Neural Network Solver for Ordinary Differential Equations

Jiawei Zhuang, Xu Si, Rui Zhao, Chin Hui Chew

December 2017

Contents

1 Introduction **2**

2 Method **3**
2.1 Fitting functions by artificial neural networks . . . 3
2.2 Solving one ODE by artificial neural networks . . . 4
2.3 Solving ODE systems by artificial neural networks . . . 5

3 Results and Discussions **5**
3.1 Simple ODEs . . . 5
3.2 Unstable, periodic ODEs . . . 8
3.2.1 Lotka-Volterra 2-species model . . . 8
3.2.2 Lotka-Volterra 3-species Model . . . 11
3.2.3 Van der Pol oscillator . . . 13
3.3 Unstable, chaotic ODEs . . . 16
3.3.1 Lorentz attractor . . . 16
3.3.2 Rossler attractor . . . 19
3.4 Stiff ODE system . . . 22
3.5 Extension: Flexible initial conditions . . . 23

4 Conclusions and Future Work **25**

5 Code Availability **25**

1 Introduction

Ordinary differential equations (ODEs) are generally solved by finite-difference methods, from the simplest forward Euler scheme to higher-order schemes like the Runge-Kutta methods. Numerical solutions obtained by those schemes are typically stored in a discretized form, i.e. in an array of floating point numbers. However, these types of solutions have some limitations: 1) storing these discretized approximations may take up huge amount of storage when high-resolution results are required. 2) every step of finite-difference methods counts on the approximate solution from the previous step, accumulating approximation error.

To tackle with the limitations, artificial neural network(ANN) was proposed to obtain ODE solutions in a closed continuous analytical form [1]. Around the same time, it was established that an ANN with only one hidden layer and a nonlinear activation function was capable of approximating any function, when sufficiently many hidden units are available [2]. Since then, an increasing number of papers have dived into using ANN to solve general initial value problems (IVP) [3], boundary value problems (BVP) [4][5] [6] and partial differential equations (PDE) [7] [8] [9] [10].

In this project, we use ANNs (Fig. 1) to obtain solutions of ODE initial value problems in a closed analytical form. The solutions are stored as neural network parameters, which requires much less memory than storing the solution as a discretized array. Also, because the solution is analytically differentiable, it can be superior in some applications like sensitivity analysis.

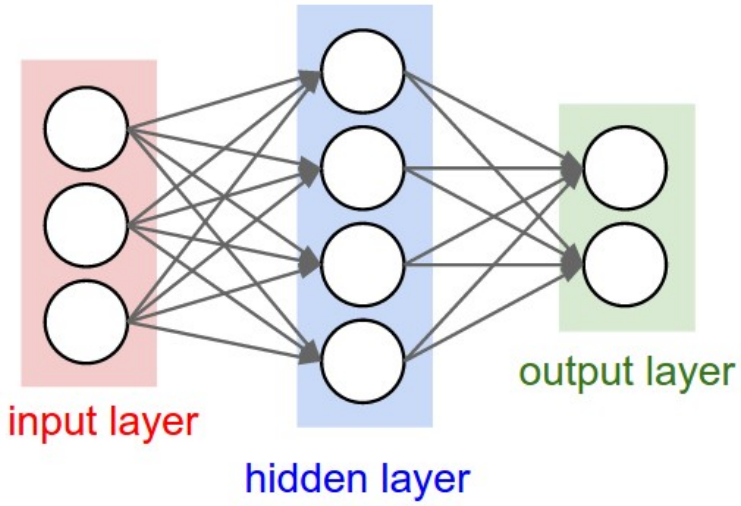


Figure 1: Artificial neural network with one hidden layer

Our main reference in this project is the paper by Lagaris et el. [11], which explored a general method for solving ODEs using feed-forward neural network. The paper focused on the deviation between the predicted values versus the analytical solutions. Another paper by Baymani et el. [3] applied neural network to solve Stokes problem and found that the approximation of the new method results in higher accuracy and requires fewer number of parameters. Chiramonte et al. [12] similarly applied neural network to solve ODEs and found that introducing a regularization term could be beneficial in reducing errors and convergence time.

This report is organized as follows. Section 2 covers the mathematical derivations. In Section 3, we apply the ANN method to a wide range of ODE initial value problems. The problems fall into four categories: 1) simple ODE, 2) unstable and periodic ODEs, 3) unstable and chaotic ODEs and 4) stiff ODE system. We evaluate the performance using average root-mean-square error (RMSE) and histogram of Neural Network (NN) solver performance over 100 fittings. At the end of Section 3, we also extend the original theory to use a single ANN to solve ODEs with arbitrary initial conditions. In Section 4, we list the key findings and future work to deal with challenges and limitations.

