NeuralODE studying

Ilya Lopatin
Optimization Class Project. MIPT

Introduction

The NeuralODE architecture of was created in paper [1], 2018. This project is devoted to studying of this neural network's architecture and carrying out numerical experiments with it.

The basic mathematical ideas of NeuralODE

We can see that forward propagation in residual neural network looks like Euler's method of numerical solving of ordinary differential equation (1). It turns out that calculating gradient is presented as a solution of the other (*adjoint*) ODE (2):

$$\begin{cases} \frac{dz(t)}{dt} = f(z(t), t, \theta), \\ t \in [t_1; t_2], \\ z(t_1) = x \end{cases} \quad (1) \qquad \begin{cases} \frac{da(t)}{dt} = -a^T(t)\frac{\partial f}{\partial z(t)} \\ a(t_2) = \frac{\partial L}{\partial y} \end{cases} \quad (2)$$

where $f(z(t), t, \theta)$ is representation of neural network, θ is set of trainable parameters and $L = L(y(x))$ is loss function. The euation (3) gives us possibility to calculate gradient $\nabla_\theta L$ and do the procedures of gradient descent.

$$\nabla_\theta L = \int_{t_1}^{t_2} a^T(t)\frac{\partial f}{\partial \theta}(z(t), t, \theta)\, dt. \qquad (3)$$

Below we can see the algorithm of the calculation gradient loss function.

Algorithm 1: Reverse-mode derivative of an ODE initial value problem:

```
Input: dynamics parameters $\theta$, start time $t_0$, stop time $t_1$, final state $\mathbf{z}(t_1)$,
    loss gradient $\partial L/\partial \mathbf{z}(t_1)$
1: $s_0 = [\mathbf{z}(t_1), \frac{\partial L}{\partial \mathbf{z}(t_1)}, \mathbf{0}_{|\theta|}]$        ▷ Define initial augmented state
2: def aug_dynamics($[\mathbf{z}(t), \mathbf{a}(t), \cdot], t, \theta$):        ▷ Define dynamics on augmented state
3:     return $[f(\mathbf{z}(t), t, \theta), -\mathbf{a}(t)^\mathsf{T}\frac{\partial f}{\partial \mathbf{z}}, -\mathbf{a}(t)^\mathsf{T}\frac{\partial f}{\partial \theta}]$        ▷ Compute vector-Jacobian products
4: $[\mathbf{z}(t_0), \frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}]$ = ODESolve($s_0$, aug_dynamics, $t_1, t_0, \theta$)        ▷ Solve reverse-time ODE
return $\frac{\partial L}{\partial \mathbf{z}(t_0)}, \frac{\partial L}{\partial \theta}$        ▷ Return gradients
```

Possible applications

- Replacing Res-block with NeuralODE block for using more powerfully solvers.
- Restoring a hidden dynamics function by given set of observations.
- Continuous normalizing flows, modeling or sampling of unknown probability density function by given points. The most common methods have $O(d^3)$ time asymptomatic when d is dimension of space. The following statement turns out to be useful for reducing time costs to $O(d)$ (see [1]):
  Let $z(t)$ be a finite continuous random variable with probability $p(z(t))$ dependent on time. Let $dz/dt = f(z(t), t)$ be a differential equation describing a continuous-in-time transformation of $z(t)$. Assuming that f is uniformly Lipschitz continuous in z and continuous in t, then the change in $\log$ probability also follows a differential equation (4) :

$$\frac{\partial \log p(z(t))}{\partial t} = -tr\left(\frac{\partial f}{\partial z(t)}\right) \qquad (4)$$

Restoring the hidden dynamics function

If we have some set of given observations or points of hidden distribution density, the adjoint sensitivity method solves an augmented ODE backwards in time. The augmented system contains both the original state and the sensitivity of the loss with respect to the state. If the loss depends directly on the state at multiple observation times, the adjoint state must be updated in the direction of the partial derivative of the loss with respect to each observation, see figure 1.

