FPT AI Residency: Literature Review

[*]Neural Ordinary Differential Equations [1]

1st Dung T. Vo *Artificial Intelligence Center*
FPT Software
Ho Chi Minh City, Viet Nam
dungvt19@fsoft.com.vn

December 3, 2021

[1]Identify applicable funding agency here. If none, delete this.

Abstract.

Machine learning and artificial intelligence have evolved over seven decades. During that time, many various algorithms were invented and along with improvements in the development process. Besides the development of computer hardware, computing tools are increasingly powerful, and they allow people to have models with millions of parameters. One of the significant milestones that can note in the history of machine learning is artificial neural networks. They give the model the ability to learn any data based on the ability to approximate every mapping with the right design and parameters for each of them. In other words, Neural Networks are universal functions approximators (Nielsen, 2019). However, in NeurIPS 2019, (Chen et al., 2019) introduced a breakthrough named Neural Ordinary Differential Equations (Neural ODE). Neural ODE is a new family of neural networks. In old methods, we always tried to approximate mapping f by a networks with several one-by-one layers $h_{t+1} = h_t + f(h_t, \theta_t)$, but the authors found a new way, they tried to add more layers and took smaller steps. Neural ODE uses differential equations to solve the output of each hidden state and use the outcome to be the input to compute the post-hidden state. The birth of Neural ODE opens up a new direction for scientists, who now have a new family of neural networks built on the knowledge of differential equations - a long-standing field of mathematics, has been developing for more than 120 years. NeuralODE is expected to bring many interesting results because the amount of knowledge of differential equations applied to Neural Networks is still minimal.

Contents

List of Tables

List of Figures

Chapter 1

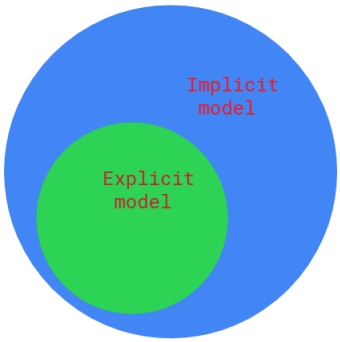
Introduction

Original deep networks is always try to approximate an explicit form of the mapping $f : \mathcal{X} \to \mathcal{Y}$. This brings us a nice technique for training various deep architect for specific task. There are some advantages still remain in the explicit form network. Such discrete models present potential difficulties when we apply it to data sets. Especially time series data can be mentioned, this kind of data prefer an continuity property rather than discrete property. (Florence et al., 2021) Experiments show that implicit models outperform explicit models in Deep Reinforcement Learning. In all research directions, researchers and scientists are all trying to apply broader knowledge more general than current results to improve an experiment and theory. Fig. 1.1 describe the generalization of implicit model. The explicit model such as ResNet, EfficientNet,... require the depth of network, and the parameter of those architectures linearly increases follow the depth. The greater the depth, the larger the number of parameters and the increased memory in the training process. Whereas implicit models can implement autograd by $\mathcal{O}(1)$ memoy Explicit model calculate the state of a system at a later time from the state of the system at the current time, while implicit model find a solution by solving an equation involving both the current state of the system and the later one. So that, Implicit model require an extra computation (solving the above equation), and they can be much harder to implement.

We have too many explicit networks built and introduced, Residual networks, Efficient networks, others. They are all calculated by forward and backward steps with data optimizers to approximate a function f mapping from data X to label y. Consequently, implicit models are studied and tested to give improved results compared to previous traditional explicit learning models. The crux of an implicit layer is that instead of specifying how to compute the layer's output from the input, we set the conditions that we want the layer's work to satisfy. That is, if we were to write explicit layers (with input $x \in \mathcal{X}$ and output $z \in \mathcal{Z}$) as an application of some explicit function $f : \mathcal{X} \to \mathcal{Z}$

$$z = f(x)$$

Figure 1.1: The figure show implicit model a more general case of explicit model



Then an implicit layer would instead be defined via a function $g : \mathcal{X} \times \mathcal{Z} \to R^n$ which is a joint function of both x and z, and where the output of the layer z is required to satisfy some constraint.

$$z \text{ satisfy } g(x, z) = 0$$

The notation $g(x, z)$ indicates a simple algebraic function. Still, in practice, this same formalism can capture differential equations that lead to various kinds of Neural Differential Equations.

Although appearing very early and there have been many works with significant results, the implicit models such as the Secondary Differential Equations that are focused on in this article still have many interesting things and many directions. for research, because the development of calculus and differential equations has spanned more than 120 years, and current work uses only a small amount of knowledge of those enormous results.

Some open questions and further directions could be:

- Apply the Neural CDE for various type of time series datasets.
- Increase the speed of Neural Differential Equation model.
- Reduce the number of model parameters
- Try to use other type of Differential Equation that may have universal approximator.

Chapter 2

Background

2.1 Neural Ordinary Differential Equation

2.1.1 Motivation

In Residual Networks, the forward step will evaluate the hidden state by residual block (He et al., 2015). Suppose we have 3 residual blocks with input x, results of blocks are $h_1, h_2, h_3, ..., h_n$ and h_n is the output of our network. $\theta_1, \theta_2, \theta_3, ...\theta_n$ are the parameters of each block. Consider x as h_0 We will have h_n as follow:

$$h_1 = h_0 + f(h_0, \theta_1)$$
$$h_2 = h_1 + f(h_1, \theta_2)$$
$$h_3 = h_2 + f(h_2, \theta_3)$$
$$...$$
$$h_n = h_{n-1} + f(h_{n-1}, \theta_n)$$

We can see the output of l-th residual blocks was calculate by the result of $l-1$ steps before.
So, the question come to us here is, what if we don't try to take "an integer" step, but we take "a real number" step. Or more precisely, what if we try to learn "adaptive" steps instead of fixed steps? To learn this question, we can consider the form of above Residual Network like the following way. We are not going to find h_{l+1} base on h_l any more. Now we base on h_l and try to find $h_{l+\varepsilon}$ with $\varepsilon \to 0$, the input of model is h_0 - initialize conditions of ODEs or known as our "data point", and finally the output of our models will be at some $t = T$. In form of equation we got

$$h_{t+\varepsilon} = h_t + f(h_t, \theta_t)$$

Figure 2.1: Normal block and Residual block from (Zhang et al., 2021)

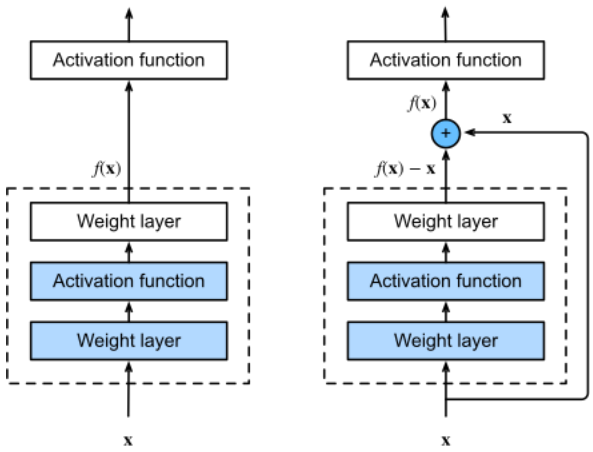


Figure 2.2: *Left*: A Residual network defines a discrete sequence of finite transformations. *Right*: A ODE network defines a vector field, which continuously transforms the state. *Both*: Circles represent evaluation locations. (Chen et al., 2019)

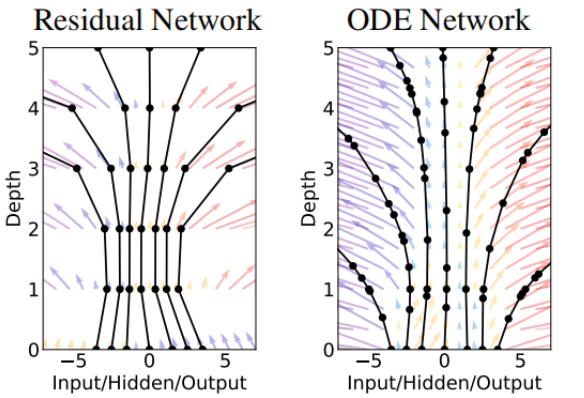


Figure 2.3: The way NODE produce the output from an input (known as initial condition in scope of ODEs

Consider the t as also a parameters, we have.

$$h(t+\varepsilon) = h(t) + f(h(t), t, \theta) \tag{2.1}$$

$$\frac{d(h(t))}{dt} = f(h(t), t, \theta) \tag{2.2}$$

Now we literally have a Ordinary Differential Equation (ODE) to solve. But before going to solve this ODEs, note that we just need a way to compute $h(T)$ (output of model) base on $h(0)$. Base on (2), we can easily show that

$$h(T) - h(0) = \int_0^T f(h(t), t, \theta)dt$$

$$h(T) = \int_0^T f(h(t), t, \theta)dt + h(0)$$

Let take a look on how the Neural Ordinary Differential Equations (NODEs) will forward a data point. Suppose the form $f(h(t), t, \theta)$ is easy to solve (For example: $f(h(t), t, \theta) = h(t) + 2$) So we can easily take the integral and have explicit output $h(T)$ from input data $h(0)$. Nevertheless, in practice, the form of $f(h(t), t, \theta)$ will usually be pretty bad and complex (in our language, this f will be a neural network) for us to calculate the integral explicitly.

