Neural ODEs on the MIT-BIH Electrocardiogram Dataset

Jimmy Chen and Laurie Luo
Math 179, Harvey Mudd College
jimchen@hmc.edu, hluo@hmc.edu

Abstract

In this paper, we discuss and motivate the concepts and implementation details of a newer class of machine learning algorithms, Neural Ordinary Differential Equation (NODE), as well two extensions – augmented NODE and adaptive depth NODE. We apply these methods to the electrocardiogram (ECG) dataset from the MIT Beth Israel Hospital (BIH). We first successfully replicate the result of a particular NODE implementation on the ECG dataset by the GitHub user abaietto, who compares a ResNet and a NODE model with similar architectures. Next, drawing inspiration from abaietto's model, we implement the two extended methods, and provide further comparison analysis on their respective performance.

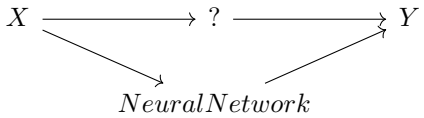
1 Neural Network and ResNet

Neural Networks (NN) are a family of algorithms that models over datasets and provides predictions. They play a central role in modern machine learning process, particularly deep learning.

For example, in a classical data modelling problem, we have N pairs of data point $X = (\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), ..., (\mathbf{x}_N, \mathbf{x}_N))$, where $\mathbf{x}_i \in X$ is the input domain and $\mathbf{y}_i \in Y$ is the output domain, and some sort of relationship exists between the two variables:

$$X \longrightarrow ? \longrightarrow Y$$

We would like to utilize the existing dataset and make prediction of the output $\mathbf{y}^*$ giving a new data point $\mathbf{x}^*$. The highly flexible NN can be iteratively trained on the dataset, ultimately describing the dataset with accuracy. Using an NN turns the classical situation into the situation below:



Internally, a generic NN consists of several layers of neurons, mimicking the way the human brain operates. In this sense, neural networks refer to systems of neurons, either organic or artificial in nature. Figure 1 is a detailed illustration of common, fully connected neural network. The input data $\mathbf{x}_i$ is first fed into the first layer, where it is transformed by the layer to some intermediate values, which are further transformed by the next layer, and so on. In this way, the neural network can be viewed as a giant composite function and, thus, the more layers there are, the deeper the network it is. For a fully connect layer, the input of each neuron is a weighted summation over the previous values with a constant term as bias, and then the result is passed through a scalar activation function, usually nonlinear.

Midterm Project for Math 179 in Harvey Mudd College.

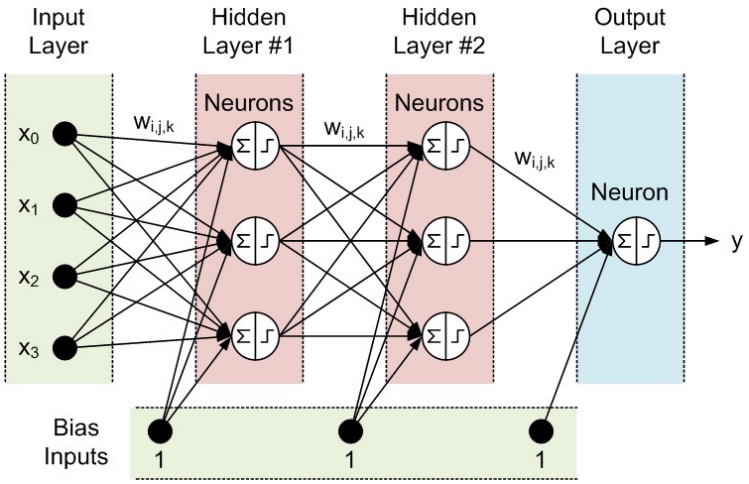


Figure 1: Illustration of a fully-connect Neural Network.

Hence, the number of layers deeply affects the performance of the neural network. Too few layer, meaning too few parameters, will cause under-fitting, wile too many layers will cause over-fitting (shown in Figure 2).

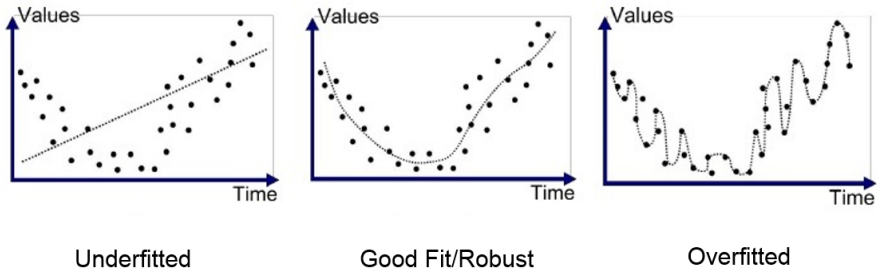


Figure 2: Underfitting and Overfitting.[1]

One class of NNs effectively allows the number of layers to be adaptively optimized for the dataset: they are the *Residual Neural Networks (ResNet)*. Whereas regular multi-layer perceptions and many other NNs chain layers together linearly through composition only, ResNets introduce shortcuts that skip over groups of layers by adding intermediate results from a few layers prior to the current output. One can consider a group of layers that has such a shortcut over it as a single residual block (or group). For the kth residual block, the output is the addition $\text{Res}_k(\mathbf{x}_k) + \mathbf{x}_k$, between the processed output of the internal layers and the original input:

$$\mathbf{x}_{k+1} = \text{ResBlock}_k(\mathbf{x}_k) = \text{Res}_k(\mathbf{x}_k) + \mathbf{x}_k \tag{1.1}$$

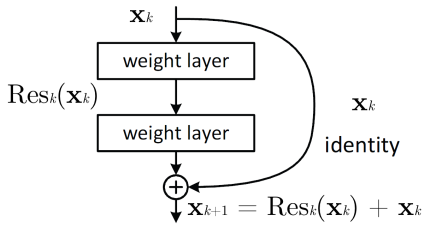


Figure 3: Illustration of a ResNet block. [2]

[1] Credit: `https://medium.com/swlh/machine-learning-how-to-prevent-overfitting-fdf759cc00a9`

The transition to Neural ODE begins by noting that equation 1.1 bears some resemblance to a forign concept. If we treat the internal process $\text{Res}_k(\mathbf{x}_k)$ as giving the derivative of a unified "variable" $\mathbf{x}$ at "time" k, then the equation effectively represents the Euler method for solving the ODE (with unit time step)

$$\left.\frac{d\mathbf{x}}{dt}\right|_{t=k} = \text{Res}(\mathbf{x}_k, k) \tag{1.2}$$

where $\text{Res}(\mathbf{x}_k, k) = \text{Res}_k(\mathbf{x}_k)$ represents Res_k at all ks.

