Neural Ordinary Differential Equations

Runwei Zhou, ECE, PhD

Abstract

Neural ODEs (Ordinary Differential Equation) is a new type of neural network that learns the evolution of continuous dynamical systems. Instead of specifying the layers of the neural network upfront, a Neural ODEs can be trained to approximate the continuous time evolution of a system by solving an initial value problem through back-propagation. This approach has several advantages over traditional neural networks, including the ability to handle variable-length inputs and outputs, improved memory efficiency, and more accurate time-dependent predictions. In this review, we re-implemented the Neural ODEs proposed by the authors and conducted experiments to validate and test our re-built solver.

1. Introduction

A neural network is a class of models used for solving sorting and fitting tasks in machine learning. A significant problem is the complex dynamical systems described by differential equations(Chen et al., 2018). Though conventional neural network setup has shown great success in solving such continuous problems, one can employ the neural ordinary differential equations network(ODE net) as follows:

$$\frac{d\mathbf{h}(t)}{dt} = f(\mathbf{h}(t), \theta) \tag{1}$$

where $\mathbf{h}(t)$ is the conventional hidden layer as the function of t, and θ is parameters defined by the hidden layer function. According to the original paper, the continuous neural network design has the following benefits:

Memory efficiency The authors proposed a novel backward propagation method by calling the reverse model derivative function with a pre-defined adjoint state and corresponding augmented dynamics. We will show the detailed derivation in section 3.

Adaptive computation One of the benefits of the Neural ODEs net is that the complexity of the learning process can be controlled by varying the implemented ODE solver. From the Euler method to the high-order Runge–Kutta method, the learning dynamics can be error-controlled(Gupta et al., 1985).

Parameter efficiency The last three benefits mentioned in the paper are all related to the continuous flow forward propagation nature in the Neural ODEs net. As shown in section 4, the proposed neural network can learn the time-dependent dynamics with simpler hidden layers which indicates that it helps to mitigate the learning parameters as well as the learning capability for time-dependent models.

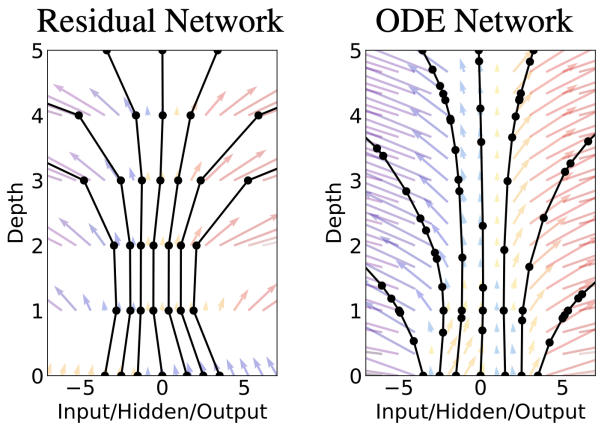


Figure 1. Difference between data flow in the residual network and ODE network. The ODE network is adaptive to the real dynamics thanks to the call of the ODE solver while the residual net takes constant time steps.

In this report, we first do a literature review to summarize the related works in section 2. Then we introduce the mathematical background proposed in the original paper in my own language in section 3. Numerical experiments are shown in section 3 which proves the advantages of the method above. In section 5, a conclusion is made.

2. Related work

The Neural ODEs is inspired by the residual network (ResNets)(Tai et al., 2017; Zagoruyko & Komodakis, 2016; He et al., 2016; 2015) which connects the hidden layers with

residual-like expressions:

$$\mathbf{h}_{t+1} = \mathbf{h}_t + f(\mathbf{h}_t, \theta) \tag{2}$$

By introducing a novel connection framework, the deeper layers are now more consistent than the initial ones which allow a larger scale hidden layer setup, which helps the conventional layer-by-layer neural nets to solve gradient vanishing risk when learning from a great number of data with a deep network design. The ResNets is a special class of Neural ODEs net by taking numerical discretization in Euler format. Such continuous implementation has shown promising results in a variety of applications, including image classification(Yan et al., 2019; Dandekar et al., 2020; Paoletti et al., 2019), generative modeling(Dockhorn, 2019; Kidger, 2022), and physics simulation(Tuor et al., 2020; Lee & Parish, 2021). The basic idea of Neural ODEs is to use a well-developed ODE solver to learn the dynamics behind the neurons and treat parameterized hidden layers as a black box function.