2 Method

2.1 Fitting functions by artificial neural networks

The universal approximation theorem [2] states that any continuous function can be approximated by a feed-forward neural network with a single hidden layer. This ANN can be written in a matrix multiplication form:

$$N(x;w) = W_2\sigma(W_1x + b_1) + b_2 \tag{1}$$

where W_1 and W_2 are weight matrices and b_1 and b_2 are bias terms. σ is a nonlinear activation function such as $tanh$. We use w to represent all parameters [W_1, W_2, b_1, b_2]. To fit a scalar function $y(x)$, the neural network takes a scalar input x and returns a scalar output $N(x)$. In this case, W_1 and W_2 degrade to row and column vectors.

The optimal parameters w can be found by minimizing the loss function

$$L(w) = \int_a^b [y(x) - N(x;w)]^2 dx \tag{2}$$

In practice, the integral is approximated by a summation

$$L(w) = \sum_i [y(x_i) - N(x_i;w)]^2 \tag{3}$$

where $\{x_i\}$ is a set of training points covering the domain [a, b].

If the loss is small enough, then the ANN can be considered as a good approximation to the original function over the domain [a, b]:

$$N(x;w) \approx y(x) \tag{4}$$

2.2 Solving one ODE by artificial neural networks

Now we consider constructing an ANN that can approximate the solution to the first-order ODE:

$$y'(t) = F(y(t), t), \;\; y(t_0) = y_0 \tag{5}$$

If we use a standard neural network

$$N(t; w) = W_2\sigma(W_1 t + b_1) + b_2 \tag{6}$$

It will not satisfy the initial condition, i.e. typically $N(t_0; w) \neq y_0$. But we can force the initial condition by rewriting the ANN solution as

$$\hat{y}(t; w) = y_0 + (t - t_0)N(t; w) \tag{7}$$

For any parameters w, there will always be $\hat{y}(t_0; w) = y_0$. We further require this ANN solution $\hat{y}(t; w)$ to satisfy the ODE:

$$\hat{y}'(t; w) \approx F(\hat{y}(t; w), t) \tag{8}$$

Note that the derivative $\hat{y}'(t; w)$ can be derived analytically without any finite-difference approximation

$$\hat{y}'(t; w) = \frac{\partial[y_0 + (t - t_0)N(t; w)]}{\partial t} = \frac{\partial(t - t_0)}{\partial t}N(t; w) + (t - t_0)\frac{\partial N(t; w)}{\partial t} \tag{9}$$

The optimal parameters can be found by minimizing the cost function

$$L(w) = \int_{t_0}^{t_1} [\hat{y}'(t; w) - F(\hat{y}(t; w), t)]^2 dt \tag{10}$$

Or in practice,

$$L(w) \approx \sum_i [\hat{y}'(t_i; w) - F(\hat{y}(t_i; w), t_i)]^2 \tag{11}$$

where $\{t_i\}$ is a set of training points covering the domain $[t_0, t_1]$.

If the loss is small enough, then the ANN solution should be able to approximate the true ODE solution over the domain $[t_0, t_1]$:

$$\hat{y}(t; w) \approx y(t) \tag{12}$$

Here the initial condition y_0 is fixed during the ANN training, which means we need to re-train the ANN for a new initial condition. At the end of Section 3 we also extend this theory so that a single ANN can be trained for multiple initial conditions.

2.3 Solving ODE systems by artificial neural networks

The above ANN method can be directly generalize to a system of ODEs. For simplicity, consider two ODEs

$$y'(t) = F_1(y(t), z(t), t), \;\; y(t_0) = y_0 \tag{13}$$
$$z'(t) = F_2(y(t), z(t), t), \;\; z(t_0) = z_0 \tag{14}$$

We can use two separate ANNs for two variables y and z

$$\hat{y}(t; w) = y_0 + (t - t_0)N_1(t; w_1) \tag{15}$$
$$\hat{z}(t; w) = z_0 + (t - t_0)N_2(t; w_2) \tag{16}$$

Then loss function is the sum of losses for two ODEs

$$L(w_1, w_2) \approx \sum_i \left[[\hat{y}'(t_i) - F_1(\hat{y}(t_i), \hat{z}(t_i), t_i)]^2 + [\hat{z}'(t_i) - F_2(\hat{y}(t_i), \hat{z}(t_i), t_i)]^2 \right] \tag{17}$$

3 Results and Discussions

In this section, we will apply the ANN method to four common types of ODE problems: 1) simple ODE, 2) unstable and periodic ODEs, 3) unstable and chaotic ODEs and 4) stiff ODE system and then compare Neural Network results with traditional finite difference method which we treat as ground truth.