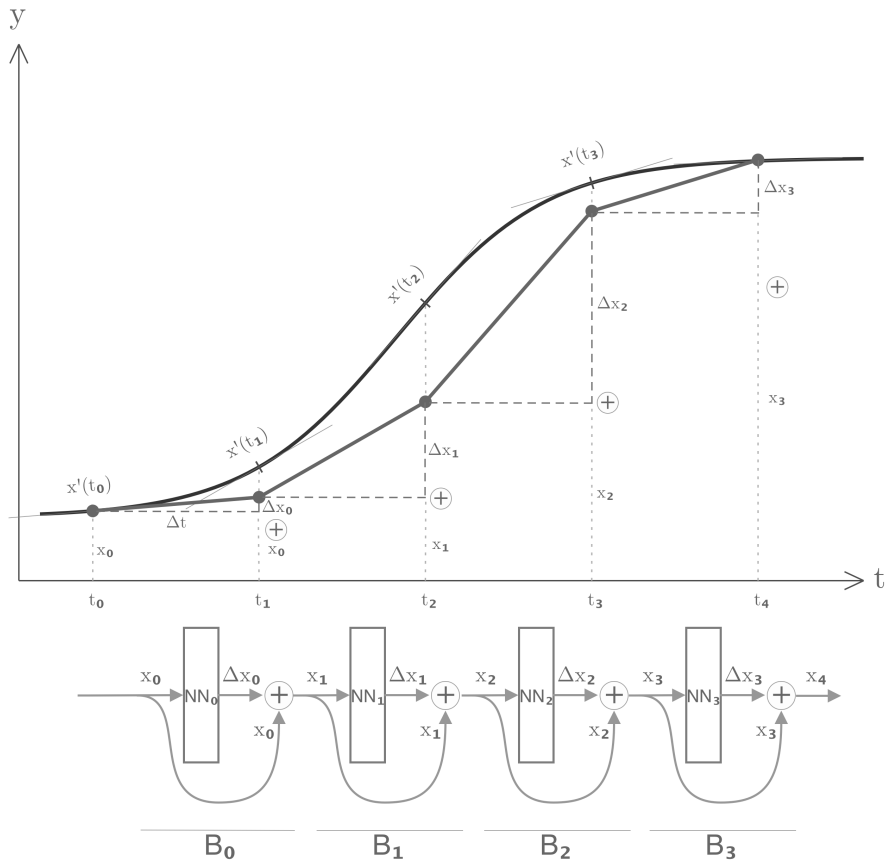


Figure 4: ResNet interpreted as the Euler method.

Figure 4 further illustrates the correlation between ResNet and the Euler method. Four ResNet blocks $\text{B}_{0\ldots3}$ are shown, with internal networks $\text{NN}_{0\ldots3}$ and shortcuts as arrow from each input to output. Now we can reinterpret each the network output as the derivative to some function of t, shown as the dark curved line. Take the first step for example. For the first block B_0, its internal network NN_0 takes in the initial point of $\mathbf{x}$, $\mathbf{x}_0$ at t_0, and produces some $\mathbf{x}'(t_0)$. To approximate the value of the function at the next time point t_1, we multiply it by the time step Δt which is assumed to be unit, so $\mathbf{x}'(t_0) = \Delta\mathbf{x}_0$, the change in value in this time step. The final estimate is the sum of the original $\mathbf{x}_0$ and the change $\Delta\mathbf{x}_0$. In the ResNet block, this is the sum of the network output and the original input through the shortcut. The step is then repeated three more times by different ResNet blocks, each corresponding to a specific time point.

[2]Credit: `https://neurohive.io/en/popular-networks/resnet/`

This does assume that the intermediate results x_k to share the same dimensions, but the assumption is less important than our purpose of illustration. The point is this: if ResNet approximates an ODE solution by the Euler method, then we can naturally extended ResNet by considering other, and better, integration methods.

2 Neural ODE

Neural ODEs (NODE) are a continuous re-imagination of ResNet. They model curves in arbitrary dimensions as solutions to first order ODEs, and are thus inherently applicable to time series. When ResNet is thought as the Euler method, each residual block corresponds to one discrete time step in the Euler method: the kth group takes the previous result $\mathbf{x}_k$ and gives the derivative of $\mathbf{x}$ at time t_k. Note that each residual group is only responsible for a single time point. Neural ODE instead uses a single NN to produce all derivatives by taking the time point as explicit input:

$$\frac{d\mathbf{x}}{dt} = \text{NN}_{\text{NODE}}(\mathbf{x}, t). \tag{2.1}$$

It follows that the solution to this ODE is the integral over time:

$$\mathbf{x}(t) = \int \text{NN}_{\text{NODE}}(\mathbf{x}, t)dt \tag{2.2}$$

From a computational standpoint, this integral can be approximated with any numerical method.