3. Method

As a competitive tool for simulating complex dynamical systems, any neural network is a universal function generator for optimizing pre-defined loss with input state and the final state as $\mathbf{z}(t_0)$ and $\mathbf{z}(t_n)$:

$$Loss = L(\mathbf{z}(t_0), \mathbf{z}(t_n), \theta) \tag{3}$$

where final state $\mathbf{z}(t_n)$ is described by a time integral:

$$\mathbf{z}(t_n) = \mathbf{z}(t_0) + \int_{t_0}^{t_n} f(\mathbf{z}(t), \theta) dt \tag{4}$$

Eq. 4 can be achieved easily by any ordinary differential equation algorithm. Here $f(\mathbf{z}(t), \theta)$ is the conventional hidden neural layers with parameters θ to optimize.

Besides, we need gradients of parameters for backward propagation, which are $\partial L/\partial \mathbf{z}(t_0)$, $\partial L/\partial t_0$, $\partial L/\partial t_1$, $\partial L/\partial \theta$. With the help of auto-gradient, one can achieve $\partial L/\partial \mathbf{z}(t_n)$ as the output in each learning epoch. We can define adjoint state $\mathbf{a}(t) = -\partial L/\partial \mathbf{z}(t)$ and get:

$$\begin{aligned} \mathbf{a}(t) &= -\frac{\partial L}{\partial \mathbf{z}(t)} \\ &= -\frac{\partial L}{\partial \mathbf{z}(t+\delta t)} \frac{\partial \mathbf{z}(t+\delta t)}{\partial \mathbf{z}(t)} \\ &= \mathbf{a}(t+\delta t) \frac{\partial \mathbf{z}(t+\delta t)}{\partial \mathbf{z}(t)} \end{aligned} \tag{5}$$

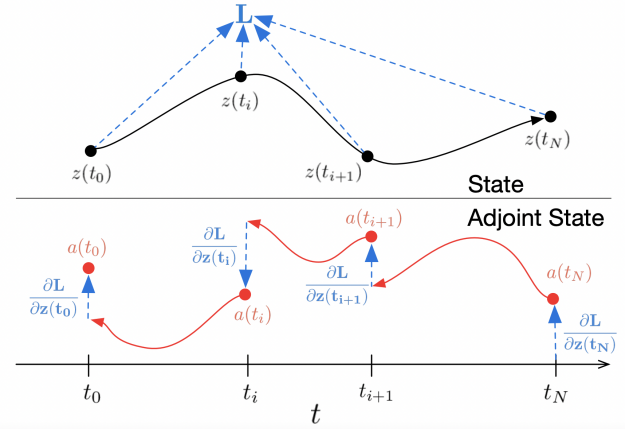


Figure 2. The author defines an adjoint state for backward propagation. By calling the same ODE solver for forward propagation, we can calculate the necessary gradients.

Take time derivative of adjoint state $\mathbf{a}(t)$ and get:

$$\begin{aligned} \frac{d\mathbf{a}(t)}{dt} &= \lim_{\delta t \to 0} \frac{\mathbf{a}(t+\delta t) - \mathbf{a}(t)}{\delta t} \\ &= \lim_{\delta t \to 0} \frac{\mathbf{a}(t+\delta t) - \mathbf{a}(t+\delta t)\frac{\partial \mathbf{z}(t+\delta t)}{\partial \mathbf{z}(t)}}{\delta t} \\ &= \lim_{\delta t \to 0} \frac{\mathbf{a}(t+\delta t) - \mathbf{a}(t+\delta t)\frac{\partial [\mathbf{z}(t) + \delta t \frac{\partial \mathbf{z}(t)}{\partial t} + O(\delta t^2)]}{\partial \mathbf{z}(t)}}{\delta t} \\ &= \lim_{\delta t \to 0} \frac{\mathbf{a}(t+\delta t) - \mathbf{a}(t+\delta t)(\mathbf{I} + \delta t \frac{\partial f(\mathbf{z}, \theta)}{\partial \mathbf{z}(t)})}{\delta t} \\ &= \lim_{\delta t \to 0} -\mathbf{a}(t+\delta t)\frac{\partial f(\mathbf{z}, \theta)}{\partial \mathbf{z}(t)} \\ &= -\mathbf{a}(t)\frac{\partial f(\mathbf{z}, \theta)}{\partial \mathbf{z}(t)} \end{aligned} \tag{6}$$

Where first-order Taylor series and the ODE relation in Eq. 1 are used for approximation. Noted that the error can be controlled as long as the time step is small and the target function is compact in the vicinity of the expansion point $\mathbf{z}(t)$.

