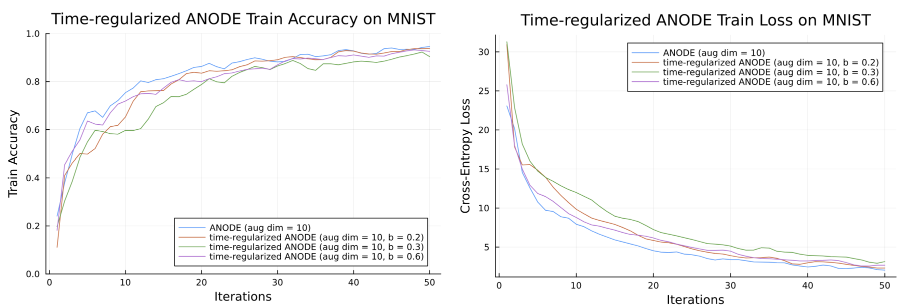

FIG. 14. *Time-regularized augmentation ANODE train accuracy and loss over training iterations.*

In Figure 15, the test accuracies and losses suggest that the temporal regularization slightly improves model performance in general (although there doesn't seem to be a clear relationship between the value of b and speed of convergence). The final test accuracies are around 81% which is slightly higher than the regular ANODE.

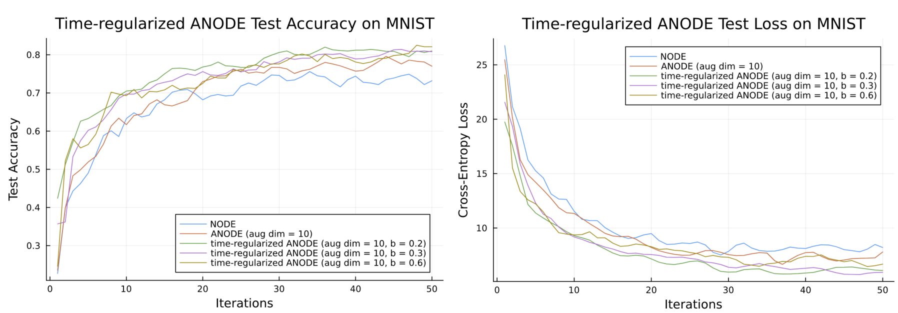

FIG. 15. *Time-regularized augmentation ANODE test accuracy and loss over training iterations.*

4.3.3. Second-Order NODEs. We implement a second-order NODE for MNIST classification. We modify the network architecture for better performance as follows. We change the output of the downsampling layer to have length 80, and the NODE part of the network to have input and output size 40 with a single hidden layer of 20 nodes. (The downsampling layer output can be viewed as the concatenation of the initial condition $h(0)$ and the initial condition for its derivative $h'(0)$, each of which

Model	NODE	ANODE	IL-ANODE	Time-regularized ANODE (b=0.2)	Second-order NODE
Final accuracy	73.2%	77.0%	80.7%	81.0%	55.4%

TABLE 1
Final test accuracy after 50 iterations for NODEs and ANODEs with $p = 10$.

have length 40 in this case.) Empirically we find that using the second order equation $\frac{d^2}{dt^2}h(t) = f(h(t), t)$ seems to produce somewhat better results than feeding in both $h(t)$ and its derivative (i.e. $\frac{d^2}{dt^2}h(t) = f(h(t), h'(t), t)$). We use the second order ODE solver in DiffEqFlux to directly compute the second-order NODE output.

Unfortunately, as seen in Figure 16, the second-order NODE performance is much worse than the first-order NODE and ANODE models and does not converge. The final test accuracy is around 55%, and even training for more epochs seems to result in convergence at only around 60%. We believe our naive approach of directly using the second order ODE solver on the downsampled inputs is likely flawed; optimization of second order NODEs seems to require a different framework and a modified adjoint sensitivity method. More discussion on potential improvements can be found in Section 6 (Future Work).