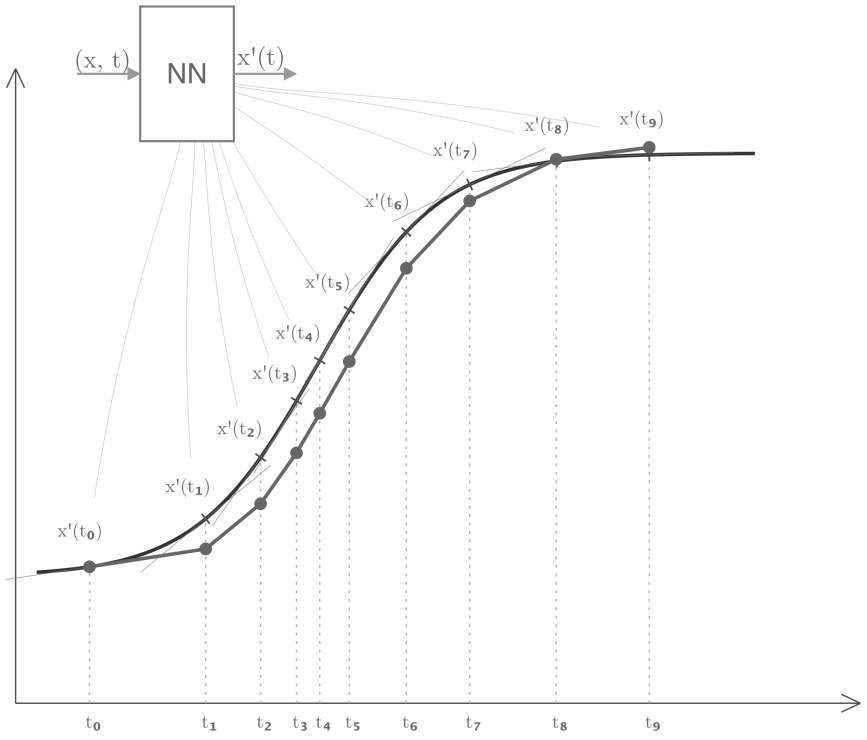


Figure 5

Figure 5 illustrates the operation of a Neural ODE. A single neural network produces all derivatives of the ODE it models: at each time point t, it takes in t and the current value $\mathbf{x}$, and the output of the NN is used as the derivative $\mathbf{x}'(t)$. The network parameters remain constant regardless of time, while the input t varies. This contrasts with ResNet as the Euler method, which uses entirely different networks for different time points.

As Neural ODEs are flexible in integration methods, integration points, the time points at which the derivative is evaluated, can be freely chosen in principle, both in amount and in value. In practice, this can reflect in adaptive strategies in choosing integration points such as reducing the step size when derivatives are large in magnitude.

Such generality has immediate advantages. Numerical methods for solving ODEs has been a rich and much-explored field, with optimized methods for accuracy, speed, and specific ODE types, most significantly superior to the Euler method. For instance, a number of fourth-order Runge–Kutta methods exist where the accumulated error scales with the fourth power of step size, whereas in the Euler's case the error is linear with step size.

Neural ODE also opens up the flexibility of choosing algorithms and of balancing between accuracy and speed. As a ResNet block corresponds to a single time step of the Euler method, the number of steps taken is tied to the ResNet's structure, namely the depth of the network. For Neural ODEs the number of steps is variable and controllable, which effectively establishes flexible depth. Larger network depth makes it more expressible and accuracy while a smaller depth cuts the time and resource cost of iterative computation. Additionally, for ResNet the memory cost of storing layers of parameters scales with depth, while Neural ODE has a constant memory profile. Finally, Neural ODES can be evaluated at any point along the solution curve, which is ideal for modeling data with irregular time point.

3 Implementation Example

In this section we replicate the result an implementation given in `https://github.com/abaietto/neural_ode_classification`. The author of this implementation (who we will refer to as abaietto, their GitHub username) compares the accuracy, time cost and memory cost of similarly structured ResNet and NODE models in the context of a supervised learning task, namely the classification of an ECG signal.

3.1 Electrocardiogram Dataset

This implementation uses MIT Beth Israel Hospital (BIH) electrocardiogram (ECG) dataset. The data is obtained from Kaggle [1], containing about 110,000 labeled data points about heartbeat classification. Each sample is annotated into the following five categories: normal (0), supraventricular premature beat (1), premature ventricular contraction (2), fusion of ventricular and normal beat (3), and finally unclassifiable beat (4). The ECGs were recorded at a frequency of 360 Hz. Thus, each sample was taken over 0.52 seconds since there are 187 measurements per sample. From the data, the proportion of each category is shown in Table 1, which shows normal heartbeats are the most common category.

Category	Proportion
0	0.83
1	0.03
2	0.07
3	0.01
4	0.07

Table 1: Proportion of each category of heartbeat

To better understand each heartbeat type, we obtained examples for each category, as shown below.

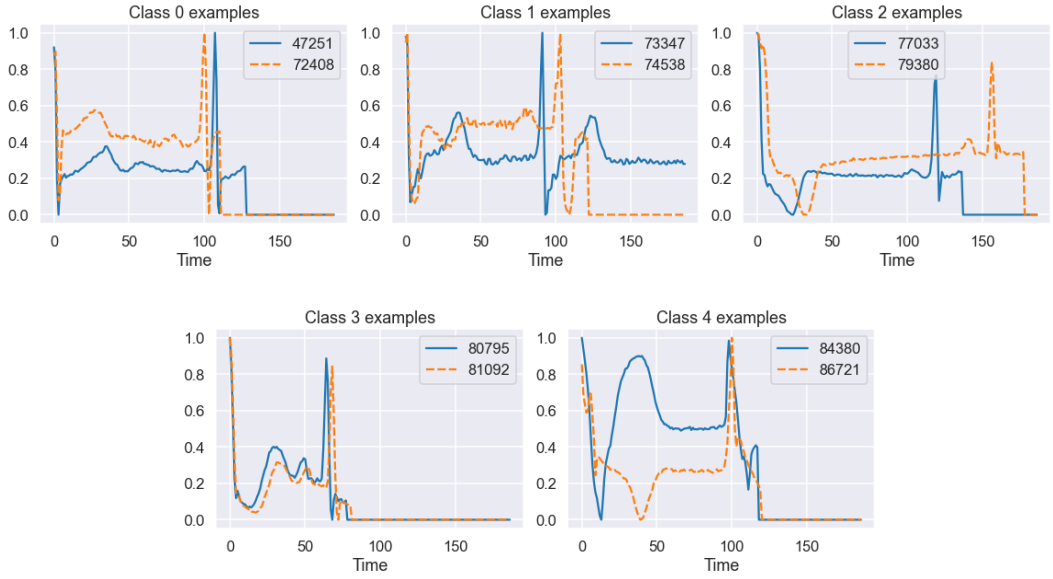


Figure 6: [3]

3.2 Network Architecture

For the purpose of comparison, the ResNet model and the NODE model here are identical in many layers. Each cardiogram sample consists of 187 values. For both models, the first three layers are one dimensional convolutions, who together results in 64 channels of time-local series, each now with 46 values. For the ResNet, this is followed by six ResNet blocks identical in structure, each with two convolutions layers plus normalization. For the NODE, the six blocks of the ResNet are replaced by one Neural ODE block, which has an internal NN consisting of two convolutions also with normalization. From this point, both models go through a fully connected layer which outputs to a size of 5 representing the categorization task.