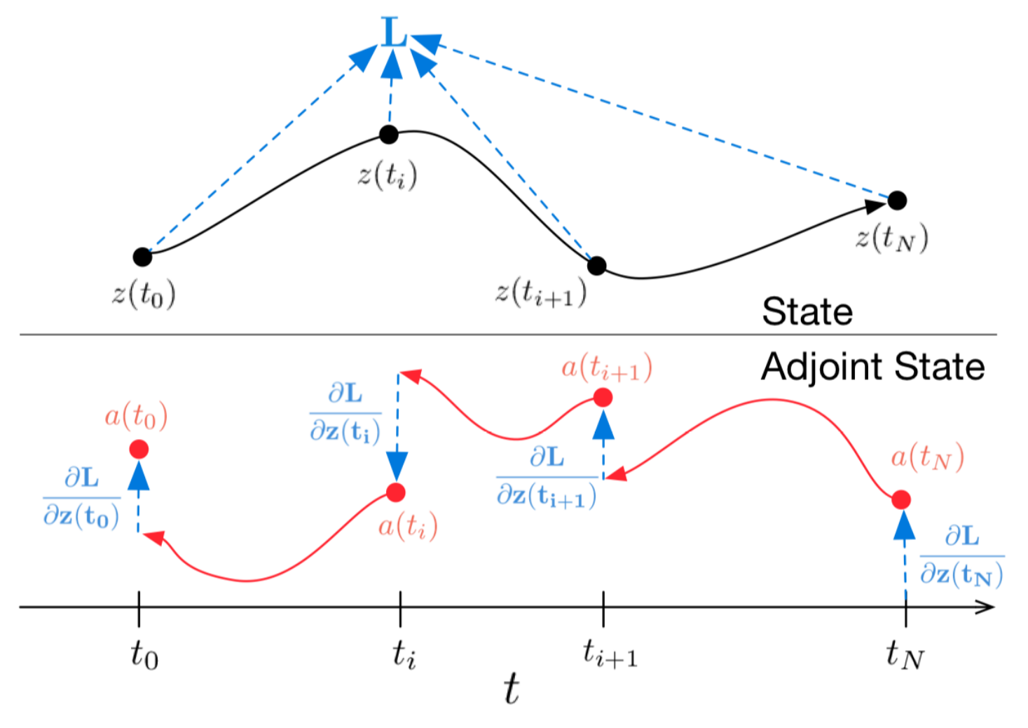

L
$z(t_0)$
$z(t_i)$
$z(t_{i+1})$
$z(t_N)$
State
Adjoint State
$a(t_0)$
$a(t_i)$
$a(t_{i+1})$
$a(t_N)$
$\frac{\partial L}{\partial z(t_0)}$
$\frac{\partial L}{\partial z(t_i)}$
$\frac{\partial L}{\partial z(t_{i+1})}$
$\frac{\partial L}{\partial z(t_N)}$
t_0
t_i
t_{i+1}
t_N
t

Figure 1

Note that during the experiments, the following problem was found. Let the hidden dynamics function conform to the equation $z' = Az$ if the constant matrix A has an eigenvalue with a positive real part, which means that the system is unstable, then the program crashes. Author's comment about it: *This underflow error indicates that the system is too stiff for an explicit method to solve. The step size needs to be extremely small in order to satisfy the desired tolerance. Ricky Chen*