To test the stability of our ANN method, we retrain the ANN 100 times and plot the histogram of its performance over 100 fittings. We have found that in most cases, ANN method can approximate the ground truth well, whereas sometimes we need to train the network multiple times to get ideal results.

In all the examples, we find the BFGS optimization method beats all other optimizers in Scipy, so we use it as the default method.

3.1 Simple ODEs

Overview

Consider a first-order ODE system [11]:

$$\frac{dx}{dt} = \cos t + x^2 + y - (1 + t^2 + \sin^2 t) \tag{18}$$

$$\frac{dy}{dt} = 2t - (1 + t^2) \sin t + xy \tag{19}$$

where $t \in [0, 3]$.

Mathematics

The analytical solutions are:

$$x = \sin t \tag{20}$$

$$y = 1 + x^2 \tag{21}$$

Ideally, our NN solver should attain the same solution as the exact analytical solutions within the time range specified.

Results

The solutions using the Neural Net solver could be found in Fig. 2. The network is trained using a grid of 20 equidistant points in $t = [0, 3]$ and 50 hidden units in the hidden layer.

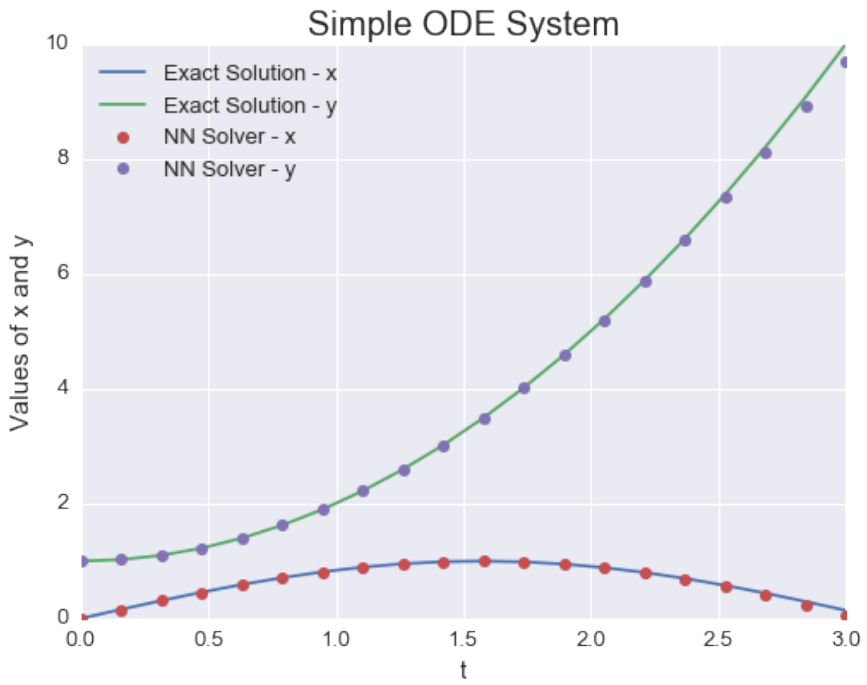


Figure 2: Comparing solutions found using neural net solver versus exact analytical solutions. Average RMSE = 0.0524.

We could see almost an exact fit with the analytical solution. In terms of training convergence, we could see that the neural net rapidly converges to a solution with log loss value of order 10^{-5} at 200 iterations from Fig. 3. BFGS is selected as the optimization method because it results in the lowest loss.