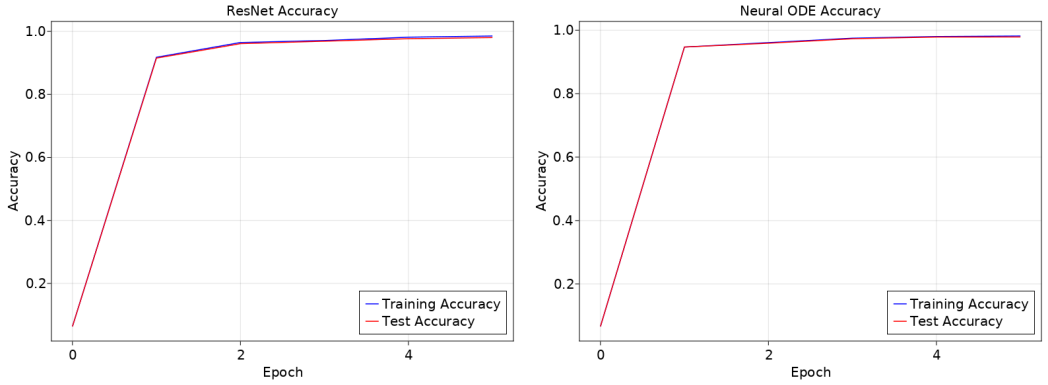


Figure 7: Comparison of accuracy between the ResNet and the Neural ODE

[3]Credit: [2].

3.3 Characteristics and Comparison Between ResNet and NODE

We are able to fully replicate the results of the the original author abaietto. Training each network for 5 epochs, the ResNet produced a final accuracy of 0.985 on the training dataset and 0.980 on the test set, while the Neural ODE produced 0.982 and 0.979 for training and test respectively. Figure 7 shows historical accuracies by epochs. We see that the final accuracies are similar between the two models.

An advantage of Neural ODEs is demonstrated when we compare the size of the models. Namely, while the ResNet contains 182853 parameters, the NODE has only 59333, thus producing the same level of accuracy as the ResNet with less than one third of the memory cost. The difference in model size is directly reflected in the network structure: the ResNet notably uses 6 residual blocks each containing 24832 parameters while the NODE uses one Neural ODE block with 25472 parameters. This verifies the aforementioned theoretical advantage of NODE networks in memory.

The implementation also reflects a potential disadvantage of Neural ODES, namely training and evaluation time. For evaluation, the ResNet takes around 9 seconds on our hardware with the test set as input while the NODE takes around 70 seconds; for training with the selected optimizer (stochastic gradient descent with momentum optimization), the ResNet takes around 1.6 to 1.8 minutes per epoch while the NODE spends around 11.7 to 16.6 minutes per. The multiplied time cost of Neural ODEs has been implicitly noted by several others, namely in [4], [5] and [6] all motivated by efficiency. An immediate area for future work are to incorporate these efficiency optimizations to this specific dataset.

4 Extensions of Neural ODE

We investigate several variants of Neural ODE to find potential improves in classification accuracy and training cost. We will refer to the basic Neural ODE model as vanilla where clarification is needed.

4.1 Augmented Neural ODEs

We begin by discussing a limitation of vanilla Neural ODEs. As shown in [7], there are certain functions that cannot be perfectly represented by Neural ODEs in an arbitrary dimension d. For example, neural ODE can never fully represent the function $g : \mathbb{R}^d \to \mathbb{R}$ defined as follows:

$$\begin{cases} g(x) = -1 & \text{if } \|x\| \leq r_1 \\ g(x) = 1 & \text{if } r_2 \leq \|x\| \leq r_3, \end{cases}$$

in which $0 < r_1 < r_2 < r_3$. Figure 8 shows $g(x)$ in two-dimension. The blue region maps to -1 and the red region points to 1.

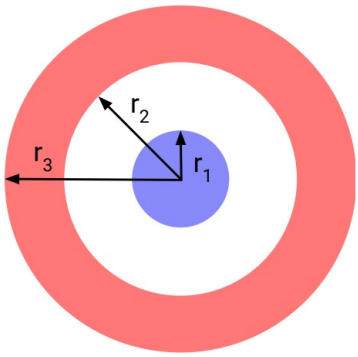


Figure 8: Diagram of $g(x)$ in two dimension ($\mathbb{R}^2$).[4]

While the proof requires the knowledge from ODE theory and topology, the reasoning is intuitive. A Neural ODE block represent a vector field, its input and output are states (or points) in the field, and the transformation by the block is a trajectory of flow of its input through the field. Crossing trajectories in a vector field can never diverge again as it would imply some points in the field to simultaneously have two distinct gradients. Consider such transformation then as a smooth topological transformation. The standard NODE classification approach, a vanilla NODE block followed by a linear layer, will not work: the NODE block make the two regions of points linearly separable for the linear layer, but function represents the blue region completely enclosed by the red hyperspherical shell; there is no way to pull the inner region outside without passing it through the output region.