Eq. 6 is a new ordinary equation from which necessary derivatives at t_0 can be achieved:

$$\frac{\partial L}{\partial \mathbf{z}(t_0)} = -\mathbf{a}(t_0) = \int_{t_n}^{t_0} \mathbf{a}(t)\frac{\partial f(\mathbf{z}, \theta)}{\partial \mathbf{z}(t)} dt \tag{7}$$

Similarly, we can derive other gradients in such integrated form:

$$\frac{\partial L}{\partial t} = \int_{t_n}^{t_0} \mathbf{a}(t)\frac{\partial f(\mathbf{z}, \theta)}{\partial t} dt \tag{8}$$

$$\frac{\partial L}{\partial \theta} = \int_{t_n}^{t_0} \mathbf{a}(t)\frac{\partial f(\mathbf{z}, \theta)}{\partial \theta} dt \tag{9}$$

Algorithm 1 Neural ODEs net

function ODESOLVER($\mathbf{z}(t_0), t_0, t_1, \theta, f$) ▷ ODE solver with initial state $\mathbf{z}(t_0), t_0, t_1, \theta$ and function handle f
  $d\mathbf{z}(t)/dt = f(\mathbf{z}(t), f)$ ▷ Ordinary differential equation
  return $\mathbf{z}(t_1)$ ▷ Return final state $\mathbf{z}(t_1)$
end function
function $f_h(t_0, t_1, \theta)$
  Hidden layers ▷ Define proper neural nets model
  return $\mathbf{z}(t_1)$
end function
function FORWARD($\mathbf{z}(t_0), t_0, t_1, \theta, f_h$) ▷ Forward propagation with predefined hidden layers f_h
  $\mathbf{z}(t_1)$=ODEsolver($\mathbf{z}(t_0), t_0, t_1, \theta, f_h$) ▷ Forward propagation with single call of ODEsolver
  return $\mathbf{z}(t_1)$ ▷ Return final state
end function
function BACKWARD($\partial L/\partial \mathbf{z}(t_1), t_0, t_1, \theta$) ▷ Backward propagation with adjoint state
  $\mathbf{z}_0$ = [$\partial f/\partial \mathbf{z}(t_1)$, $\mathbf{0}$, $\partial f/\partial t_1$, $\partial f/\partial \theta$] ▷ Initial state for backward propagation
  [$\partial L/\partial \mathbf{z}(t_0)$, $\partial L/\partial t_0$, $\partial L/\partial t_1$, $\partial L/\partial \theta$] = ODEsolver($\mathbf{z}_0$,$t_1$, t_0, θ, Dynamic Function) ▷ Backward propagation with ODEsolver. Dynamic Function is defined in the previous context
  return $\partial L/\partial \mathbf{z}(t_0)$, $\partial L/\partial t_0$, $\partial L/\partial t_1$, $\partial L/\partial \theta$ ▷ Return necessary gradients
end function

Combining Eq. 7, Eq. 8, Eq. 9, we can get the gradient with the "augmented state dynamics" algorithm in the original paper. In summary, we can describe the structure of the Neural ODEs net in the algorithm. 1 where we change the conventional forward and backward propagation method.

4. Experiment

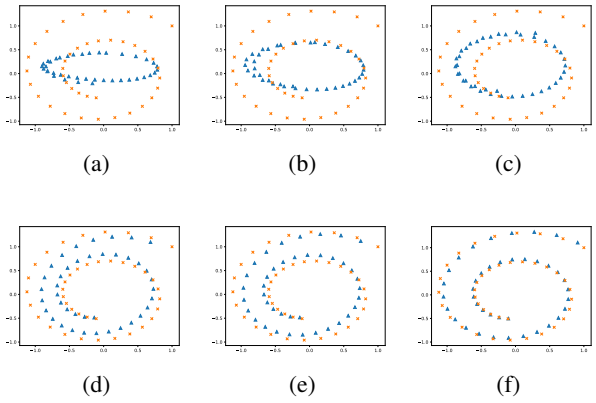
To evident that the proposed neural network setup not only preserved the good aspect of conventional neural networks but also is indeed good at modeling continuous dynamics, we conducted three numerical experiments. First, we let our neural net learn spiral dynamics to validate the idea and our re-implementation. Then to test the claim that ResNets is a special class of Neural ODEs, we test the performance of our code in solving partial differential equation problems by combining the continuous net architecture with physics informed neural network(PINN).