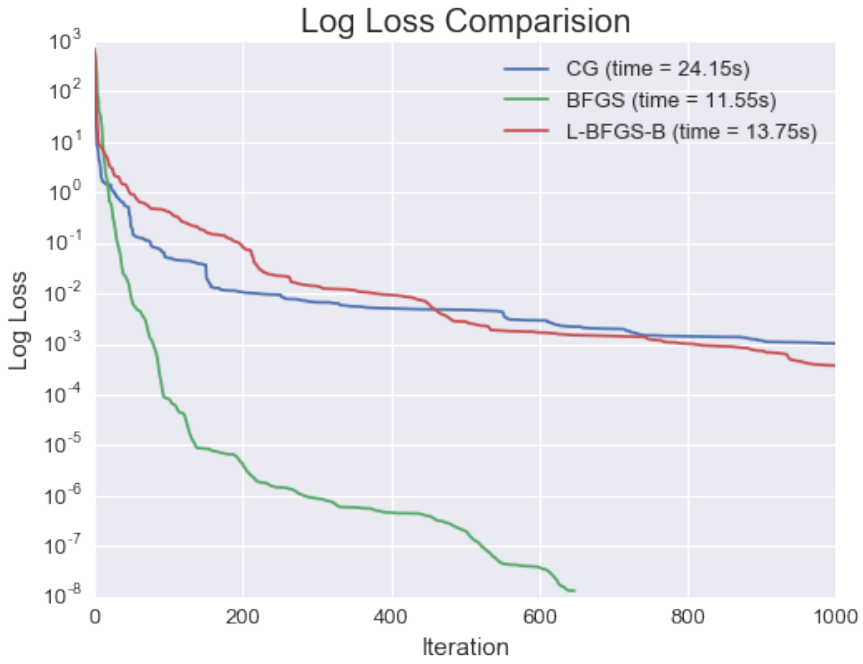


Figure 3: Comparison across different optimization methods for simple ODE system.

Discussions

To evaluate the ANN solution, we propose taking an average of the RMSE:

$$RMSE_{overall} = \frac{1}{2}[RMSE_x + RMSE_y] \tag{22}$$

The fitting attained as shown in Fig. 2 has $RMSE_{overall} = 0.0524$.

Because of different initializations of the weights, the fit could vary during each training, reaching different local minimum. To test the reproducibility of ANN training, we repeat the same fitting process 100 times. The average RMSE histogram distribution for 100 fittings is shown in Fig. 4. 48% of the fittings are very close to the exact analytical solution, with RMSE of up to 0.0378 (our plot in Fig. 2 has RMSE = 0.0524). But about half of the initializations result in a relatively large loss around 3.5 and cannot fit the solution very well.

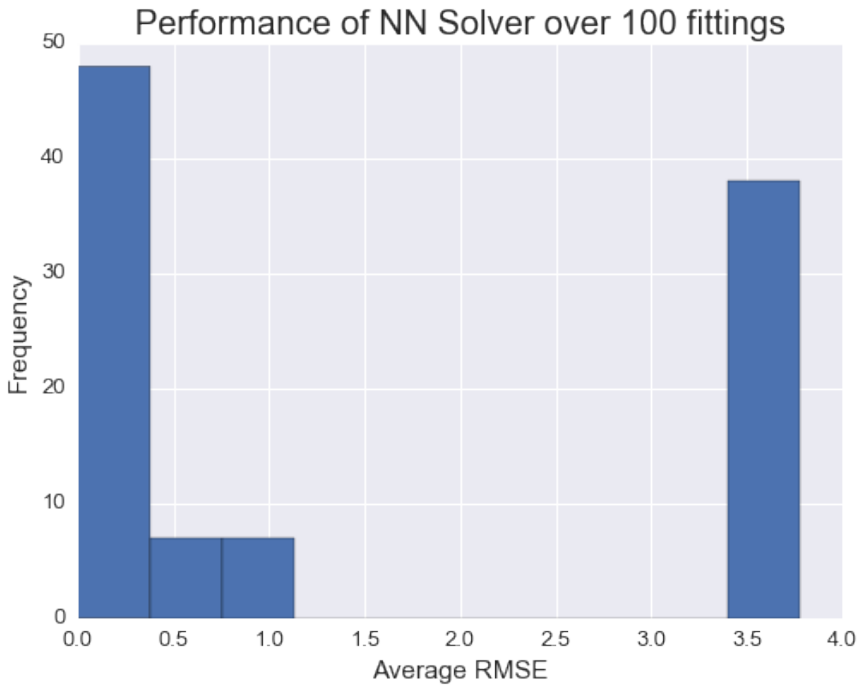


Figure 4: Average RMSE Distribution for 100 Repeated Fittings for Simple ODE System.

Please refer to **AM205_final/notebook/CHC/SimpleODE.ipynb** for the code and visualizations of this section.

3.2 Unstable, periodic ODEs

3.2.1 Lotka-Volterra 2-species model

Overview

The Lotka-Volterra prey-predator equations are a pair of first-order, non-linear differential equations commonly used to describe the interactions between two species, the prey and predator.

Mathematics

The populations of the prey and predator can be described by the following equations:

$$\frac{dx}{dt} = ax - bxy \tag{23}$$

$$\frac{dy}{dt} = -cy + dxy \tag{24}$$

where

- x is the population density of prey
- y is the population density predator
- t represents time
- $a, b, c, d > 0$

Eq. 23 indicates that in the absence of a predator ($y = 0$), the prey would grow at a constant rate a, with the assumption that the prey have an unlimited food supply. The rate of predation upon the prey is assumed to be proportional to the rate at which the predators and the prey are present concurrently, represented by bxy. If either x or y is zero, then no predation is possible.