The feature can be generalized to a statement that summarize what Neural ODEs can represent: they can only continuously deform the input space and cannot tear a connected region apart. For the class of functions g, this is a limitation for NODEs. However, one can circumvent such limitation by considering the problem in higher dimensions. For the concentric hypersphere problem of modeling g, only one additional dimension is needed:

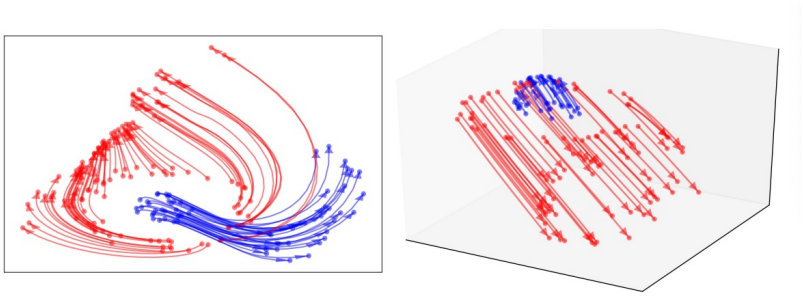


Figure 9: Illustration of the flow required for modeling two-dimensional g, in two dimensions (left) and three dimensions (right). Separating the two regions in 2D through vector field flow is impossible and can only be approximated by complex flow, while in 3D this can easily be done by moving the two regions in opposite directions along the third dimension. [5]

This is the method of *augmented Neural ODEs* (ANODEs). ANODEs introduce additional dimensions to the input space $\mathbb{R}^d$ and find a flow in $\mathbb{R}^{d+p}$, which can drastically reduce the complexity of the

[4] Credit: [7].
[5] Credit: [7].

model. Given an input $\mathbf{x}_0 \in \mathbb{R}^d$, ANODEs consider it in $\mathbb{R}^{d+p}$ as $\mathbf{x}'_0$ and evolves it in the higher dimension field accordingly. In standard practice, the coordinates of the extra dimensions are initially set to zero, that is,

$$\mathbf{x}'_0 = \begin{bmatrix} \mathbf{x}_0 \\ \mathbf{0}^p \in \mathbb{R}^p \end{bmatrix} \tag{4.1}$$

Note that the extra dimension values are only initially zero; as the initial state evolves in the higher dimension field it will traverse into the extra dimensions. If the ANODE block has initial network NN_{ANODE} then the state evolution is described as

$$\mathbf{x}'(t) = \int \text{NN}_{\text{ANODE}}(\mathbf{x}', t)dt \tag{4.2}$$

the same as vanilla NODEs.

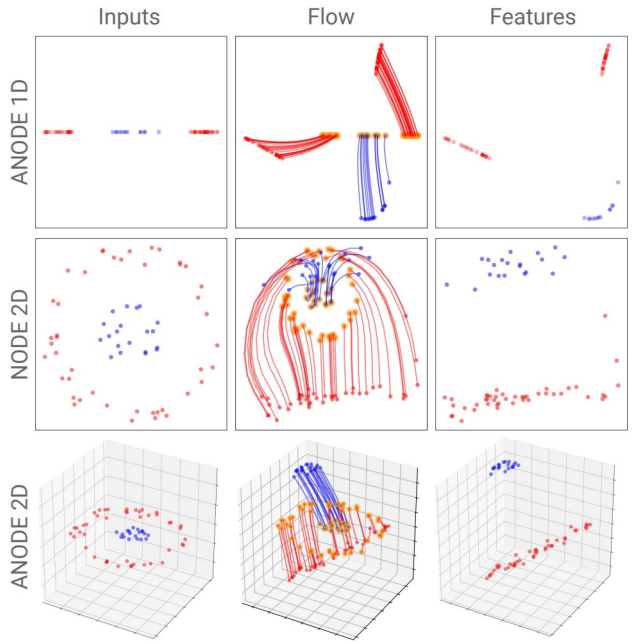


Figure 10: Flows learned by NODEs and ANODEs. [6]

A more concrete example shows in Figure 10. Figure 10 shows $g(x)$ in different dimensions. When trained on $g(x)$, augmented neural ODE is able to describe the function while NODE cannot. Augmented NODE represent the flow of function when $d = 1$ in two dimensions. As a result, it separates the points and got a simpler flow without crossings. Also, when $d = 2$, augmented neural ODE represents the flow in three dimension while the NODE was trained in two dimension. The resulting flow of NODE is complicated with a lot of crossing. However, the flow resulted from augmented Node is simple because of the introduction of the higher dimension.

4.2 Adaptive-Depth Neural ODEs

As mentioned in last section, it is impossible for a Neural ODE to solve a function by crossing trajectory. An alternative to augmentation is the method of *adaptive depth NODEs* (AD NODES).

Depth here refers to the time bounds of the ODE solution in a NODE block, i.e. the time t_0 that the input state $\mathbf{x}_0$ is interpreted to be at, and the time t_1 that the state is evolved to. The meaning of this time span (t_0, t_1) varies with context. A NODE block can directly model after a time series as an

[6] Credit: [7].

ODE, such that the input, intermediate and output states of the NODE are of the same space of the data, and that the internal network of the NODE is expected to produce the derivatives of the time series. The time variable of the time series is thus shared by the NODE, and so the integration time span is that of the data. On the other hand, a NODE block can also be used in a blackbox fashion, where the vector field and transformation it describes has little specific meaning. The time variable of such a NODE block also does not correspond to any aspect of the input. In practice, the time span is usually set to $[0, 1]$ (see Appendix for a discussion in detail).

Adaptive depth NODEs allows the depth to be dependent on each data sample by using a small NN_{depth} to decide it. The NN's input is the NODE block's input, and its output is the ending time point, so that the integration time span is $[0, \text{NN}_{\text{depth}}(\mathbf{x}_0)]$. The output of the entire AD NODE block is thus

$$\mathbf{x}_{\text{out}} = \int_0^{\text{NN}_{\text{depth}}(\mathbf{x}_0)} \text{NN}_{\text{ADNODE}}(\mathbf{x}, t)dt \tag{4.3}$$

Figure 11 shows an example of how the adaptive-depth neural ODE avoid the crossing trajectory.

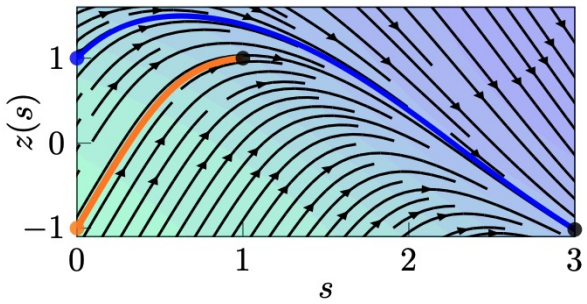


Figure 11: Illustration of an adaptive depth NODE mode, taken from the original paper [8]. The notation differs from ours, with s representing time and $z(s)$ the state at time s. Depending on the initial state, the blue and orange trajectories end at different times, 3 and 1 respectively. This is a solution to the "reflection problem" as proposed by [8]: the function $\varphi x = -x, x \in \{-1, 1\}$. With constant depth the trajectories would have to cross each other, but with adaptive depth this is easily solvable without relying on augmentation.[7]

5 Experiments on the ECG Dataset

We implemented the two extension methods of NODE, and ran all model types on large hyperparameter ranges. Here we discuss the models considered in detail and compare the performance of vanilla, augmented, and adaptive depth NODEs.