4.1. Learning spiral dynamics

We first validate our solver by learning ordinary equation-based spiral dynamics. The learning data is generated with the following ODE:

$$\frac{d\mathbf{z}}{dt} = \begin{bmatrix} -0.1 & -1.0 \\ 1.0 & -0.1 \end{bmatrix} \mathbf{z} \tag{10}$$

It can be easily solved by calling any ODE solver. Here we employ `odeint(func, y0, t)` function in the author's original source code, which supports the tensor data type. The initial condition is set to be $\mathbf{z}(0) = [1;\ 1]$. We set the hidden layer function to be a single 2 by 2 linear projection and wrap up the dynamics to the re-implemented neural network. The full experiments are conducted by Adam optimizer with a learning rate equal to 0.1 and the loss function is defined as mean squared loss to each testing point. We can achieve the learning patterns in fig. 4. The orange dots indicate the observation points while the blue dots indicate the predicted point.



(a) (b) (c) (d) (e) (f)

Figure 3. Learning pattern in different epochs: (a) epoch=5, (b) epoch=6, (c)epoch=7, (d) epoch=10, (d) epoch=11, (d) epoch=15

Good agreement has been achieved within 15 learning epochs. Also, we observed that the learning process is continuous and data-driven. That's because the Neural ODEs gives a global picture for our model to catch while the point-by-point learning could be troublesome for conventional neural network setups.

In addition, we try to modify the spiral equation to:

$$\frac{d\mathbf{z}}{dt} = \begin{bmatrix} -1.0 & -1.0 \\ 1.0 & -0.5 \end{bmatrix} \mathbf{z} \tag{11}$$

We used a more complex hidden layer function with a lower learning rate equal to 0.001. Noise is added to the learning data with `pytorch.randn_like` handle.

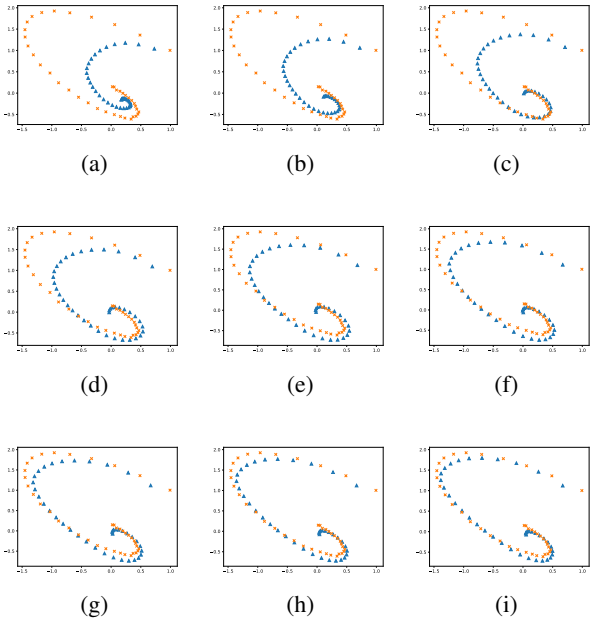


Figure 4. Learning pattern in different epochs. (a)-(i) learning epoch from 1 to 9.

Although the least required unknowns is still 4(four matrix elements in the matrix). We can not get the net converged with the simper hidden layer that we used previously. On one hand, the distribution of the learning data is different. The previous dataset is more coarse and the points are evenly distributed. However, the new example has a more violent dynamic. On the other hand, the renewed spiral dynamic is ill-conditioned so a single call of ODE solver might import more numerical error.

The success of learning the spiral dynamics validate our implemented solver. We proved that it can be used in learning time-variant trajectories and the learning process is indeed continuous like what is reported.