So, we will need an efficient way to solve those Differential Equations. Fortunately, with more than 200 years of development, there are many ways to solve or approximate the result of Differential equations based on initialized conditions.

The most popular way in Neural Ordinary Differential Equations is to use various numerical methods such as Runge-Kutta, Euler, Dopri, others.

2.1.2 Preliminaries

As mentioned above, a Residual Block can be seen as the discrete approximation of ODE by setting the discretization to be one. When the discretization step approaches zero, it yields a new family of neural networks - Neural Ordinary Differential Equations.

Formally, in Neural ODEs, the relation between input and output is characterized by the following set of equations:

$$\frac{d\mathbf{z}(t)}{dt} = f_\theta(\mathbf{z}(t), t), \quad \mathbf{z}(0) = \mathbf{z}_{\text{in}}, \quad \mathbf{z}_{\text{out}} = \mathbf{z}(T) \tag{2.3}$$

where $f_\theta : R^d \times [0, \infty) \to R^d$ denotes the trainable layers parameterized by weights θ and $\mathbf{z} : [0, \infty) \to R^d$ represents the $d-$dimensional state of Neural ODEs. We assume the f_θ will be continuous in t and globally Lipschitz continuous in $\mathbf{z}$.

Given the input $\mathbf{z}_{\text{in}}$, the output $\mathbf{z}_{\text{out}}$ can be computed by solving the ODE in (3). If T is fixed, $\mathbf{z}_{\text{out}}$ just depends on input $\mathbf{z}_{\text{in}}$ and the dynamic f_θ. Therefore

$$\mathbf{z}_{\text{out}} = \mathbf{z}_0 + \int_0^T f_\theta(\mathbf{z}(t), t)dt = \phi_T(\mathbf{z}_{\text{in}}, f_\theta)$$

2.1.3 Gradient of scalar-valued Loss Function

In Machine Learning, the main issue is how the Machine can learn by updating parameters. In the origin architecture like Neural Networks, the parameters is updated by the optimizer after take gradient following be each parameters. For example if the optimizer was Gradient Descent so the parameter will update following $\theta = \theta - \alpha\nabla_\theta f_\theta(\mathbf{z}_{\text{in}})$ with $\alpha \in R$ is some learning rate. And the gradient for each parameter in deep networks will be theoretically calculated following chain rule. For more specific, suppose we have a network with two hidden layers h_1, h_2 and the parameters we want to calculate is w_{12} so we will have

$$\frac{\partial \mathcal{L}}{\partial w_{12}} = \frac{\partial \mathcal{L}}{\partial h_2}\frac{\partial h_2}{\partial h_1}\frac{\partial h_1}{\partial w_{12}}$$

But the main problem in training NODEs or any continuous-depth networks is high-memory cost. There can be up to hundreds state that we need compute backward.
In ODEs, we treat ODE solver as a black box, and compute gradients using the *adjoint sensitivity method* (Pontriagin et al., 1962). This approach computes gradients by solving a second, augmented ODE backwards in time, and is applicable to all ODE solvers. This approach scales linearly with problem size, has low memory cost, and explicitly controls numerical error. (Chen et al., 2019).
Let the scalar-valued loss function of training process is $\mathcal{L}$.

$$\mathcal{L}(\mathbf{z}(t_1)) = \mathcal{L}\left(\mathbf{z}(t_0) + \int_{t_0}^{t_1} f(\mathbf{z}(t), t, \theta)dt\right) \tag{2.4}$$

$$= \mathcal{L}(\text{ODESolve}(\mathbf{z}(t_0), f, t_0, t_1, \theta)) \tag{2.5}$$

For the optimizers' work, we need the gradient of $\mathcal{L}$ respect to θ e.g. $\frac{\partial \mathcal{L}}{\partial \theta}$. The first step is to determine how the gradient of the loss depends on the hidden state at each instant time. This quantity called the "*adjoint*" $\mathbf{a}(t) = \frac{\partial \mathcal{L}}{\partial \mathbf{z}(t)}$. To calculate this quantity, we can prove the following equation:

$$\frac{d\mathbf{a}(t)}{dt} = -\mathbf{a}(t)^\top \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \mathbf{z}} \tag{2.6}$$

Note that the (5) is also a differential equation. By calling the ODE Solver, we can have the $\mathbf{a}(t)$ value at any t that we want. This solver will run backwards, starting from initial value of $\frac{\partial \mathcal{L}}{\partial \mathbf{z}(t_1)}$. Solving this ODE requires the value of $\mathbf{z}(t)$ along its entire trajectory. However, we can recompute $\mathbf{z}(t)$ backward in time together with the adjoint, starting from its final value $\mathbf{z}(t_1)$

Why we need to "determine how the gradient of the loss depends on the hidden state at each instant time"? Because computing the gradients with respect to parameters θ depends on both $\mathbf{z}(t)$ and the adjoint $\mathbf{a}(t)$. We need compute following integral.

$$\frac{dL}{d\theta} = -\int_{t_1}^{t_0} \mathbf{a}(t)^\top \frac{\partial f(\mathbf{z}(t), t, \theta)}{\partial \theta} dt \tag{2.7}$$

To compute (6) we just only have to call ODE Solver with initial valued $\mathbf{0}_{|\theta|}$. Algorithm 1 was quoted in (Chen et al., 2019) show how to construct the necessary dynamics, and call an ODE Solver to compute all gradients at once.

Table 2.1: (Chen et al., 2019)Performance on MNIST. From (LeCun et al., 2001)

	Test Error	#Param	Memory
1-Layer MLP	1.60%	0.24M	-
ResNET	0.41%	0.60 M	O(L)
RK-Net	0.47%	0.22M	O(L)
OED-Net	0.42%	0.22M	O(1)

Table 2.2: Performance on MNIST

	Test Error	#Param
ResNet	0.28%	0.141 M
ODE-Net	0.24%	0.142 M

2.1.4 Experiments

Authors' experiment

This section was quoted from (Chen et al., 2019) - Section 3. The author experiments with a small residual network that downsamples the input twice then apply six standard residual blocks (He et al., 2016), which are replaced by an ODESolve module in the ODE-Net variant. The author also tests a network with the same architecture but where gradients are backpropagated directly through a Runge-Kutta integrator, RK-Net. Table 1 shows test error, number of parameters, and memory cost.

My experiment

In my setup with ODENet, I tried to build a network with the same parameters between ResNet and ODENet and train on MNIST. I found something awe-inspiring that the training process of ODENet will be slower than ResNet around three times with deterministic ODE solvers like Euler Method or Runge-Kutta (rk4). Furthermore, the training will last around fifty times slower than ResNet with the adaptive step ODE Solvers with the same setup. One more thing is that the convergence rate of ODENet was usually slower than ResNet. My experiment builds on the instruction from (Yan et al., 2021).

I also tried to run two architecture on CIFAR-10 and SVHN dataset and got the following results in Table 2.3

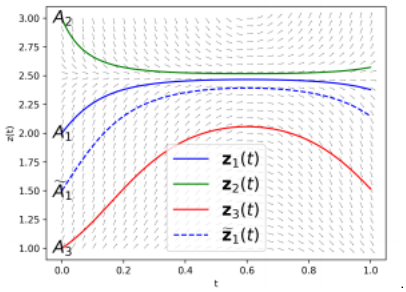
When running and testing the model on various tries, I have found something we must note for ODE-Net

- About the depth of the ODE Network. We can explicitly define the depth of the ODE-Net, especially when the ODE Solvers use the adaptive step solvers like Dormand–Prince method or the Bogacki-Shampine method. As the depth of ODENet is based on the number "the number of function evaluations" (NFEs).

Table 2.3: Performance on CIFAR-10 (2 first line) and SVHN (2 last line)

Dataset	Model	Test Accuracy	#Param
CIFAR-10	ResNet	94.216%	0.141 M
	ODE-Net	93.730%	0.142 M
SVHN	ResNet	98.454%	0.260M
	ODE-Net	95.349%	0.263M

Figure 2.4: No integral curves intersect. The integral curve starting from $\tilde{A}_1$ is always sandwiched between two integral curves starting from A_1 and A_3



- The scalar-valued loss function in those above experiments usually slowly converges. It does not like CNNs, which always converge at around epoch 40th to 45th per 100 epochs. ODE was converged in somewhere start from 90th epoch.
- In my experiments' setting, the ODENet takes about $2,5$ hours for the on $(3, 32, 32)$-image datasets with mini-batch size 256 with fixed-step solvers. With the adaptive-step solvers, the model takes about 24 hours for identical setups.
- Although the Figures show the ODENet can not be competitive compared to the CNNs, when I did the experiments, I think I didn't let the ODENet use all of its full power. In CIFAR-10, as the convergence rate of ODENet is slower than the CNNs, so with the same setup, the loss of ODENet had not converged yet, and I stopped the training process.

2.1.5 Some application to another Networks

In the general intuitive, we can suppose that for every kind of network or sub-network with the same shape at input and output, we can always replace those (sub-)networks with ODEs and perform a solver for them. So the application of ODE is wide. Also, many types of research were inspired by ODEs.