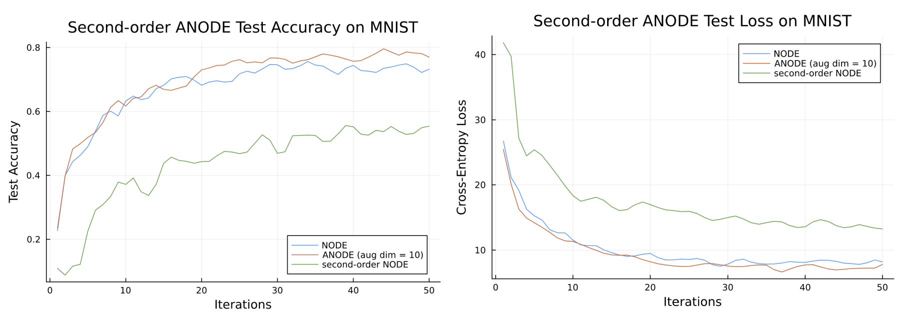

FIG. 16. *Second-order NODE test accuracy and loss over training iterations.*

The final test accuracies for NODEs and ANODEs with augmentation dimension $p = 10$ are summarized in Table 1. Overall, our results suggest that augmentation enables the NODE model to achieve somewhat higher accuracy, and our performance improvements seem effective (with the exception of the second-order system).

5. Conclusion. In this paper, we have reviewed Augmented Neural ODEs and implemented several variants with applications to toy examples and image classification. Our empirical results support the notion that augmenting the space on which we solve a neural ODE increases expressivity of the model and allow it to learn a broader class of functions, leading to faster convergence and smoother ODE flows. Furthermore, we showed that optimizations such as input-layer augmentation and time regularization can further improve model performance.

6. Future Work. For future work, we would like to improve on our experiments with second (or higher) order NODEs, as it would be interesting to better understand why our naive approach failed. One alternative approach would be to recursively use the first-order adjoint method, as higher order NODEs can be decomposed into a series of coupled first-order ODEs as shown in Section 3.3. However, there are also more sophisticated approaches for higher-order adjoint sensitivity methods. In particular,

[4] proposes an optimal control programming method for second-order optimization of Neural ODEs, and shows that it empirically improves convergence time because only a single backward pass is needed to find all derivatives.

It would also be of interest to try to apply Augmented Neural ODEs to a real-world setting. Neural ODEs have shown to be especially useful for modeling irregularly sampled time series [3], and it would be nice to explore potential applications to areas such as climate and weather forecasting where data may be temporally sparse and dynamical systems modeling is commonly used.

REFERENCES

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

[2] E. Dupont, A. Doucet, and Y. W. Teh, *Augmented neural ODEs*, Advances in Neural Information Processing Systems, 32 (2019).

[3] A. Ghosh, H. Behl, E. Dupont, P. Torr, and V. Namboodiri, *Steer: Simple temporal regularization for neural ODE*, Advances in Neural Information Processing Systems, 33 (2020), pp. 14831–14843.

[4] G.-H. Liu, T. Chen, and E. Theodorou, *Second-order neural ODE optimizer*, Advances in Neural Information Processing Systems, 34 (2021), pp. 25267–25279.

[5] S. Massaroli, M. Poli, J. Park, A. Yamashita, and H. Asama, *Dissecting neural ODEs*, Advances in Neural Information Processing Systems, 33 (2020), pp. 3952–3963.

[6] A. Norcliffe, C. Bodnar, B. Day, N. Simidjievski, and P. Liò, *On second order behaviour in augmented neural ODEs*, Advances in Neural Information Processing Systems, 33 (2020), pp. 5911–5921.

[7] C. Rackauckas, M. Innes, Y. Ma, J. Bettencourt, L. White, and V. Dixit, *DiffEqFlux.jl-a Julia library for neural differential equations*, arXiv preprint arXiv:1902.02376, (2019).

[8] C. Rackauckas and Q. Nie, *DifferentialEquations.jl–a performant and feature-rich ecosystem for solving differential equations in Julia*, Journal of open research software, 5 (2017).

[9] K. Zubov, Z. McCarthy, Y. Ma, F. Calisto, V. Pagliarino, S. Azeglio, L. Bottero, E. Luján, V. Sulzer, A. Bharambe, et al., *NeuralPDE: Automating physics-informed neural networks (PINNs) with error approximations*, arXiv preprint arXiv:2107.09443, (2021).