5.1 Implementation Details

As we can see in Table 1, there is significant class imbalance. One disadvantage of training a model directly on imbalanced datasets, as is with abaietto's implementation, is that accuracy can be misleading. An elementary model which always reports class 0 can achieve an accuracy of 83%; the high first epoch accuracy shown in Figure 7 also demonstrates this. Thus, we have randomly re-sampled 81920 cases from the training set and 20480 from the validation set, where each case has an equal probability chosen from each class. The number of cases where chosen to be similar to the original dataset sizes (87553 and 21891). All models referenced below were trained and tested on the re-sampled, balanced dataset, and references to accuracy scores below assume a balanced dataset.

[7] Credit: [8].

Our model architectures are partially based on abaietto's NODE network.

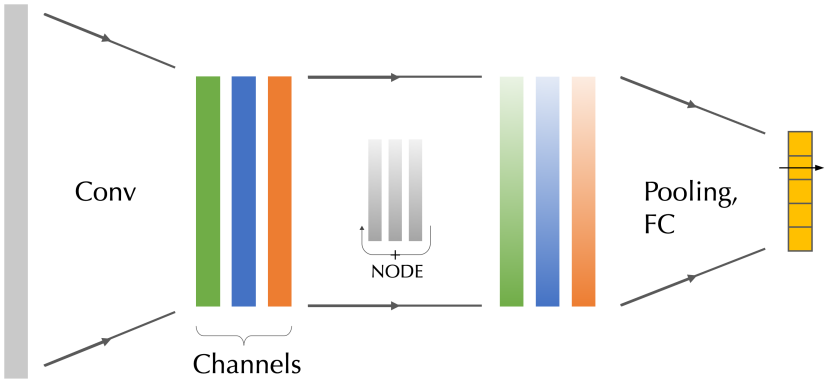


Figure 12: Illustration of the used vanilla NODE architecture. Each sample of ECG data is fed into the network as single time series (gray, left); through 1 or more layers of 1D convolution, the input is downsampled, thus shorter in length, but expanded into multiple channels (multiple color, left); the data is then treated as a single state in a vector field, whose gradients defined by an internal network, and evolves while maintaining dimension (multiple gradient colors, right); finally, a pooling and a fully connect layer outputs a score for each ECG class.

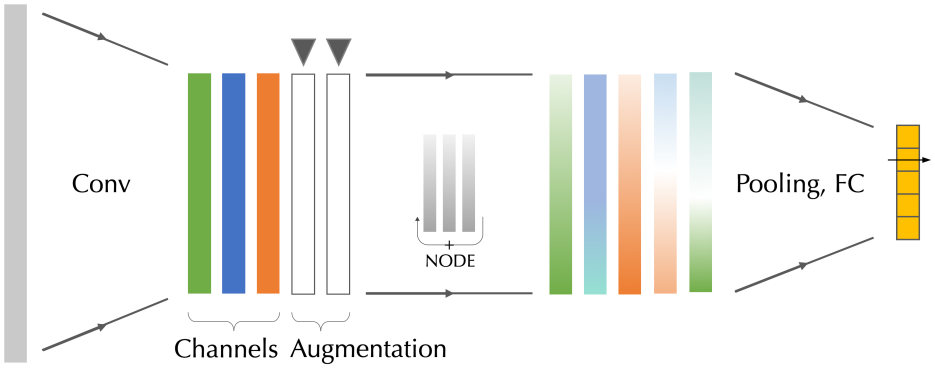


Figure 13: Illustration of the used Augmented NODE architecture. The difference from the vanilla NODE is the addition of zero-filled channels around the NODE block. The added channels are then utilized by the NODE block in the same fashion as the original channels, by the fact that they are no longer zero afterwards.

For the AD NODE architecture we follow the approach originally proposed by [8]. On top of the vanilla NODE, we use a small NN to control the integration depth of the ODE solver. The small NN is a fully connected network with one tunable-size hidden layer; its input is a single channel of the NODE block's input, as the latter full input is usually large and using it with a dense network will contribute a significant portion of parameters; finally, its output is transformed by $s \to |1 + s|$ and the final time span is $[0, |1 + s|]$.

5.2 Results and Discussion

The two figures below display the resulting performances of vanilla, augmented, and adaptive depth NODE models. Variable hyperparameters include the number of channels produced during downsampling (and separately the number of augmented channels if applicable), kernel sizes and stride lengths of convolution layers, number of layer groups for downsampling section and the

NODE block separately (either one or two groups each with one group normalization, activation and convolution in that order), size of the hidden layer of the depth-deciding NN for AD models, and learning rate[8].

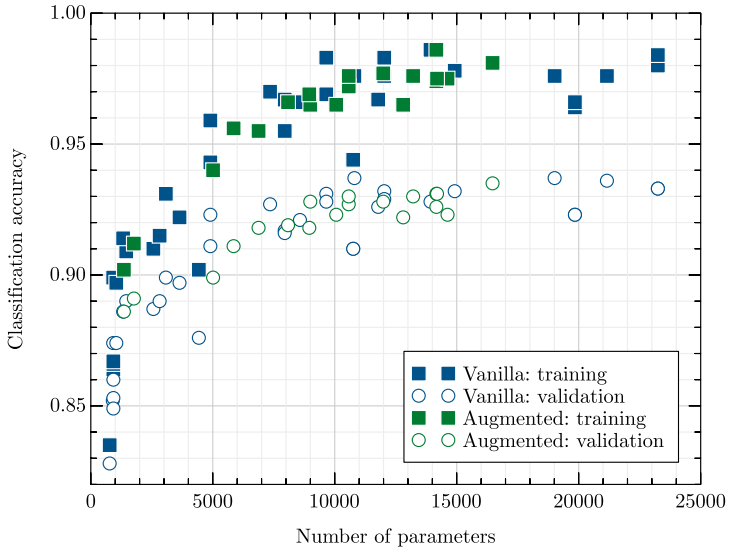


Figure 14: Plot of classification accuracy versus number of parameters for vanilla and augmented NODE networks. Each pair of square and hollow circle point represent a NODE model with a specific set of hyperparameters, trained for 15 epochs. The hyperparameters are randomly distributed in predetermined ranges. As expected, accuracies on the training dataset are general better than the validation dataset. Notably, model performance seems to correlate strongly with model size, while we expected to find more variation from the varying hyperparameters. In addition, performances between vanilla and augmented models are effectively the same given model size.