- First thing we need to count on the transformer, we have a published in Apr 2021 named ODE Transformer (Li et al., 2021)
- Graph Neural Ordinary Differential Equations (Poli et al., 2021)
- ODE in Variational Inference named ODE-VAE (Çağatay Yıldız et al., 2019)
- And many more works.

2.1.6 Robustness of Neural ODEs

Any deep structures require data to learn the parameters. But the data we use to learn and the data we meet when we apply the model to reality, we need also consider "never-met" data. So we need to examine the robustness of data by many methods. In this section, we will prove the robustness of ODENet is better than CNNs ones by experiments.

Intuitive

Consider following figure 2.4 from (Yan et al., 2021)

Theorem 1 (ODE integral curves do not intersect (Coddington & Levinson, 1955; Dupont et al., 2019)) *Let* $\mathbf{z}_1(t)$ *and* $\mathbf{z}_2(t)$ *be two solutions of the ODE in* (1) *with different initial conditions, i.e* $\mathbf{z}1(0) = \mathbf{z}_2(0)$. *In* (1), f_θ *is continuous in* t *and globally Lipschitz continuous in* $\mathbf{z}$. *Then, it holds that* $\mathbf{z}_1(t) = \mathbf{z}_2(t)$ *for all* $t \in [0, \infty)$.

Figure 2.5: The general architect of ODENet (Yan et al., 2021)

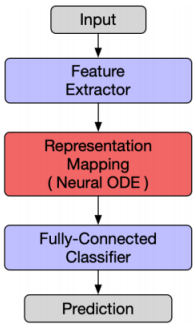


Table 2.4: Performance on MNIST across five distinct run

	$\sigma = 0$	$\sigma = 1e-5$	$\sigma = 50$	$\sigma = 75$	$\sigma = 100$
ResNet	99.6 ± 0.2	99.6 ± 0.1	61.2 ± 20.3	45.0 ± 16.5	35.1 ± 11.5
ODENet	99.8 ± 0.1	99.6 ± 0.1	95.9 ± 2.4	90.1 ± 5.8	79.1 ± 10.7

The Fig 2.4 illustrate Theorem 1. From Fig 2.4 We can see that the trajectories output of 2 data points that are close to each other will respectively be close to each other. Thus, we opine that due to this non-intersecting property, ODENet is intrinsically robust.

Experiments

To prove these intuitive results, we set up an experiment. In the experiments, we use the two types of perturbed data.

- Gaussian Perturbations
- Adversarial Data.

In the first experiments, we use the same architect for two ODENet and also CNNs. The general architect is described in Fig. 2.5. With CNNs, we use normal Feed Forward for Representation Mapping layers group instead of ODE solver.

With the first perturbed dataset, we trained on original MNIST (Deng, 2012) dataset and valid is perturbed by normal distribution $\mathcal{N}(0, \sigma)$ with σ is uniformly drawn from $25, 30, 40$. All valid and train set contain all of 60000 images. The implementation can be found *here*. Authors' implementation can be found *here*. The authors' results table can be found at Table 1 and Table 2 in (Yan et al., 2021). My experiments was running on $2 \times$ RTX 3090Ti GPUs with the batch size is 256, 100 epochs and learning rate is $1e-3$. Following Table 2.4

As the MNIST is a dataset, which is quite easy for the deep neural networks can approximate, I also tried two models on more complex dataset such as SVHN and CIFAR-10 and got the following results in Table 2.6 and Table 2.5

In adversarial attacked data, unfortunately that I have not finish the implementation for this part, so please checkout the experiments for Adversarial Attack following authors' sourceode was published on Github.

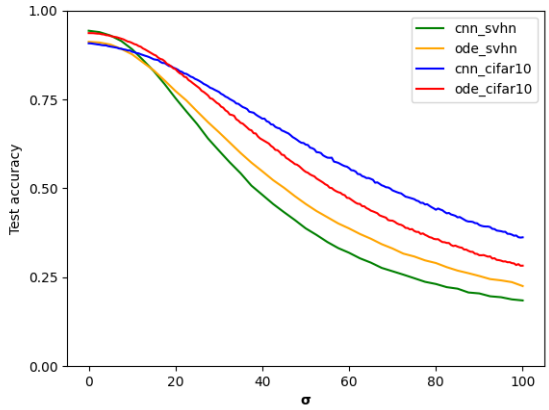
Table 2.5: Performance on SVHN in one run

	$\sigma = 0$	$\sigma = 1e-5$	$\sigma = 50$	$\sigma = 75$	$\sigma = 100$
ResNet	98.45	92.47	61.99	44.73	33.37
ODENet	95.35	91.52	67.66	52.61	41.33

Table 2.6: Performance on CIFAR-10 in one run

	$\sigma = 0$	$\sigma = 1e-5$	$\sigma = 50$	$\sigma = 75$	$\sigma = 100$
ResNet	72.71	72.71	66.87	60.74	53.61
ODENet	81.23	81.23	75.90	69.80	63.01

Figure 2.6: CIFAR10: Relation between σ and accuracy with the model structure in Tabel 9 of (Yan et al., 2021)



Based on the results I got in 2.4 2.6, 2.5 we can easily see that the difference between the $\sigma = 0$ and others is smaller with ODENet rather than CNNs. So by experiments, we proved the ODENet is more robust than what CNNs have.

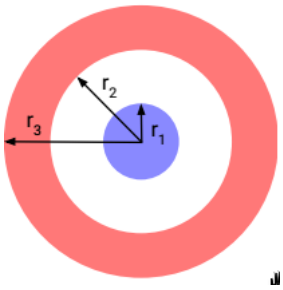
But, in another experiment, I tried to use the structure mentioned in Table 9 of (Yan et al., 2021), I trained for both CIFAR10 and SVHN dataset by this structure, Fig. 2.6 shows the relation when I tried to enhance standard deviation σ of the Gaussian noise with the model train on original CIFAR10 dataset of `torchvision`. With the result in 2.6 we can see that the robustness of ODEs may be unstable in some specific cases. The more complex the data is, the less robust the data performance is compared to the traditional Convolution network. Or this may be caused by the methods that we used in the training process. With my experiment, the method is the Euler method - the most straightforward case that approximates ODEs.

2.1.7 Advantages of Neural ODE

In this section, we will take a look into ODENet and Neural ODEs and their advantages compared to the same-architecture CNNs.

- From Fig. 2.2 we can see that the first advantage is smoothness. The networks contain more "linear" pieces, and the prediction trajectories will be more "near" with the ground-truth trajectories.
- Although some results about the CIFAR-10 dataset and SVHN dataset give the accuracy is a bit lower than what CNNs can take, and the experiments arguments setup may cause this. Until now, I did not try many different sets of arguments. I will try to enhance the number of epochs in the training process.
- In section 2.1.3, we showed how to compute gradients of a scalar-valued loss concerning all inputs of any ODE Solver, *without backpropagating through the operations of the solver*. Not storing any intermediate quantities of the forward pass allows us to train our models with constant memory cost as a function of depth, a major bottleneck of training deep models.
- NODE allows adaptive computation. The Euler method is perhaps the most straightforward method for solving ODEs. Besides the Euler method, one more general case is the Runge-Kutta method, which usually uses Runge-Kutta 4. However, those methods are "fixed-step" methods, with behaving a fixed

Figure 2.7: The figure illustrate $g(x)$ for the case $2d$-dimension



NFEs. We can use "adaptive-step" solvers like Dormand–Prince methods, which have adaptive NFEs in each epoch of the training process. Remember, the more NFEs, the more smooth trajectories.

- In (Chen et al., 2019) there are two more advantages in Normalizing Flows (Kobyzev et al., 2021) and continuous time-series models. In the framework of the article, I would like not to mention these two issues.
- **Uniqueness** Last but not least, Picard's existence theorem ((Coddington & Levinson, 1955)) states that the solution to an initial value problem exists and is unique if the differential equation is uniformly Lipschitz continuous in $\mathbf{z}$ and continuous in t. This theorem holds for our model if the neural network has finite weights and uses Lipshitz nonlinearities, such as `tanh` or `relu`. (Kim et al., 2021; Dasoulas et al., 2021) showed that the CNNs and Linear layers have Lipschitz property. So the original Deep Networks based on CNNs and Linear Layers allow us to apply the NODE methods.

2.1.8 Limitations

Training Time The first and the easiest limitation we can see throughout the article is the training time. With the training time up to 10 times of CNNs, the ODENet can not be applied in real-life problems with enormous data or too deep a Networks architecture. I tried to apply Neural ODE with about 624K parameters with the CIFAR-10 dataset, and it took about 2.5 hours to finish the first 100 epochs with 50K image data points, batch size 256. Meanwhile, ResNet50 with over 23M parameters can more time-efficiency training with the same setup. ODEs, until now, still can not be efficiently applied for the significant architect or extensive dataset and traditional CNNs. To enhance the training time, there are many methods to consider. One of them is that we can use Symmetry Methods for Differential and Difference Equations. This is also an exciting direction to research on. One more work of Mrs. Kidger about Faster ODE Adjoints via Seminorms (Kidger et al., 2021), this work reduces the NFEs by about 62%

Uniqueness In 2.1.7 I mentioned about the advantages when we can easily apply NODE for CNNs and Linear layers. At first glance this may seem like an advantage, but consider the situation described in the Fig 2.8, Due to the uniqueness output, If the trajectories of inputs intersects somewhere at time t_i so for any time t_j with $t_j > t_i$ the hidden state at time t_j of them will be the same. So there will be some function that the NODE can not represent (Dupont et al., 2019). E.g. g be a function such that, with $r_1 < r_2 < r_3$ (Fig 2.7)

$$\begin{cases} g(x) = 1 & \text{if} \quad ||x|| \leq r_1 \\ g(x) = -1 & \text{if} \quad r_2 \leq ||x|| \leq r_3 \end{cases} \tag{2.8}$$

The proof for representation of NODE with $g(x)$ is given in Appendix D (Dupont et al., 2019).