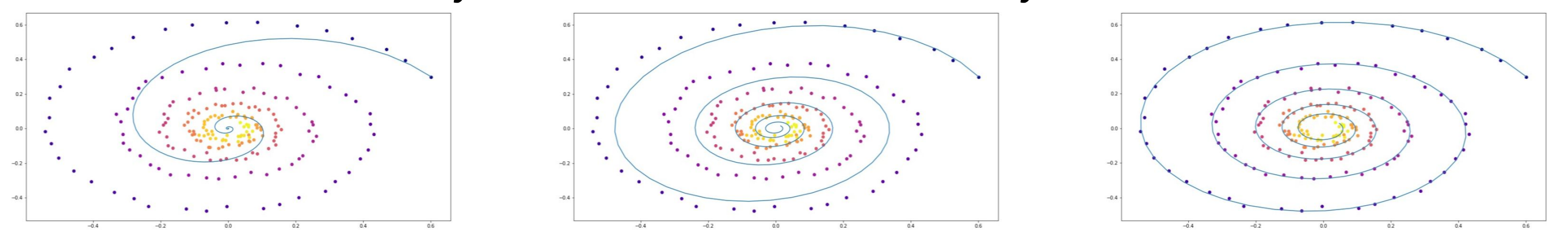


Figure 2: The linear function of dynamics

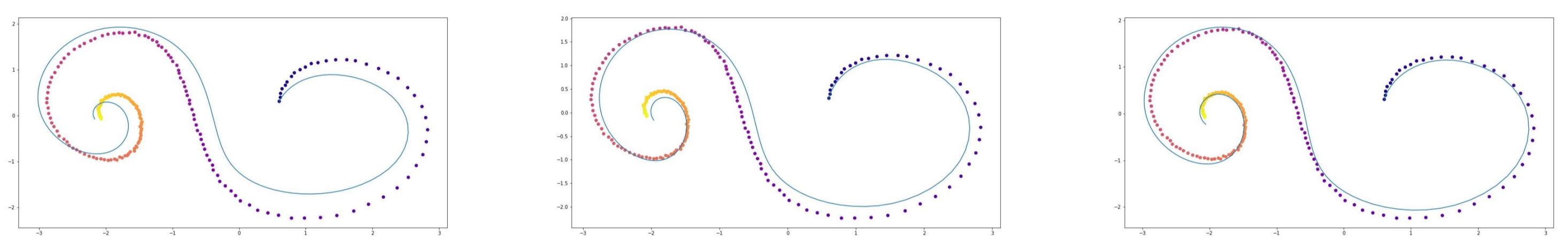


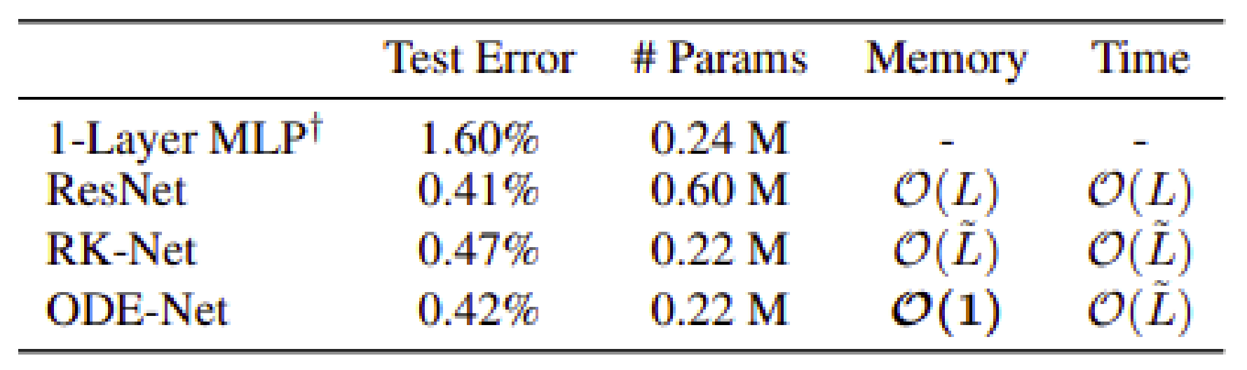
Figure 3: The nonlinear function of dynamics

In the first case the ODE is $z' = Az$ when A is constant matrix, in the second case the ODE is $z = \sigma(<x, x_0>)A(x - x_0) + \sigma(-<x, x_0>)B(x + x_0)$ when σ is sigmoida, A, B are constant matrix and x_0 is constant vector. In [5] you can see gif images of restoring processes.

MNIST test

The first experiments were devoted replay to classic MNIST test of recognizing numbers. In figure 4 we can see result of this experiment from [1]. We get the similar results.

L denotes the number of layers in the ResNet, and $\tilde{L}$ is the number of function evaluations that the ODE solver requests in a single forward pass, which can be interpreted as an implicit number of layers.



	Test Error	# Params	Memory	Time
1-Layer MLP†	1.60%	0.24 M	-	-
ResNet	0.41%	0.60 M	$\mathcal{O}(L)$	$\mathcal{O}(L)$
RK-Net	0.47%	0.22 M	$\mathcal{O}(\tilde{L})$	$\mathcal{O}(\tilde{L})$
ODE-Net	0.42%	0.22 M	$\mathcal{O}(1)$	$\mathcal{O}(\tilde{L})$

Figure 4

Our experiments demonstrate that it takes 4 times longer for NeuralODE to achieve equivalent results with ResNet, but at the same time NeuralODE requires 3 times less trainable parameters and a constant quantity of memory.

Let's note a few more of our observations. Firstly, RK-net is NeuralOde but we don't solve adjoint task (2), but use the values obtained during forward propagation. This reduces the time by about half, but increases the cost of memory up to $O(\tilde{L})$. Secondly, the time cost of training of NeuralODE directly depends on the solution of ODE (1), (2) method used and the value of its step. The experiment clearly shows the obvious linear dependence of time on the number of stages k of the Runge-Kutta method and the smallness h^{-1} of the step, in other words $O(k/h)$. The dependence of accuracy on the method and the step size will be examined in detail later. Negative observation, an increase in the order of the Runge-Kutta method has little effect on the accuracy of recognition of the test sample.

Conditional normalizing flows

[2] contains some examples of working of CNF in case two-dimensional, in [3] CNF method is used for sampling in three-dimensional case, see figures 5, 6.