Similarly, Eq. 24 shows that in the absence of prey ($x = 0$), the density of predators would decrease at a constant rate c, due to natural death or emigration. This equation assumes that the predator population only preys upon the same species of prey identified in Eq. 23. dxy represents the growth of the predator population [15].

The model assumes that throughout the process the external environment remains the same and none of the species is favored [16].

Results

From Fig. 5 we see that our neural net solver attains a close fit to the ODE solver. Training was done with a grid of 100 equidistant points in $t = [0, 10]$ and 50 hidden units in the hidden layer.

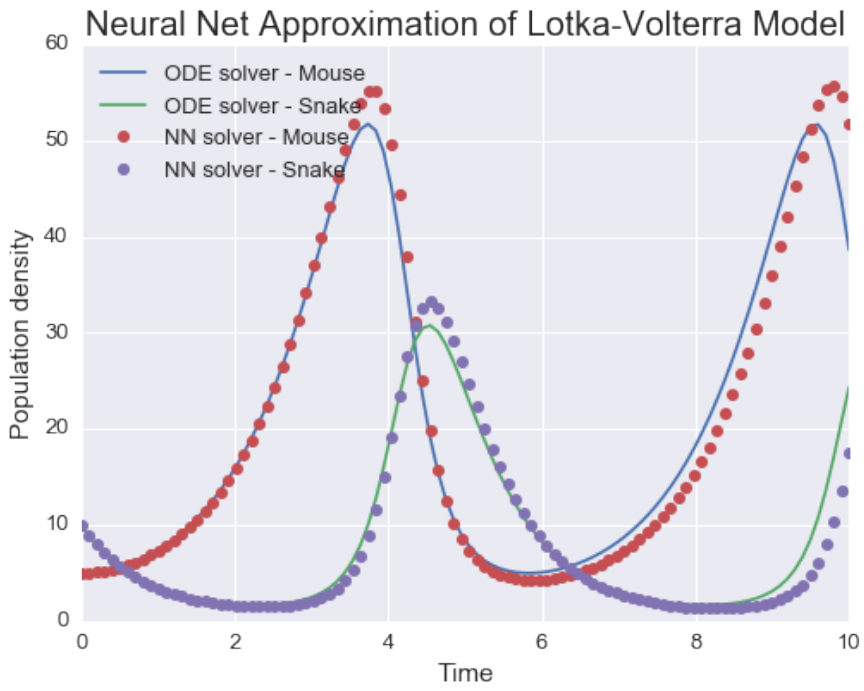


Figure 5: Comparing across results using standard ODE solver from Scipy library versus using neural net solver, at initial parameters of $a = 1, b = 0.1, c = 1.5, d = 0.75$. Average RMSE = 2.204.

In terms of training convergence, from Fig. 6, we can see that BFGS is selected as the optimization method as it results in the lowest loss compared to other methods.

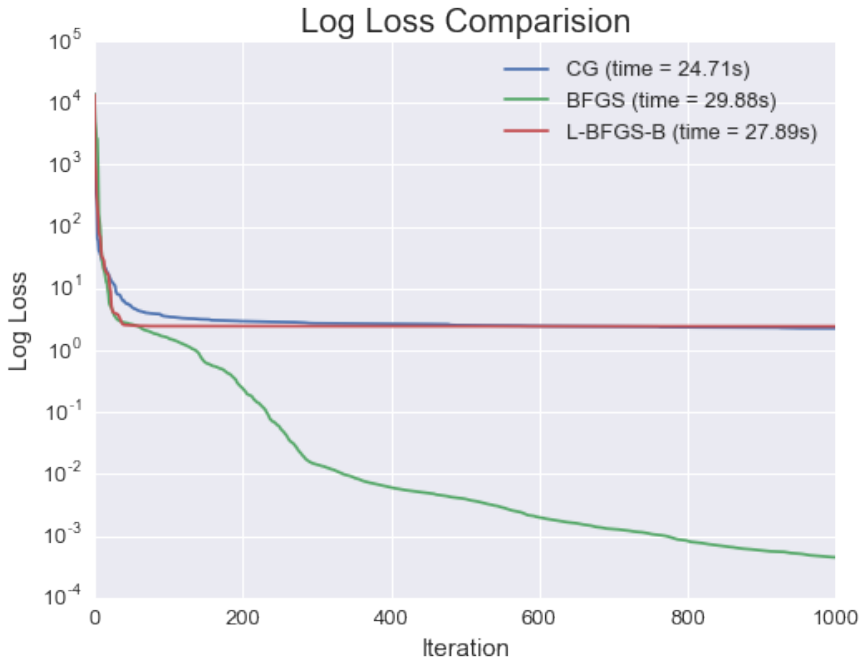


Figure 6: Comparison across different optimization methods for Lotka-Volterra 2-species model.