Or another case, Fig 2.9 show a straightforward issue that ODEs cannot solve and give us desired outputs. If there is a way to separate these two mappings for them, do not intersect each other. For easy visualization example the simplest case, the mappings blue is on the ground floor, and the mappings red is on the top floor, so we need an extra dimension. To solve this problem, remind the main idea of Kernel Methods, which mappings dataset of dimension d_1 to new dataset space with dimension $d_2 > d_1$. In ODEs, any solver will map the input to the output **in the same space**. Unlike original ODEs, authors of (Dupont et al., 2019) propose a new methods, they augment the space on which we learn and solve the ODEs they turn the $R^n \to R$ problem to $R^n \to R^{n+d}$. They call the new methods "Augmented Neural ODEs" (ANODEs)

Figure 2.8: Trajectories of ODE Solver with two different inputs have the same value at some time t_i will have the hidden state the same all at any time t_j which $t_i \leq t_j \leq T$

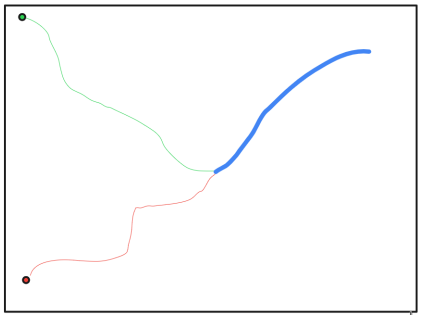
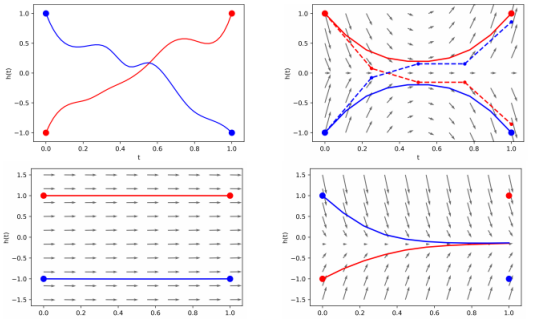


Figure 2.9: From (Dupont et al., 2019) (Top left) Continuous trajectories mapping -1 to 1 (red) and 1 to -1 (blue) must intersect each other, which is not possible for an ODEs. (Top right) Solutions of the ODE are shown in solid lines and solutions using the Euler method (which corresponds to ResNets) are shown in dashed lines. As can be seen, the discretization error allows the trajectories to cross. (Bottom) Resulting vector fields and trajectories from training on the identity function (left) and $g_{1d}(x)$ (right) (Dupont et al., 2019).



With ANODEs, they achieve better results in experiments, the results is reported in Table 2.7. For more details in experiments, please checkout (Dupont et al., 2019)

Recall that the condition for ODE solver can efficiently find output that is net ODEs function must have **uniqueness** outcome which means the function has to have Lipschitz property (Coddington & Levinson, 1955). (Kim et al., 2021; Dasoulas et al., 2021) also showed that Transformer (Vaswani et al., 2017) architecture does not have Lipschitz property. Intuitively not only Transformer but also a bunch of other potential architects does not have Lipschitz property. However, (Kim et al., 2021; Dasoulas et al., 2021) suggested two different ways to *re-formulate* the self-attention architect have Lipschitz property. For more specific details, please check out those papers.

Figure 2.10: Figure show the broken parts of data in training process, which separate 2 parts of label (Dupont et al., 2019).

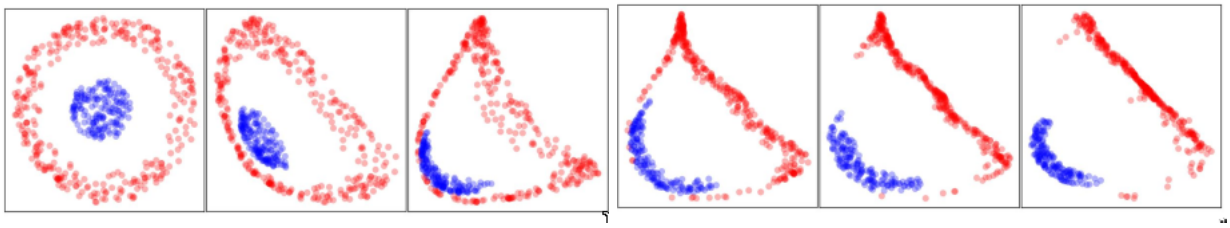


Table 2.7: Performance of ANODEs from (Dupont et al., 2019).

	NODE	ANODE
MNIST	96.4 ± 0.5	$\mathbf{98.2 \pm 0.1}$
CIFAR10	53.7 ± 0.2	$\mathbf{60.6 \pm 0.4}$
SVHN	81.0 ± 0.6	$\mathbf{83.5 \pm 0.5}$

2.2 Neural Stochastic Differential Equation

In this part, we will look at a new continuous neural network framework called Neural Stochastic Differential Equation (Neural SDEs) network, which naturally incorporates various commonly used regularization mechanisms based on random noise injection.

2.2.1 Motivation

In section 2.1.6 we have proved that ODENet takes advantage of robustness when compared with original CNNs models. In discrete neural networks, regularization mechanisms such as Dropout or Gaussian Noise are commonly used, which is missing in (Chen et al., 2019), those layers can take place inside a residual block or between layers to enhance the "randomness" of networks and to volume up the robustness of the model. It would be great if there were a way to present those layers in Differential Equation languages. It turns out that the keyword "randomness" in differential equations helps us to relate "stochastic processes" and "stochastic differential equations" - a fascinating perspective in math.

Neural SDEs framework can model various noise injections frequently used in discrete networks for regularization purposes, such as dropout and additive/multiplicative noise in each ODE block.

2.2.2 Some note on Stochastic Differential Equation

Brownian Motion

In SDEs' form 2.2.2, remind "$\mathbf{W}$ take an important place which represents the "stochastic" property of Differential Equation, without $\mathbf{W}$ an SDEs turn into ODEs." So what exactly "Brownian motion" is?

(Evans, 2012) A real-valued stochastic process $W(\cdot)$ is called a Brownian motion or Wiener process if

i. $W(0) = 0$

ii. $W(t) - W(s)$ is $N(0, t-s)$ for all $t \geq s > 0$

iii. For all times $0 < t_1 < t_2 < ... < t_n$, the random variables $W(t_1), W(t_2) - W(t_1)$

Notice that

$$E(W(t)) = 0, E(W^2(t)) = t \quad \text{for each time} t \geq 0 \tag{2.9}$$

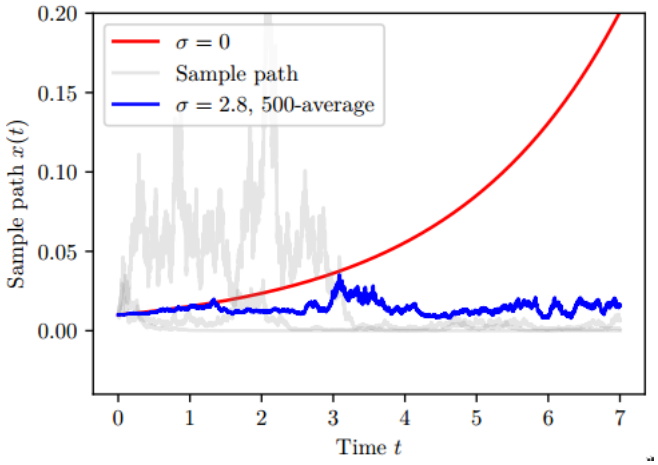
Form of a SDEs

Assume all measure theory conditions around stochastic differential equations are satisfied. Let Y is a R^n-valued stochastic process $\mathbf{X}$ is a solution of *Itô stochastic differential equation* In Equation (2.10), $\mathbf{W}$ is called "Brownian motion" (Sec 2.2.2), while $\mathbf{b} : R^n \times [0, T] \to R^n$ and $\mathbf{B} : R^n \times [0, T] \to R^n$ is two mappings

$$\text{(SDE)} \quad \begin{cases} d\mathbf{X} = \mathbf{b}(\mathbf{X}, t)dt + \mathbf{B}(\mathbf{X}, t)d\mathbf{W} \\ \mathbf{X}(0) = \mathbf{X}_0 \end{cases} \quad \forall t, 0 \leq t \leq T \tag{2.10}$$

The notation $\mathbf{W}$ take an important place which represent the "stochastic" property of Differential Equation, without $\mathbf{W}$ a SDEs turn into ODEs. So we can see that Stochastic Differential Equation is a more general case of Ordinary Differential Equation. By intuition, we can see that the Stochastic Differential Equation will produce more optimal results than what Ordinary Differential Equations can do.