4.2. Combining physics-informed learning for solving partial differential equations

To test the performance of the Neural ODEs, we try to combine a physics-informed neural network setup with our re-implemented solver. Particularly, we hope our Neural ODEs to solve partial differential equations. In this test example, we test the Poisson equation with the Dirichlet boundary condition

$$\Delta\Phi = \rho \tag{12}$$

The Poisson equation is a partial differential equation that describes the behavior of the electric potential or gravitational potential in a region where a charge or mass density is present. It is named after the French mathematician Siméon Denis Poisson, who first studied its properties in the 19th century. Where Δ is the laplacian operator and ρ is known source function. The Poisson equation is a fundamental equation in many areas of physics, including electromagnetism, gravitation, and fluid dynamics. It has important applications in fields such as electrical engineering, solid-state physics, and astrophysics. One can numerically solve the Poisson equation by finite element method(FEM) or using a machine learning-based solver like the physics-informed neural network.

A Physics Informed Neural Network (PINN) is a type of artificial neural network that incorporates physical principles and laws into its architecture to solve partial differential equations (PDEs) and other complex problems in physics and engineering. Instead of data-driven loss, the PINN employs PDE loss and boundary loss as its loss function. For the Poisson equation, the loss function is:

$$L = L_{PDE} + L_{bound} = \|\Delta\Phi - \rho\| + \|\Delta\Phi + \Phi\| \tag{13}$$

Here we define source ρ as:

$$\rho = 2\pi \sin(x) \sin(y) \tag{14}$$

for 2-D Poisson equation. Standard solution can be achieved via finite difference method as:

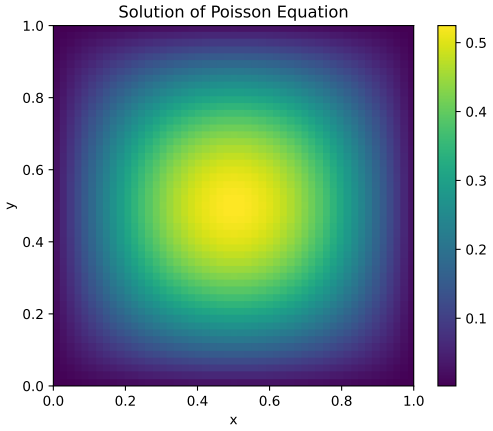


Figure 5. Final State

Where the solution domain is defined in $[0, 1] \times [0, 1]$. We then try to solve the same problem with our neural ordinary differential equation solver and get Fig. 6.

After 60 steps setting the learning rate to 0.1, we got to a relatively stable stage and have the boundary Loss be 0.102908, PDE Loss be 0.076783. The convergence plot

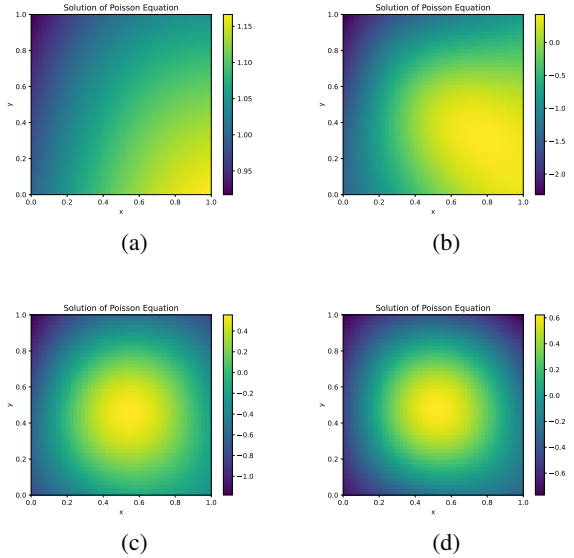


Figure 6. Learning pattern in different epochs: (a) epoch=1, (b) epoch=20, (c)epoch=40, (d) epoch=60

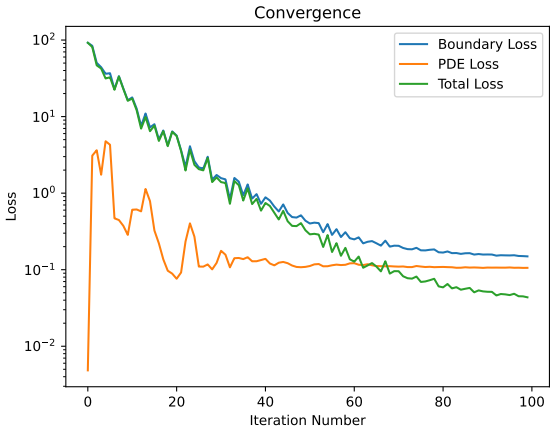


Figure 7. Convergence plot of solving the Poisson equation

is shown in fig. 7. The boundary loss takes advantage at first and gets converged together with the PDE loss. In this example, we validate our solver again. In addition, we tested its performance by solving real-world partial differential equations. Since the Neural ODEs is an extended model for ResNets, it should be able to handle most of the machine-learning tasks and not be restricted by time-variant problems. This example becomes good evidence for the capability of the proposed Neural ODEs architecture.