Discussions

The same as before, to enable us to evaluate the fit of the two functions in this system of ODEs, we propose taking an average of the RMSE, in this case we will have:

$$RMSE_{overall} = \frac{1}{2}[RMSE_{prey} + RMSE_{predator}] \tag{25}$$

For this particular plot, the overall average RMSE is at 2.204. It is found that while our proposed solver using neural net is able to provide a continuous function, the results tend to fluctuate highly. As such, we ran the same fitting over 100 times as shown in Fig. 7. 32% of the fittings result in relatively good fitting with average RMSE of less than 5.

In this problem, we also notice that at places where rapid changes in the magnitude occur, the results may not be as good as places with smooth transition. It may be improved by using more training data points where the rapid changes occur, instead of using equi-distant spacing of training data points everywhere.

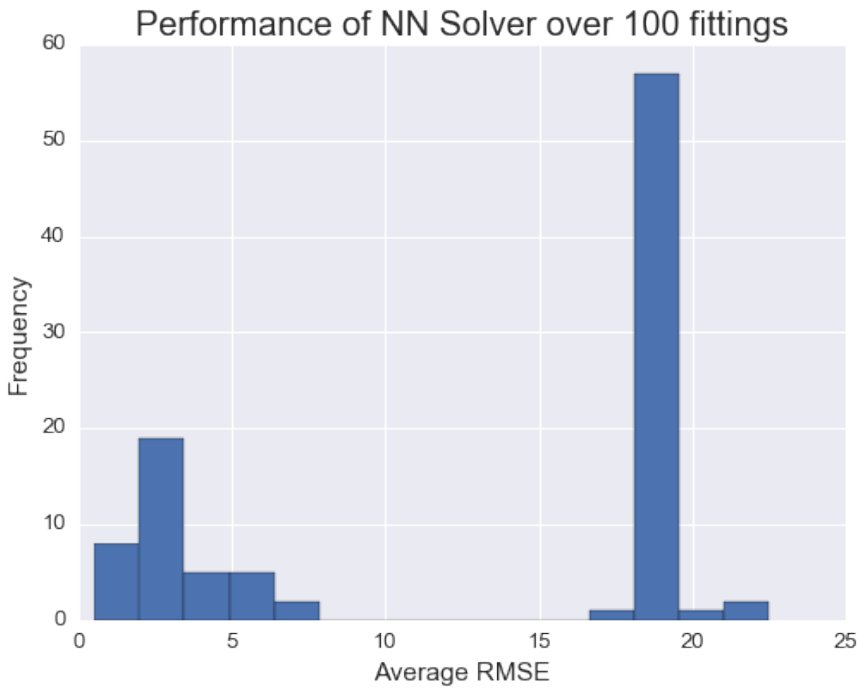


Figure 7: Average RMSE Distribution for 100 Repeated Fittings for Lotka-Volterra 2-species model

Please refer to **AM205_final/notebook/CHC/LV2.ipynb** for the code and visualizations of this section.

3.2.2 Lotka-Volterra 3-species Model

Extending to the 2-species model discussed earlier, we wish to model a linear three-species food chain where the lowest-level prey x is preyed upon by a mid-level species y, which, in turn, is preyed upon by a top level predator z. An example of a three-species food chain is mouse-snake-owl [17].

$$\frac{dx}{dt} = ax - bxy \tag{26}$$

$$\frac{dy}{dt} = -cy + dxy - eyz \tag{27}$$

$$\frac{dz}{dt} = -fz + gyz \tag{28}$$

where

- $a, b, c, d, e, f, g > 0$

- a, b, c, d are as in the 2-species Lotka-Volterra equations
- e represents the effect of predation on species y by species z
- f represents the natural death rate of species z in the absence of prey
- g represents the reproduction rate of species z in the presence of prey y

Results

The neutral net is trained with 50 hidden units over 100 equidistant points over $t \in [0, 10]$. As before, the optimization method is BFGS because the log loss is the smallest compared to other methods at the 200 iterations as shown in Fig. 8.