Figure 2.11: Toy example. By comparing the simulations under $\sigma = 0$ and $\sigma = 2.8$, we see adding noise to the system can be an effective way to control x_t. Average over multiple runs is used to cancel out the volatility during the early stage (Xuanqing et al., 2019).



Intuition

In this part, we show the intuition that how the SDEs can slightly enhance the robustness of ODEs, and also before going to multi-dimensional SDEs, we also need to take a first look the SDEs in small dimension first. Consider the case $dx_t = x_t dt + \sigma x_t dW$ with W_t is standard Brownian motion, which mentioned in 2.2.2. Look the Fig 2.11, when the $\sigma = 0$ the differential equation become $dx_t = x_t$, now $x_t = c_0 e^t$ (blue line), when $\sigma \neq 0$ the differential equation become $dx_t = x_t dt + \sigma x_t dW$, and when we solve this SDEs $x_t = (1 - \frac{\sigma^2}{2})t + \sigma W_t$ The toy example in Figure 2.11 reveals that the behavior of solution paths can change significantly after adding a stochastic term. This example is inspiring because we can control the impact of perturbations on the output by adding a stochastic term to neural networks.

2.2.3 Apply Stochastic Differential Equations to re-formulate architecture

The Fig. 2.12 show the simple Neural Stochastic Differential Equation Network (SDENet). This model contain 3 parts

- The Convolution layers, we will call those layers is *Feature extraction* layers
- Followed is Neural SDE network (will be explained in 2.2.3). Recall that both Neural ODE and SDE are dimension preserving.
- Lastly, the Fully connected layer.

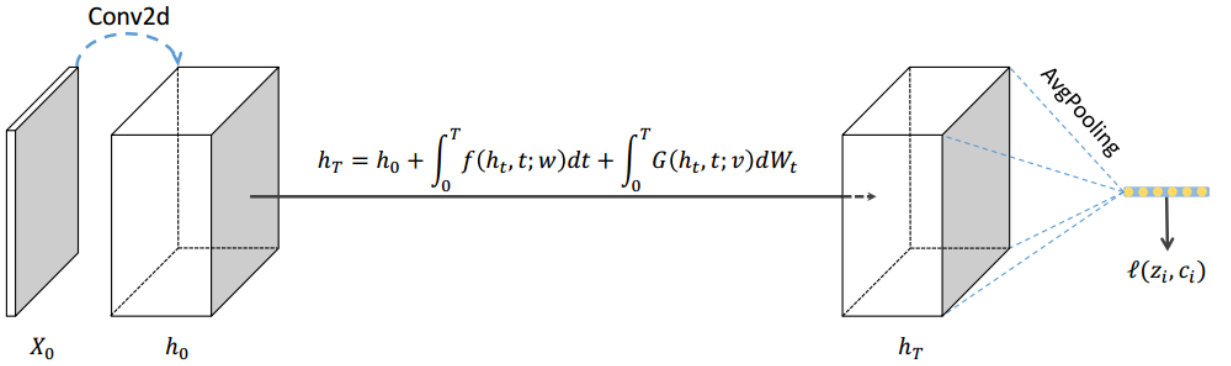
Modeling randomness in neural networks

In the Neural ODE system, a little bit perturbed in original input data point will be amplified in deep layers (as shown in Fig. 2.11), which makes the whole system unstable to input and prone to overfitting (in some of my experiments with Neural ODEs, this situation happen very likely). Randomness is an essential component in discrete-deep architecture (e.g., Dropout or Regularization). There is no work about adding randomness onto the continuous deep neural networks to solve this problem. To solve the problem, the authors propose to add a single diffusion term into Neural ODE as:

$$d\mathbf{h}_t = \mathbf{f}(\mathbf{h}_t, t, \omega_f)dt + \mathbf{G}(\mathbf{h}_t, t, \omega_G)d\mathbf{W}_t, \tag{2.11}$$

Where $\mathbf{W}_t$ is the standard Brownian motion, which is a continuous time stochastic process such that $\mathbf{B}_{t+\epsilon} - \mathbf{B}_t \sim \mathcal{N}(0, s)$ following (2.9), $\mathbf{G}(\mathbf{h}_t, t, \omega_G)$ is a transformation parameterized by $\omega_{\mathbf{G}}$. This equation is general case of many existing randomness injection model

Figure 2.12: Figure show the SDENet Architect. The initial value of SDE is the output of a convolutional layer, and the value at time T is passed to a linear classifier after average pooling (Xuanqing et al., 2019).



- **Gaussian noise injection:** Consider a simple case in (2.11) where $\mathbf{G}(\mathbf{h}_t, t, \omega_t)$ is a diagonal matrix, so we can model both additive and multiplicative noise as:

$$\begin{aligned} \text{Additive: } \boldsymbol{h}_t &= \boldsymbol{f}\left(\boldsymbol{h}_t, t; \boldsymbol{w}\right) \mathrm{d}t + \boldsymbol{\Sigma}(t) \mathrm{d}\mathbf{B}_t \\ \text{Multi..: } \mathrm{d}\boldsymbol{h}_t &= \boldsymbol{f}\left(\boldsymbol{h}_t, t; \boldsymbol{w}\right) \mathrm{d}t + \boldsymbol{\Sigma}\left(\boldsymbol{h}_t, t\right) \mathrm{d}\mathbf{B}_t \end{aligned} \tag{2.12}$$

  Where $\boldsymbol{\Sigma}$ is a diagonal matrix and its diagonal elements control the variance of the noise added to hidden states. This can be viewed as a continuous approximation of noise injection techniques in discrete neural network. For example, the discrete version of the additive noise can be written as:

$$\boldsymbol{h}_{n+1} = \boldsymbol{h}_n + \boldsymbol{f}\left(\boldsymbol{h}_n; \boldsymbol{w}_n\right) + \boldsymbol{\Sigma}_n \boldsymbol{z}_n \tag{2.13}$$

  with $\boldsymbol{\Sigma}_n = \sigma_n \boldsymbol{I}, \boldsymbol{z}_n \overset{\text{i.i.d.}}{\sim} \mathcal{N}(0, 1)$ which injects Gaussian noise after each residual block.

- **Dropout:** SDEs can also model the dropout layers, which randomly disables some neurons in the output of each residual blocks. Let see how us can modify dropout under Neural SDE language. First, we note that the discrete version of dropout

$$\boldsymbol{h}_{n+1} = \boldsymbol{h}_n + \boldsymbol{f}\left(\boldsymbol{h}_n; \boldsymbol{w}_n\right) \odot \frac{\boldsymbol{\gamma}_n}{p} \tag{2.14}$$

  Then we can interpret (2.14) at

$$\boldsymbol{h}_n + \boldsymbol{f}\left(\boldsymbol{h}_n; \boldsymbol{w}_n\right) + \boldsymbol{f}\left(\boldsymbol{h}_n; \boldsymbol{w}_n\right) \odot \left(\frac{\boldsymbol{\gamma}_n}{p} - \boldsymbol{I}\right) \tag{2.15}$$

  Where $\gamma_n \overset{\text{1.i.d.}}{\sim} \mathcal{B}(1, p)$ and $\odot$ indicates the Hadamard product. Note that we divide γ_n by p in (6) to maintain the same expectation. Furthermore, we have

$$\begin{aligned} \frac{\boldsymbol{\gamma}_n}{p} - I &= \sqrt{\frac{1-p}{p}} \cdot \sqrt{\frac{p}{1-p}} \left(\frac{\boldsymbol{\gamma}_n}{p} - \boldsymbol{I}\right) \\ &\approx \sqrt{\frac{1-p}{p}} \boldsymbol{z}_n, \quad \boldsymbol{z}_n \overset{\text{i.i.d.}}{\sim} \mathcal{N}(0, 1) \end{aligned} \tag{2.16}$$

  The boxed part above is approximated by standard normal distribution (by matching the first and second order moment). The final SDE with dropout can be obtained by combining (2.14), (2.15) and (2.16)

$$\mathrm{d}\boldsymbol{h}_t = \boldsymbol{f}\left(\boldsymbol{h}_t, t; \boldsymbol{w}\right) \mathrm{d}t + \sqrt{\frac{1-p}{p}} \boldsymbol{f}\left(\boldsymbol{h}_t, t; \boldsymbol{w}\right) \odot \mathrm{d}\mathbf{B}_t \tag{2.17}$$

- **Others:** Lu et al. (2020) include some formulated under Neural SDEs of other stochastic layers such as shake-shake regularization Gastaldi (2017) and stochastic depth Huang et al. (2016). Both of them are used as regularization techniques that work very similarly to dropout.