An shown in Figure 14, performance of vanilla and augmented NODE models are similar given any model size. As ANODEs can be seen as a generalization of vanilla NODEs (with the latter as a specialization with zero augmented channels), it means that the additional hyperparameter of augmented channels has no impact on performance beside influence the number of trainable parameters in the model. A reason for this may lay in our architecture design. Given the number of channels is variable in our case, we do not observe an significant decrease in performance by cutting said number any more beside the reduction in model size. Hence, the number of channels may not be a bottleneck in performance. This is in contrast with the case of the concentric hypersphere problem discussed in Section 4.1, where the input dimension is a limiting factor of the problem. Our nonlinear downsampling layers may have effectively augmented the input by expanding channels. Nonetheless, the additional flexibility is still a desirable properties of ANODEs, as it can further optimize other properties such as the time cost of training. We leave this as a potential direction for further research.

[8]Admittedly, learning rate were not strictly controlled for, and instead generally chosen to be the best performing ones at the end of training. In practice, the set of learning rates is mostly 0.001, and the rest are mostly between 0.0001 and 0.001.

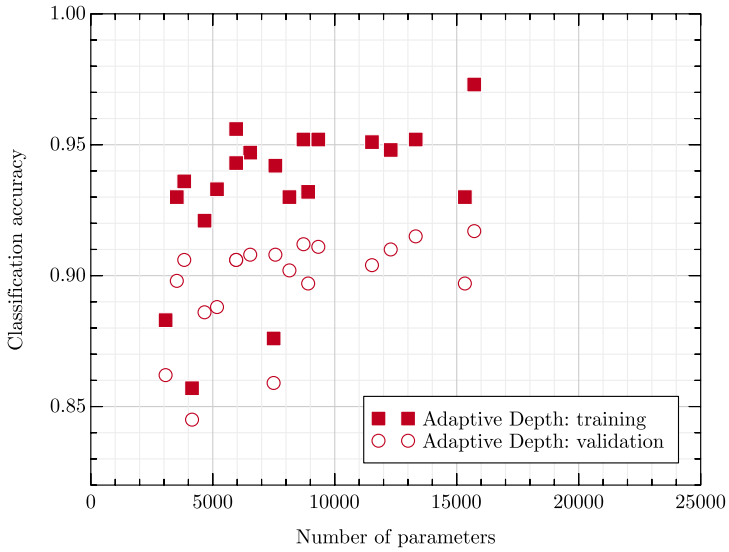


Figure 15: Plot of classification accuracy versus number of parameters for adaptive depth NODE networks. Models has randomly distributed hyperparameters and are trained for 10 epochs. The correlation between accuracy and size is much weaker for AD models than for vanilla and augmented models.

AD NODES demonstrate a weaker correlation between model size and accuracy, as seen in Figure 15 with a noisier distribution. Further empirical investigations suggest that no hyperparameter consistently affect model accuracy beyond contributing to model size. We hypothesize that our AD NODE architecture may be more sensitive to initial parameters and has complex loss landscape (in the sense of non-convexity). The latter point is supported by the fact the AD NODE models take significantly longer to train, as shown in Figure 16.

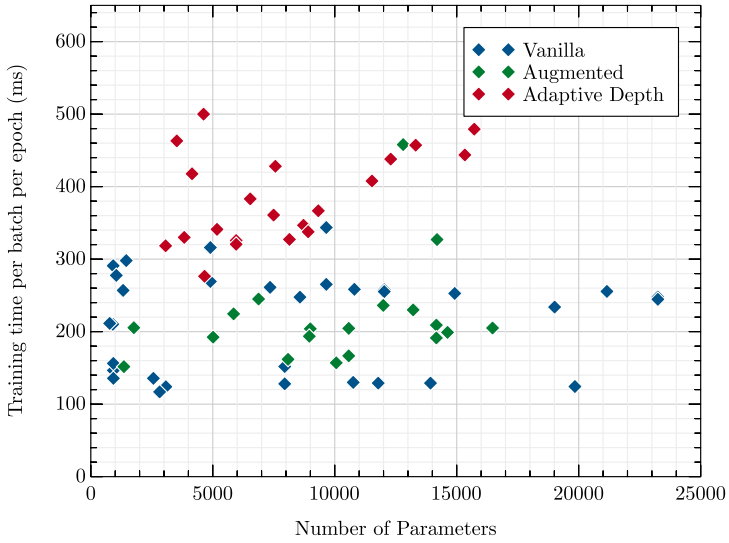


Figure 16: Plot of average model training time per minibatch versus model size. Although other hyperparameters affecting training time were not controlled for, they are randomly distributed in roughly the same range such that the contrast in the figure is still meaningful. And the contrast is clear: AD NODEs (red) takes much longer to train than vanilla (blue) and augmented (green) NODEs. The latter two have similar training time.

A second evidence pointing towards complex loss landscape comes from the learning curve. Informally, a learning curve is a descent down the loss landscape. As seen in Figure 17, AD models' learning curve tend to be jagged compared to that of vanilla and augmented models, hinting at rougher paths in AD model's landscape.