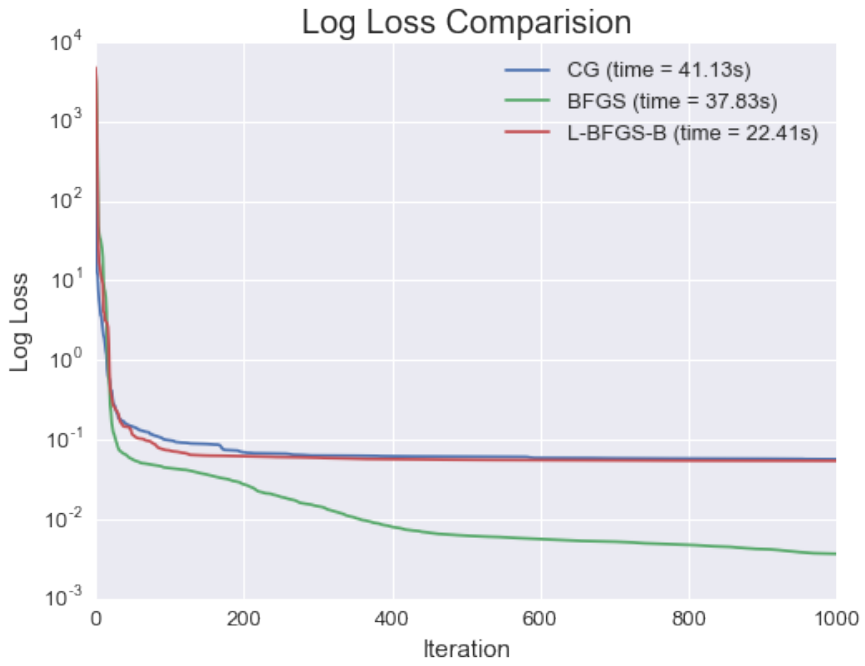


Figure 8: Comparison across different optimization methods for Lotka-Volterra 3-species model.

From Fig. 9, we can see that the performance is worse compared to the 2-species Lotka-Volterra model. The fit is close to the solution from ODE solver only up to $t = 2.5$ for the three equations. No further model fitting is carried out to examine reproducibility as the fitting is poor and the average RMSE score is not a good gauge to identify strong fit.

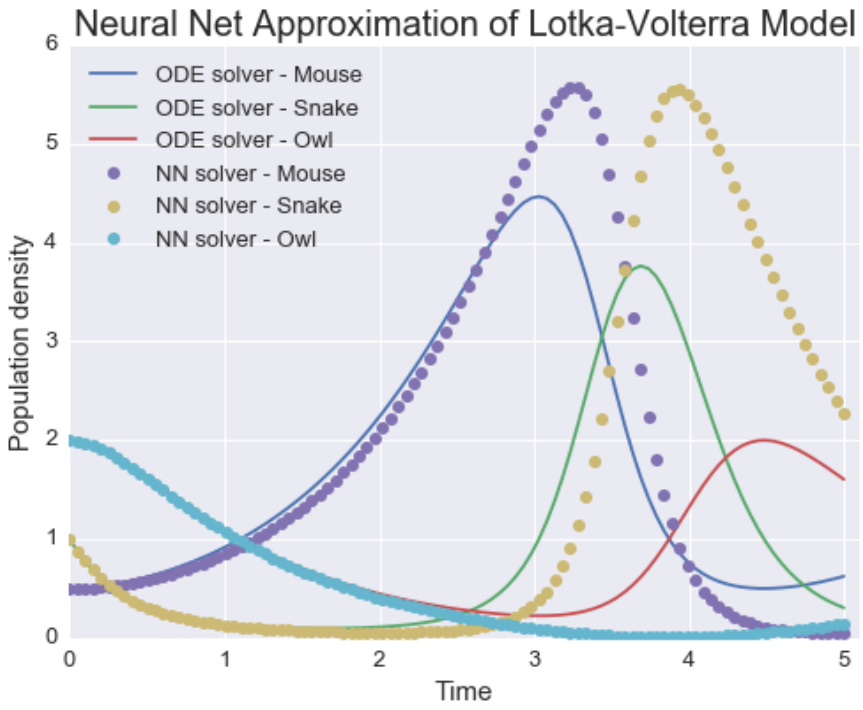


Figure 9: Comparing across results using standard ODE solver from Scipy library versus using neural net solver, at initial parameters of $a = b = c = d = e = f = g = 1$. Average RMSE score of 0.803.

Please refer to **AM205_final/notebook/CHC/LV3.ipynb** for the code and visualizations of this section.

3.2.3 Van der Pol oscillator

Overview

Van der Pol oscillator is a non-conservative oscillator with a linear spring force and a non-linear damping force [18]. The equation is given by:

$$\frac{d^2x}{dt^2} - \mu(1 - x^2)\frac{dx}{dt} + x = 0 \tag{29}$$

Mathematics

Applying the Liénard transformation $y = x - \frac{x^3}{3} - \frac{1}{\mu}\frac{dx}{dt}$, the equation can be written as a system of ODE [19]:

$$\frac{dx}{dt} = \mu(x - \frac{1}{3}x^3 - y) \tag{30}$$

$$\frac{dy}{dt} = \frac{x}{\mu} \tag{31}$$

Results

From Fig. 10 we see that our neural net solver attains a close fit to the ODE solver. Training was done with a grid of 40 equidistant points in $t = [0, 10]$ and 20 hidden units in the hidden layer. BFGS is used as the optimization method as we could see from Fig. 11 that the method attains the lowest log loss.