2.2.4 Back-propagating through SDE integral

Recall that any deep model needs to learn the set of parameters by an optimizer algorithm. Any modern deep architect requires that we must find a gradient of scalar-valued loss function $\mathcal{L}$ with respect to the parameter ω. So we in this part, we take a look into how they can perform the gradient of the scalar-valued loss function. Also, as Neural ODEs, the biggest problem of Neural SDEs is about memory efficiency problem when calculating through SDEs solver, if the SDE solver discrete the segment $[t_0, t_1]$ into N segments so that the SDEs will have N nodes and the memory cost will be $\mathcal{O}(N)$. Applying the Neural SDE to the great structure, which contains enormous parameters, becomes unavailable. To solve this issue, the first calculates the *expected loss* conditioning on the initial value $\mathbf{h}_0$, denoted as $L = E\left[\ell\left(\boldsymbol{h}_{t_1}\right) \mid \boldsymbol{h}_0\right]$. Then their goal is calculate $\frac{\partial L}{\partial \omega}$. They follow the theorem 2

Theorem 2 *For continuously differentiable loss $\ell(\mathbf{h}_{\mathbf{t_1}})$, we can obtain unbiased gradient estimator*

$$\frac{\widehat{\partial L}}{\partial \boldsymbol{w}} = \frac{\partial \ell\left(\boldsymbol{h}_{t_1}\right)}{\partial \boldsymbol{w}} = \frac{\partial \ell\left(\boldsymbol{h}_{t_1}\right)}{\partial \boldsymbol{h}_{t_1}} \cdot \frac{\partial \boldsymbol{h}_{t_1}}{\partial \boldsymbol{w}} \tag{2.18}$$

Moreover, if we define $\boldsymbol{\beta}_t = \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{w}}$, then $\boldsymbol{\beta}_t$ follows another SDE

$$\begin{aligned} \mathrm{d}\boldsymbol{\beta}_t = \left(\frac{\partial \boldsymbol{f}\left(\boldsymbol{h}_t, t; \boldsymbol{w}\right)}{\partial \boldsymbol{w}} + \frac{\partial \boldsymbol{f}\left(\boldsymbol{h}_t, t; \boldsymbol{w}\right)}{\partial \boldsymbol{h}_t}\boldsymbol{\beta}_t\right) \mathrm{d}t \\ + \left(\frac{\partial \boldsymbol{G}\left(\boldsymbol{h}_t, t; \boldsymbol{w}\right)}{\partial \boldsymbol{w}} + \frac{\partial \boldsymbol{G}\left(\boldsymbol{h}_t, t; \boldsymbol{w}\right)}{\partial \boldsymbol{h}_t}\boldsymbol{\beta}_t\right) \mathrm{d}\mathbf{B}_t \end{aligned} \tag{2.19}$$

We can see that β_t is similar to a_t (the adjoint), we can easily check that if $\mathbf{G} \equiv \mathbf{0}$ the β_t turns exactly to a_t

Similar to the adjoint method in Neural ODE, we will solve (2.19) jointly with the original SDE dynamics (2.11), this process can be done iteratively without memorizing the middle states, which makes it more memory efficient than autograd (Xuanqing et al. (2019)).

2.2.5 Robustness of Neural SDE

In 2.1.6, we proved by numerical experiments that the ODEs give better robustness on various architectures and datasets, but we did not mention about theoretical behind it. We have shown a case when the perturbation is significant. ODEs maybe not be robust like origin CNNs in some settings. Xuanqing et al. (2019) theoretically analyzed the stability of Neural SDE, they proved that the randomness term could indeed improve the robustness of the model against **small** input perturbation. This also explains why noise injection can improve the robustness in discrete networks. We also can apply this theoretical result for ODEs by setting the $\mathbf{G} \equiv \mathbf{0}$. The proof contains knowledge about **Measure Theory**, so I still can not understand clearly the author's idea. Please consider taking a look at that paper. The prove for the stability of Neural SDEs can be found in Xuanqing et al. (2019).

2.2.6 Experiments

Unfortunately, I have not implemented the Neural SDEs yet. To cover this part, I will use the authors' results (see Xuanqing et al. (2019)). Through out all experiments, they set $\mathbf{f}(\cdot)$ to be a neural network with several convolution blocks. As to $\mathbf{G}(\cdot)$ we have the following choices:

- **Neural ODE**: $\boldsymbol{G}\left(\boldsymbol{h}_t, t; \boldsymbol{\omega}_{\mathbf{G}}\right) = \mathbf{0}$
- **Additive noise**, when the diffusion term is independent of $\boldsymbol{h}_t$, here they simply set it to be diagonal $\boldsymbol{G}\left(\boldsymbol{h}_t, t; \boldsymbol{v}\right) = \sigma_t \boldsymbol{I}$
- **Multiplicative noise**, when the diffusion term is proportional to $\boldsymbol{h}_t$, or $\boldsymbol{G}\left(\boldsymbol{h}_t, t; \boldsymbol{w}\right) = \sigma_t \boldsymbol{h}_t$.
- **Dropout noise**, when the diffusion term is proportional to the drift term $\boldsymbol{f}\left(\boldsymbol{h}_t, t; \boldsymbol{w}\right)$, i.e. $\boldsymbol{G}\left(\boldsymbol{h}_t, t; \boldsymbol{v}\right) = \sigma_t \operatorname{diag}\left\{\boldsymbol{f}\left(\boldsymbol{h}_t, t; \boldsymbol{w}\right)\right\}$

Table 2.8: Evaluating the model generalization under different choices of diffusion matrix $\mathbf{G}(\mathbf{h}_t; t; v)$ introduced above. For the last three noise types, we search a suitable parameter σ_t for each so that the diffusion matrix $\mathbf{G}$ properly regularizes the model. TTN means testing time noise (Xuanqing et al. (2019))

Dataset	#Param	Accuracy@1 — w/o TTN			
		ODE	Additive	Multipli...	Dropout
CIFAR-10	115k	81.63	83.65	83.26	83.60
STL-10	2.44 M	58.03	61.23	**60.54**	61.26
Tiny-ImageNet	2.49 M	45.19	45.25	46.94	**47.04**
Dataset	#Param	Accuracy@1 — w/ TTN			
		ODE	Additive	Multipli...	Dropout
CIFAR-10	115k	-	83.89	83.76	**84.55**
STL-10	2.44 M	–	62.11	**62.58**	62.13
Tiny-ImageNet	2.49 M	–	45.39	46.65	**47.81**

Table 2.9: Testing accuracy results under different levels of non-adversarial perturbations.

Data	Noise type	mild ← Accuracy → severe				
		Level				
		1	2	3	4	5
CIFAR10	ODE	75.89	70.59	66.52	60.91	53.02
	Dropout	77.02	71.58	67.21	61.61	53.81
	Dropout+TTN	**79.07**	**73.98**	**69.74**	**64.19**	**55.99**
Tiny-ImgNet	ODE	23.01	19.18	15.20	**12.20**	**9.88**
	Dropout	22.85	18.94	14.64	11.54	9.09
	Dropout+TTN	**23.84**	**19.89**	**15.28**	12.08	9.44

In the first experiment, they show small noise helps generalization. However, note that our noise injection is different from randomness layer in the discrete case, for instance, dropout layer adds Bernoulli noise at training time, but the layer are then fixed at testing time; whereas their Neural SDE model keeps randomness at testing time and takes the average of multiple forward propagation.

We see that for all datasets, Neural SDE consistently outperforms ODE, and the reason is that adding moderate noise to the models at training time can act as a regularizer and thus improve testing accuracy. Based on that, we will obtain even better results if we continue testing time noise and ensemble the outputs. Table 2.8 give the authors' results.

The second experiment is on non-adversarial robustness. The corrupted datasets contain tens of defects in photography, including motion blur, Gaussian noise, fog,... (Hendrycks & Dietterich (2019)). Corrupted data is divided into five different levels. The authors ran Neural ODE and Neural SDE with dropout noise for each noise type and gathered the testing accuracy. Both models are trained on immaculate data, which means the corrupted images are not visible to them during the training stage, nor could they augment the training set with the same types of corruptions.

Next experiments is about checking the adversarial robustness. Clearly, this scenario is strictly harder than previous cases: by design, the adversarial perturbations are guaranteed to be the worst case within a small neighborhood. In their experiment they adopt multi-step $\ell_\infty -$ PGD attack Madry et al. (2019) (Fig. 2.13)

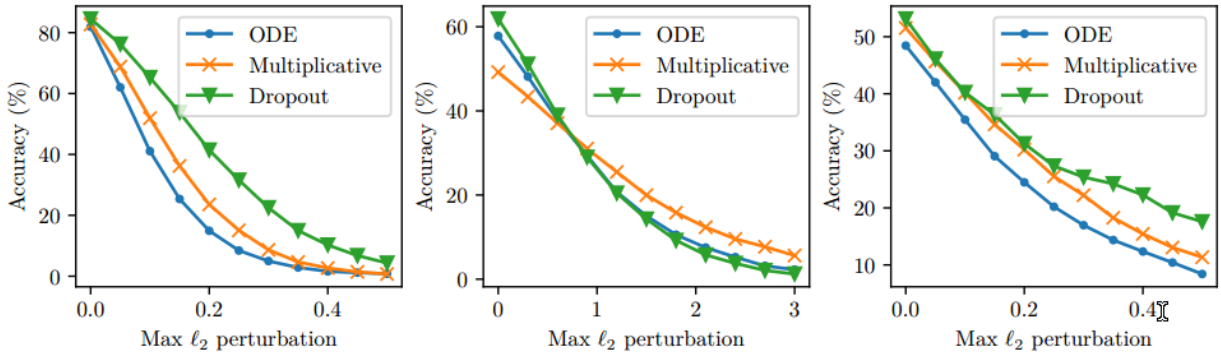
2.2.7 Advantages

The advantages of Neural SDE are shown in the above parts. To sum up, by both experiments and theoretical, we can easily see that Neural SDE can achieve better generalization and improve the robustness to both adversarial and non-adversarial noises compared with Neural ODE and original CNNs.