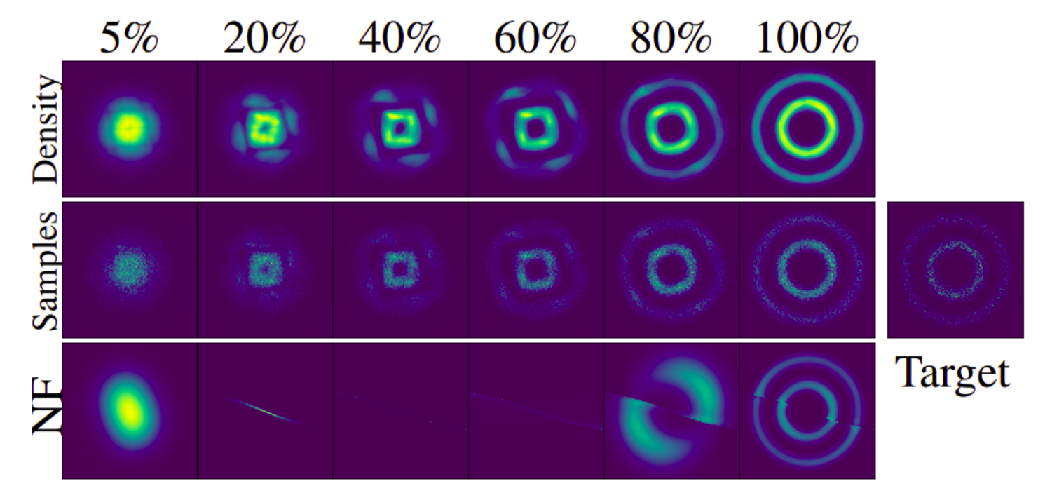


Figure 5

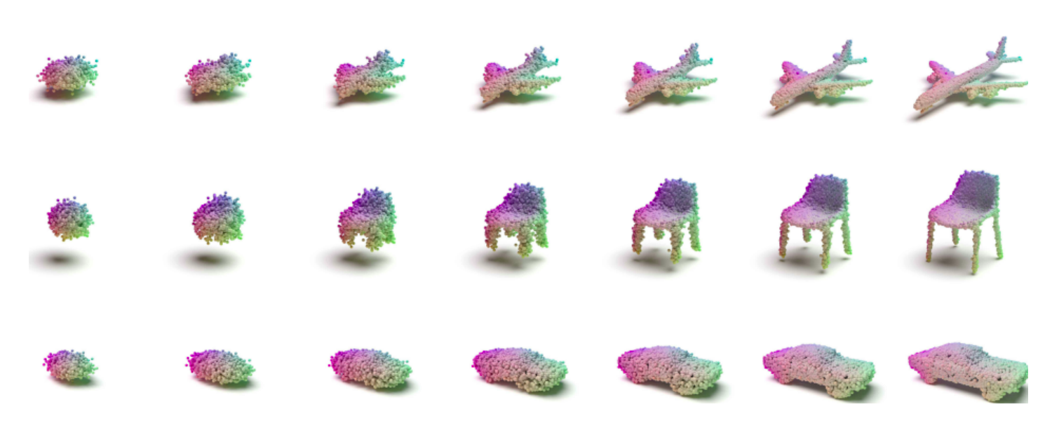


Figure 6

More gifs can be found in [5].

Conclusion

During our experiments, we partially tested some of the advantages and possible applications of NeuralODE. We got positive results on the MNIST test and in the task of restoring hidden dynamics in the case of a stable system, it was mentioning that NeuralODE did a good result even with non-linear dynamics. We produced the tests with CNF and get positive results in 2-dimensional and 3-dimensional cases.

Acknowledgements

This material is based on paper [1]. The source code of the neural network was taken from [2]. The code from [4] turned out very helpful because it contains all code in one python notebook. Paper [3] contains code with applying CNF method in three-dimensional case. I would like to express my special gratitude to Daniil Merkulov for him initiative to make this project, references and advices.

References

[1] Ricky T., Q. Chen, Yulia Rubanova, Jesse Bettencourt. *Neural Ordinary Differential Equations*. University of Toronto, Vector Institute, 2018.

[2] https://github.com/rtqichen/torchdiffeq

[3] Guandao Yang, Xun Huang, Zekun Hao, Ming-Yu Liu, Serge Belongie, Bharath Hariharan. *PointFlow: 3D Point Cloud Generation with Continuous Normalizing Flows*. Preprint arxiv.org, 2019.

[4] https://github.com/msurtsukov/neural-ode

[5] https://github.com/Ilya-Lopatin/NeuralODE_experiments - my repository.