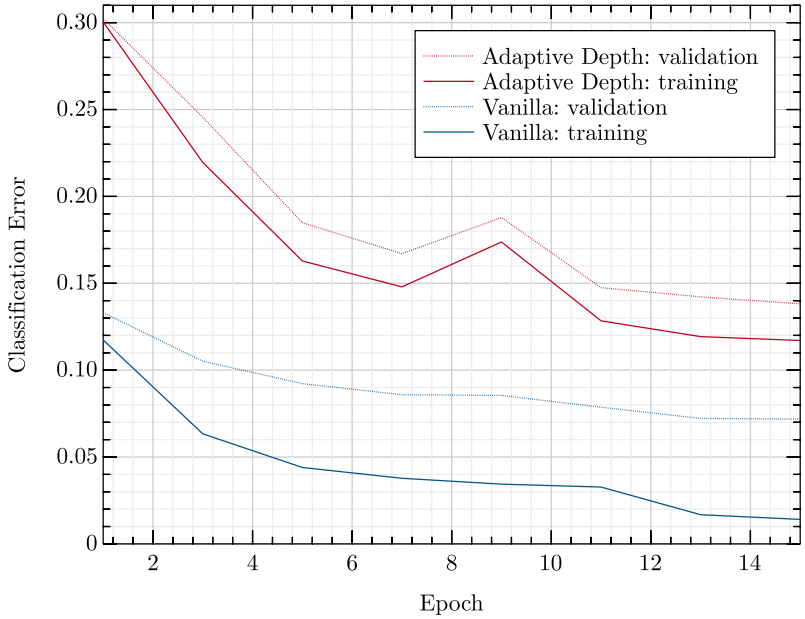


Figure 17: Plot of typical learning curves between vanilla and adaptive depth NODE models. AD models tend to have noisier learning curves with temporary rises and falls in error rate, while vanilla models, as well as augmented models, usually show a smooth reduction in error.

For our implementation and use case, AD models generally have worse performance than vanilla and augmented models if trained for the same number of epochs. If our hypothesis about AD's complex landscape holds then it is likely the reason behind this loss in performance. While AD models are technically also a generalization of vanilla models (whose depth-deciding function is just constant), the additional dimension of versatility can complicate a problem for the optimizer, which is reflected in complex loss landscape.

6 Conclusion

In this work, we discussed the theoretical aspect of Neural ODE and its connection to ResNet. We also focus on two extensions of NODEs, namely augmented and adaptive depth NODEs. Based on an existing implementation, we built our own architectures of the three methods. For each method, we trained a large number of models similar in architecture over a set range of hyperparameters, and reported observations and analysis. We found that for our model design and dataset, augmented NODE models perform similar to vanilla models with the same number of model parameters, while adaptive depth NODE models tend to perform worse. We hypothesis that AD NODE complicates the loss landscape compared to the other two methods. Our work focuses on a narrow range of architectures for this specific ECG dataset; future work can expand upon said range and make more general statements about the investigated methods.

References

[1] https://www.kaggle.com/datasets/shayanfazeli/heartbeat

[2] https://github.com/abaietto/neural_ode_classification

[3] Chen, R. T. Q., Rubanova, Y., Bettencourt, J., & Duvenaud, D. (2018). 'Neural Ordinary Differential Equations'. arXiv. https://doi.org/10.48550/ARXIV.1806.07366

[4] Kidger, Patrick, Ricky T. Q. Chen, Terry J. Lyons. ''Hey, that's not an ODE': Faster ODE Adjoints with 12 Lines of Code'. CoRR abs/2009.09457 (2020): n. pag. Web.

[5] Bilos, Marin. 'Neural Flows: Efficient Alternative to Neural ODEs'. CoRR abs/2110.13040 (2021): n. pag. Web.

[6] Kelly, Jacob. 'Learning Differential Equations that are Easy to Solve'. CoRR abs/2007.04504 (2020): n. pag. Web.

[7] Dupont, Emilien, Arnaud Doucet, & Yee Whye Teh. 'Augmented Neural ODEs'. 2019. Web. https://arxiv.org/abs/1904.01681

[8] Massaroli, Stefano et al. 'Dissecting Neural ODEs'. 2020. Web. https://arxiv.org/abs/2002.08071

Appendix

6.1 A Discussion on Integration Time Span

Many Neural ODE implementations, including the MNIST example from the official library *torchdiffeq*, has made the seemingly arbitrary choice of $[0, 1]$ as the solver time span. Since little explanation can be found, here describes own interpretation of the choice. We informally argue that, under some generous constraints on interal NN, the integration time span can be chosen arbitrarily without change in performance.

Say we have an ODE IVP problem with $\frac{d\mathbf{x}}{dt} = f(\mathbf{x}(t), t)$ and $\mathbf{x}_0 = \mathbf{x}(t = 0)$, and we want to find $\mathbf{x}(t = 2)$. Beside integrating f by the interval $t \in [0, 2]$, we can integrate a modified ODE by the interval $[0, 1]$ and arrive at the same solution by a transformation on the time variable:

$$\int_0^2 f(\mathbf{x}(t), t)dt = \int_0^1 2f(\mathbf{x}(t'), t')dt' \tag{6.1}$$

with $t' = t/2$. In general, to change an arbitrary integration span to $[0, 1]$, we have make a linear transform to t, $t' = mt + b$, and multiple the derivative function by $1/m$. It can be shown that for many NNs, if it can model $f(\mathbf{x}(t), t)$, it can also model $f(\mathbf{x}(mt + b), mt + b)/m$. First, there is the linear transformation on the input time; many NN layers already linearly transform their inputs, so they can fully express the additional linear transformation on time. The other change is the scaling of output by $1/m$. This can be achieved if the final layer is linear, i.e. without a non-linear activation function, with the weight and bias scaled accordingly.

Our NODE architecture indeed satisfies both requirements: the NODE block does not take time as an input, thus doesn't require the first layers to be linear, and the final layer is a convolution without nonlinear activation. Thus, setting the time span to $[0, 1]$ should not impact expressibility or performance. Some models however does not, such as the *torchdiffeq*'s MNIST model. Regardless, it can still be argued that most neural network, with linear or nonlinear layers, are flexible enough to approximate such a linear transformation in time variable, such that using the time span $[0, 1]$ make no significant difference in performance.

It is worth mentioning that our AD NODE implementation utilizes this transformation of integral bounds to realize adaptive depth. To train the AD models in batches would require ODE solver to integrate parts of the congregated, high-dimension arrays by different time spans, which our used library does not support. Alternatively, training AD model by each single case is impractical in time cost. Our implementation instead factors the derivative array, output of the internal network, by batched array multiplication, which has CUDA support and can be done efficiently. The solver then integrates the modified derivatives on $t \in [0, 1]$. This trick only works because our NODE block is autonomous, i.e. the internal derivative network does not take a time variable input.