2.3 Neural Controlled Differential Equation

Since the beginning of the article, we have gone through many parts of differential equations in deep learning. The experiments performed in the above works are all experiments using categorical datasets. As mentioned

Figure 2.13: Xuanqing et al. (2019) Comparing the robustness against ℓ_2-norm constrained adversarial perturbations, on CIFAR-10 (left), STL-10 (middle) and Tiny-ImageNet (right) data. We evaluate testing accuracy with three models, namely Neural ODE, Neural SDE with multiplicative noise and dropout noise.



above, we can somehow change part of the network to a Neural ODE wherever there is a deep learning network. So some questions naturally arise here is:

- Whether for tasks involving time-series datasets, does ODE still work well?
- If they do not work well, why and is there an alternative?
- Is there any form of differential equation that can generalize to deep learning axes like Recurrent Neural Network or Long short-term memory (Sherstinsky, 2020)?

Time series data is a series of point data, usually consisting of allowed measurement links from the same source over a period of time. Time series are typically assumed to be generated at regularly spaced interval of time, and so are called regular time series. The data can include a timestamp explicitly or a timestamp can be implied based on the intervals at which the data is created. There is two main types of time series data.

- **Regular time series** Time series without an associated timestamp are automatically.
- **Irregular time series** is the opposite of a regular time series. The data in the time series follows a temporal sequence, but the measurements might not happen at a regular time interval. For example, the data might be generated as a burst or with varying time intervals. Account deposits or withdrawals from an ATM are examples of an irregular time series

The application of time series data is extensive, covering all fields, especially in which it is necessary to mention stock problems, market value prediction, customer behavior analysis, and so on. a lot more.

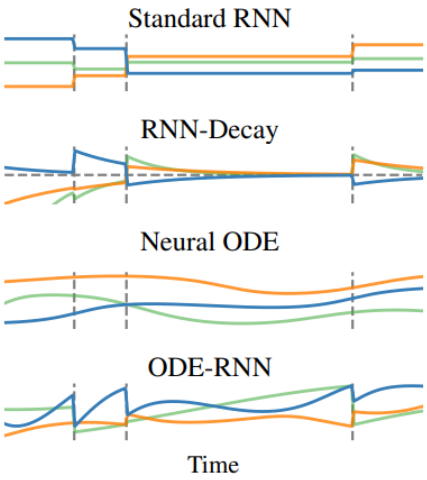
2.3.1 Motivation

Previous work on time series datatypes such as RNNs, GRUs, GRU-decay has had remarkable results, but as mentioned above, wherever there is a deep learning network, we can always replace the part (or maybe all) of that deep learning network with a continuous analog of deep network. (Rubanova et al., 2019) define continuous-time hidden dynamics defined by replace ordinary differential equations. They generalize state transitions in RNNs to continuous time dynamics specified by a neural network, as in Neural ODEs (Chen et al., 2019). After that, they use this for refine the Latent ODEs (Chen et al., 2019) by using the ODE-RNN as a recognition network. Both model naturally handle time gaps between observations, and remove the need to group observations into equally-timed bins (Rubanova et al., 2019). Since visualize the intuitive why the ODE-RNN and (refine) Latent ODE can improve the Standard RNNs, please consider to look Fig. 2.14

Fig. 2.14 shows us that four previous work on continuous analogue of time series tasks have some specific disadvantages:

- **Standard RNN** has a constants states between observations.

Figure 2.14: (Rubanova et al., 2019) Hidden state trajectories. Vertical lines show observation times. Lines show different dimensions of the hidden state. Standard RNNs have constant or undefined hidden states between observations. The RNN-Decay model has states which exponentially decay towards zero, and are updated at observations. States of Neural ODE follow a complex trajectory but are determined by the initial state. The ODE-RNN model has states which obey an ODE between observations, and are also updated at observations.



- **RNN-decay** has states which exponentially decay towards zero and are updated at observations. All the same between observations, so RNN-decay does not seem like generalize enough.
- **Neural ODE** has more complex states all overtime period, but those states are unique and were defined by the initial state, which we do not want whenever we work with time-series data.

Except the Neural ODE, the three remain **does not have continuity**, and each observation just can be calculated by the nearest observation-time state. When working with a time series dataset, we prefer the outcome based on all of the previous observations or all previous states. By intuition, this way will be more generalized than just looking into the nearest observation-time state.

In this paper, the authors demonstrate how Controlled Differential Equation may extend the Neural ODE model. Just as Neural ODEs are the continuous analog of a ResNet, the Neural CDEs (Kidger et al., 2020) is the continuous analog of an RNNs.

Authors also provide additional theoretical results showing that our model is a universal approximator and that it subsumes apparently-similar ODE models in which the vector field depends directly upon continuous data. However, in this article, we will not go deep into the theory behind Controlled Differential Equations because we need more knowledge about **Rough Analysis** to learn about Controlled Differential Equations.

2.3.2 Preliminaries

Suppose for simplicity that we have a fully-observed but potentially irregularly sampled time series $\mathbf{x} = ((t_0, x_0), (t_1, x_1), \ldots, (t_n, x_n))$, with each $t_i \in R$ the timestamp of the observation $x_i \in R^v$ and $t_0 < \cdots < t_n$.

Let $X : [t_0, t_n] \to R^{v+1}$ be the natural cubic spline with knots at $t_0, \ldots, t_n$ such that $X_{t_i} = (x_i, t_i)$. As x is often assumed to be a discretisation of an underlying process, observed only through $\mathbf{x}$, then X is an approximation to this underlying process.

Let $f_\theta : R^w \to R^{w \times (v+1)}$ be any neural network model depending on parameters θ. The value w is a hyperparameter describing the size of the hidden state. Let $\zeta_\theta : R^{v+1} \to R^w$ be any neural network model depending on parameters θ. Then we define the neural controlled differential equation model as the solution of the CDE

$$z_t = z_{t_0} + \int_{t_0}^{t} f_\theta(z_s)\, \mathrm{d}X_s \quad \text{for } t \in (t_0, t_n] \tag{2.20}$$

Model	Test Acc Dropped			Memory (MB)
	30%	50%	70%	
GRU-ODE	92.6% ± 1.6%	86.7% ± 3.9%	89.9% ± 3.7%	1.5
GRU-t	93.6% ± 2.0%	91.3% ± 2.1%	90.4% ± 0.8%	15.8
GRU-D	94.2% ± 2.1%	90.2% ± 4.8%	91.9% ± 1.7%	17.0
ODE-RNN	95.4% ± 0.6%	96.0% ± 0.3%	95.3% ± 0.6%	14.8
Neural CDE	**98.7% ± 0.8%**	**98.8% ± 0.2%**	**98.6% ± 0.4%**	**1.3**

Table 2.10: Test accuracy (mean ± std, computed across five runs) and memory usage on CharacterTrajectories. Memory usage is independent of repeats and of the amount of data dropped.

where $z_{t_0} = \zeta_\theta(x_0, t_0)$. This initial condition is used to avoid translational invariance. Analogous to RNNs, the output of the model may be taken to be the evolving process z, or the terminal value z_{t_n}, and the final prediction should typically be given by a linear map applied to this output. (Kidger et al., 2020)

Evaluating Neural CDE model

Evaluating the Neural CDE model is straightforward. Note that in 2.20, X does not need to be *natural cubic splines*, it can be any interpolation coefficient, the work on how we should choose specific X for a specific task, which X will be good (or bad) was published (see Morrill et al. (2021)). The most crucial property of X is that it must be differentiable as 2.20. In this case, we define

$$g_{\theta,X}(z,s) = f_\theta(z)\frac{\mathrm{d}X}{\mathrm{d}s}(s) \tag{2.21}$$

so that for $t \in (t_0, t_n]$,

$$z_t = z_{t_0} + \int_{t_0}^{t} f_\theta(z_s)\,\mathrm{d}X_s = z_{t_0} + \int_{t_0}^{t} f_\theta(z_s)\frac{\mathrm{d}X}{\mathrm{d}s}(s)\mathrm{d}s = z_{t_0} + \int_{t_0}^{t} g_{\theta,X}(z_s,s)\,\mathrm{d}s \tag{2.22}$$

Thus it is possible to solve the Neural CDE using the same techniques as for Neural ODEs.

2.3.3 Experiments

Unfortunately, the code for reaching X is quite complex, so I still can not finished the implementation for this paper, too. So in this part, I will propose the authors' experiments.

Authors benchmark the Neural CDE to variety previous models. These are: GRU-Δt, which is Gated Recurrent Unit with the time difference between observations additionally used at input; GRU-D, which modifies the GRU-Δt with learnt exponential decays between observations; GRU-ODE and ODE-RNN. Each model is run five times, and they report the mean and standard deviation of the test metrics.