5. Conclusion

In conclusion, this paper has presented a comprehensive review of the mathematical background and implementation of Neural Ordinary Differential Equations (Neural ODEs). Two different experiments were conducted to demonstrate the effectiveness and flexibility of this approach.

The first experiment involved a simple classification task using spiral dynamics, which demonstrated the ability of Neural ODEs to learn and represent complex functions with fewer parameters compared to traditional neural networks. The second experiment focused on a more complex task involving a differential equation from electrodynamics, where Neural ODEs were used to approximate the solution of the equation with high accuracy and efficiency.

Overall, the results of both experiments suggest that Neural ODEs have great potential for solving a wide range of problems in physics and engineering, particularly those involving continuous-time dynamics or differential equations. Moreover, the flexibility and scalability of Neural ODEs allow for the development of new models that can capture more complex and realistic dynamics in real-world applications.

During the exploration, we find some limitations to this Neural ODEs implementation beyond those mentioned in the original context:

1. **Computational Limitations**: The speed of training Neural ODEs is strongly limited by the complexity of hidden layers. As the number of data points or the complexity of the system increases, the computational cost of training the neural network can become prohibitive.

2. **Interpretability**: When using Neural ODEs to solve different tasks, one may find it hard to translate certain problems to an ODE based-net. The understanding cost is considerable compared to other neural net architecture. It may cause obstacles in designing proper neural ODE functions for different objectives.

3. **Tuning Hyperparameters**: Tuning hyperparameters often becomes very difficult when working with Neural ODEs. One often needs to be very careful when trying to design the structure of the hidden function which is unpredictable with the ODE-facilitated forward propagation. This can be time-consuming and requires expertise to achieve the best net structure.

Future research could explore the application of Neural ODEs in other domains, such as biology or finance, and further investigate the interpretability and robustness of these models. Additionally, improvements to the training process, such as regularization techniques or more advanced optimization algorithms, could help to further enhance the performance of Neural ODEs.

References

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

Dandekar, R., Chung, K., Dixit, V., Tarek, M., Garcia-Valadez, A., Vemula, K. V., and Rackauckas, C. Bayesian neural ordinary differential equations. *arXiv preprint arXiv:2012.07244*, 2020.

Dockhorn, T. Generative modeling with neural ordinary differential equations. Master's thesis, University of Waterloo, 2019.

Gupta, G. K., Sacks-Davis, R., and Tescher, P. E. A review of recent developments in solving odes. *ACM Computing Surveys (CSUR)*, 17(1):5–47, 1985.

He, K., Zhang, X., Ren, S., and Sun, J. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pp. 1026–1034, 2015.

He, K., Zhang, X., Ren, S., and Sun, J. Identity mappings in deep residual networks. In *Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part IV 14*, pp. 630–645. Springer, 2016.

Kidger, P. On neural differential equations. *arXiv preprint arXiv:2202.02435*, 2022.

Lee, K. and Parish, E. J. Parameterized neural ordinary differential equations: Applications to computational physics problems. *Proceedings of the Royal Society A*, 477(2253):20210162, 2021.

Paoletti, M. E., Haut, J. M., Plaza, J., and Plaza, A. Neural ordinary differential equations for hyperspectral image classification. *IEEE Transactions on Geoscience and Remote Sensing*, 58(3):1718–1734, 2019.

Tai, Y., Yang, J., and Liu, X. Image super-resolution via deep recursive residual network. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 3147–3155, 2017.

Tuor, A., Drgona, J., and Vrabie, D. Constrained neural ordinary differential equations with stability guarantees. *arXiv preprint arXiv:2004.10883*, 2020.

Yan, H., Du, J., Tan, V. Y., and Feng, J. On robustness of neural ordinary differential equations. *arXiv preprint arXiv:1910.05513*, 2019.

Zagoruyko, S. and Komodakis, N. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.