Varying amounts of missing data on CharacterTrajectories

We begin with irregularly sampled time series. Consider the CharacterTrajectories dataset from the UEA time series classification archive (see Bagnall et al. (2018)). They run three experiments, in which we drop either 30%, 50% or 70% of the data. The observations to drop are selected uniformly at random and independently for each time series observations are removed across channels, so that the resulting dataset is irregularly sampled but completely observed. The randomly removed data is the same for every model and every repeat. The results are shown in Table 2.10. Neural CED outperforms every other model considered, and furthermore it does so whilst using an order of magnitude less memory.

Regular time series with Speech Commands

Finally they demonstrate the efficacy of Neural CDE models on regularly spaced, fully observed time series, where they might hypothesise that the baseline models will do better. They used the Speech Commands dataset. This consists of one-second audio recordings of both background noise and spoken words such as 'left', 'right', and so on.

Model	Test Accuracy	Memory usage (GB)
GRU-ODE	47.9% ± 2.9%	**0.164**
GRU-t	43.3% ± 33.9%	1.54
GRU-D	32.4% ± 34.8%	1.64
ODE-RNN	65.9% ± 35.6%	1.40
Neural CDE	**89.8% ± 2.5%**	0.167

Table 2.11: Test Accuracy (mean ± std, computed across five runs) and memory usage on Speech Commands. Memory usage is independent of repeats

Note: I tried to run the authors source code was publised on Github, with various combination of class from 2 class to 10 class. The results is quite identical to the Table 2.11

As we can see in the Table 2.11, the accuracy of Neural CDE significantly outperforms other models, Neural CDE also take a small memory usage due to the adjoint backpropagation (see Chen et al. (2019), Appendix B)

For further experiments and experimential details see Kidger et al. (2020) Appendix D.

2.3.4 Advantages

By experiment, we can see many advantage of Neural CDE for Time Series data. To sum up:

- Take advantage of data from previous states rather than nearest state. Neural CDE interpolate X to be continuous function contain all given observation, and the form in (2.20) show that when we compute a new state, we had used all the knowledge attain by the term $\mathrm{d}X$. This property is very useful in online learning task since the new data will share all knowledge with the data we known. Neural CDE for Online Prediction Tasks (see Morrill et al. (2021) research on X chosen and by experiment, they give some specific interpolation for the specific task.

- **Universal Approximation** is a famous theorem in CDEs that in some sense they represent general functions on streams. This may be applied to show that Neural CDEs are universal approximators, which we summarise in the following informal statement (see Kidger et al. (2020), Appendix B)

- **Evaluating** Since the CDE can be interpret as 2.22, we can use the Neural ODE techniques to apply to Neural CDE.

- **Partially observed data** One advantage of our formulation is that it naturally adapts to the case of partially observed data. Each channel may independently be interpolated between observations to define X in exactly the same manner as before.

2.3.5 Limitation

Speed of computation Neural CDEs were still roughly fives times slower than RNN models.

Number of parameters If the vector field $f_\theta : R^w \to R^{w\times(v+1)}$ is a feedforward neural network, with final hidden layer of size ω, then the number of scalars for the final affine transformation is of size $\mathcal{O}(\omega vw)$, which can easily be very large. In our experiments we have to choose small values of w and ω for the Neural CDE to ensure that the number of parameters is the same across models. We did experiment with representing the final linear layer as an outer product of transformations $R^w \to R^w$ and $R^w \to R^{v+1}$. This implies that the resulting matrix is rank-one, and reduces the number of parameters to just $\mathcal{O}(\omega(v + w))$, but unfortunately we found that this hindered the classification performance of the model.

Chapter 3

Progress Report and Further work

3.1 Learning Progress

Under the active guidance of the mentors, I feel that I have learned a lot, the figure below describes my six months of sticking with the program.

3.2 Further work

Because I started reading about Differential Equations in Deep Learning and Implicit Methods over the last 2 months, I can't have a specific research question yet. I have listed some directions that I and two mentors mentioned in meetings.

First of all, I will try to apply Neural CDE source code to bigger time series datasets to see how Neural CDE can be have on the pratical situation.

Figure 3.1: Six-month learning

Chapter 4

Conclusion

Bibliography

Bagnall, A., Dau, H. A., Lines, J., Flynn, M., Large, J., Bostrom, A., Southam, P., & Keogh, E. (2018). The uea multivariate time series classification archive, 2018.

Chen, R. T. Q., Rubanova, Y., Bettencourt, J., & Duvenaud, D. (2019). Neural ordinary differential equations.

Coddington, A., & Levinson, N. (1955). *Theory of Ordinary Differential Equations*. International series in pure and applied mathematics. McGraw-Hill Companies.
URL `https://books.google.com.vn/books?id=LvNQAAAAMAAJ`

Dasoulas, G., Scaman, K., & Virmaux, A. (2021). Lipschitz normalization for self-attention layers with application to graph neural networks.

Deng, L. (2012). The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, *29*(6), 141–142.

Dupont, E., Doucet, A., & Teh, Y. W. (2019). Augmented neural odes.

Evans, L. (2012). *An Introduction to Stochastic Differential Equations*. American Mathematical Society.
URL `https://books.google.com.vn/books?id=VYoCAQAAQBAJ`

Florence, P., Lynch, C., Zeng, A., Ramirez, O., Wahid, A., Downs, L., Wong, A., Lee, J., Mordatch, I., & Tompson, J. (2021). Implicit behavioral cloning.

Gastaldi, X. (2017). Shake-shake regularization.

He, K., Zhang, X., Ren, S., & Sun, J. (2015). Deep residual learning for image recognition.

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Identity mappings in deep residual networks.

Hendrycks, D., & Dietterich, T. (2019). Benchmarking neural network robustness to common corruptions and perturbations. *Proceedings of the International Conference on Learning Representations*.

Huang, G., Sun, Y., Liu, Z., Sedra, D., & Weinberger, K. (2016). Deep networks with stochastic depth.

Kidger, P., Chen, R. T. Q., & Lyons, T. J. (2021). "hey, that's not an ode": Faster ode adjoints via seminorms. In M. Meila, & T. Zhang (Eds.) *Proceedings of the 38th International Conference on Machine Learning*, vol. 139 of *Proceedings of Machine Learning Research*, (pp. 5443–5452). PMLR.
URL `https://proceedings.mlr.press/v139/kidger21a.html`

Kidger, P., Morrill, J., Foster, J., & Lyons, T. (2020). Neural controlled differential equations for irregular time series.

Kim, H., Papamakarios, G., & Mnih, A. (2021). The lipschitz constant of self-attention.

Kobyzev, I., Prince, S. J., & Brubaker, M. A. (2021). Normalizing flows: An introduction and review of current methods. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *43*(11), 3964–3979.
URL `http://dx.doi.org/10.1109/TPAMI.2020.2992934`

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (2001). Gradient-based learning applied to document recognition. In *Intelligent Signal Processing*, (pp. 306–351). IEEE Press.

Li, B., Du, Q., Zhou, T., Zhou, S., Zeng, X., Xiao, T., & Zhu, J. (2021). Ode transformer: An ordinary differential equation-inspired model for neural machine translation.

Lu, Y., Zhong, A., Li, Q., & Dong, B. (2020). Beyond finite layer neural networks: Bridging deep architectures and numerical differential equations.

Madry, A., Makelov, A., Schmidt, L., Tsipras, D., & Vladu, A. (2019). Towards deep learning models resistant to adversarial attacks.

Morrill, J., Kidger, P., Yang, L., & Lyons, T. (2021). Neural controlled differential equations for online prediction tasks.

Nielsen, M. (2019). *Neural Networks and Deep Learning*.

Poli, M., Massaroli, S., Park, J., Yamashita, A., Asama, H., & Park, J. (2021). Graph neural ordinary differential equations.

Pontriagin, L., Boltjanskij, G., Boltânskij, V., Collection, K. M. R., Gamkrelidze, V., Trirogoff, K., Neustadt, L., Gamkrelidze, R., Neustadt, W., Mišenko, E., et al. (1962). *The Mathematical Theory of Optimal Processes*. Interscience publishers. Interscience Publishers.
URL https://books.google.com.vn/books?id=ntNSAAAAMAAJ

Rubanova, Y., Chen, R. T. Q., & Duvenaud, D. (2019). Latent odes for irregularly-sampled time series.

Sherstinsky, A. (2020). Fundamentals of recurrent neural network (rnn) and long short-term memory (lstm) network. *Physica D: Nonlinear Phenomena*, *404*, 132306.
URL http://dx.doi.org/10.1016/j.physd.2019.132306

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2017). Attention is all you need.

Xuanqing, L., Tesi, X., Si, S., Qin, C., Sanjiv, K., & Cho-Jui, H. (2019). Neural sde: Stabilizing neural ode networks with stochastic noise.

Yan, H., Du, J., Tan, V. Y. F., & Feng, J. (2021). On robustness of neural ordinary differential equations.

Zhang, A., Lipton, Z. C., Li, M., & Smola, A. J. (2021). Dive into deep learning. *arXiv preprint arXiv:2106.11342*.

Çağatay Yıldız, Heinonen, M., & Lähdesmäki, H. (2019). Ode^{2}vae: Deep generative second order odes with bayesian neural networks.