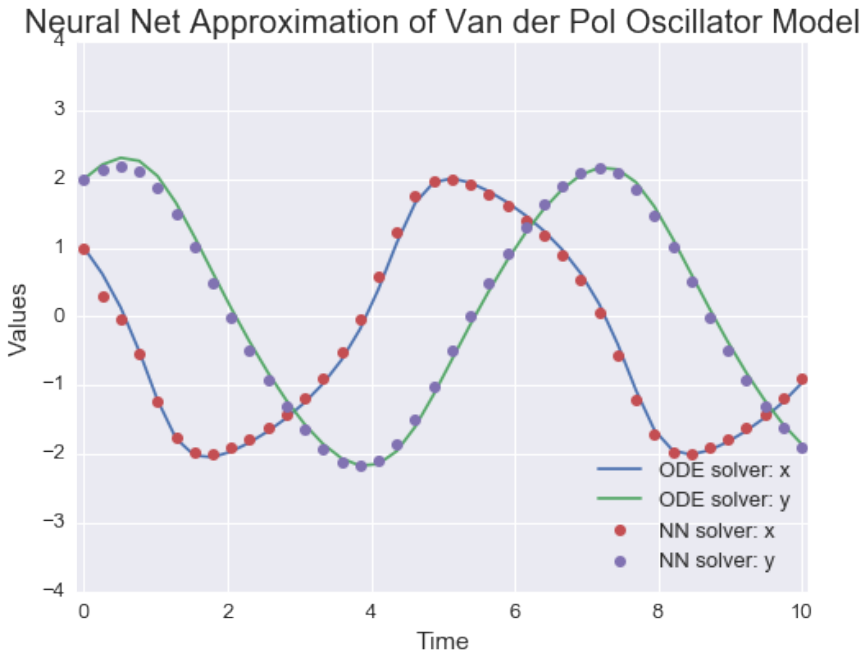


Figure 10: Comparing across results using standard ODE solver from Scipy library versus using neural net solver, at initial parameter of $\mu = 1$. Average RMSE = 0.0999.

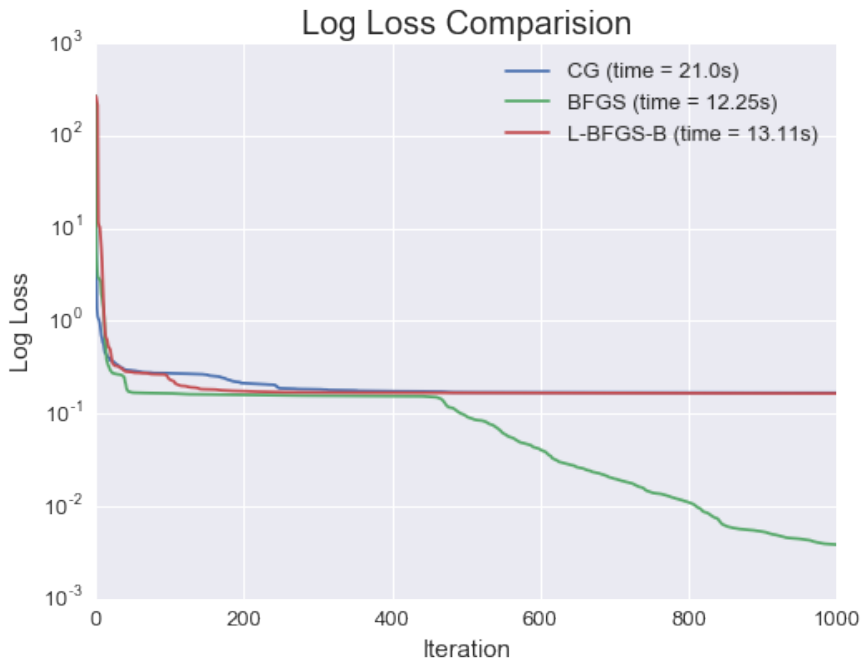


Figure 11: Comparison across different optimization methods for Van der Pol oscillator.

Discussions

The same as before, to evaluate the fit of the two functions in this system of ODEs, we propose taking an average of the RMSE, in this case the overall average RMSE for our fit in Fig. 10 is at 0.0999.

Since the fitting is satisfactory, we run the same fitting over 100 times to ensure reproducibility. From Fig. 12, 76% of the fittings result in relatively good fitting with average RMSE of less than 3.2.

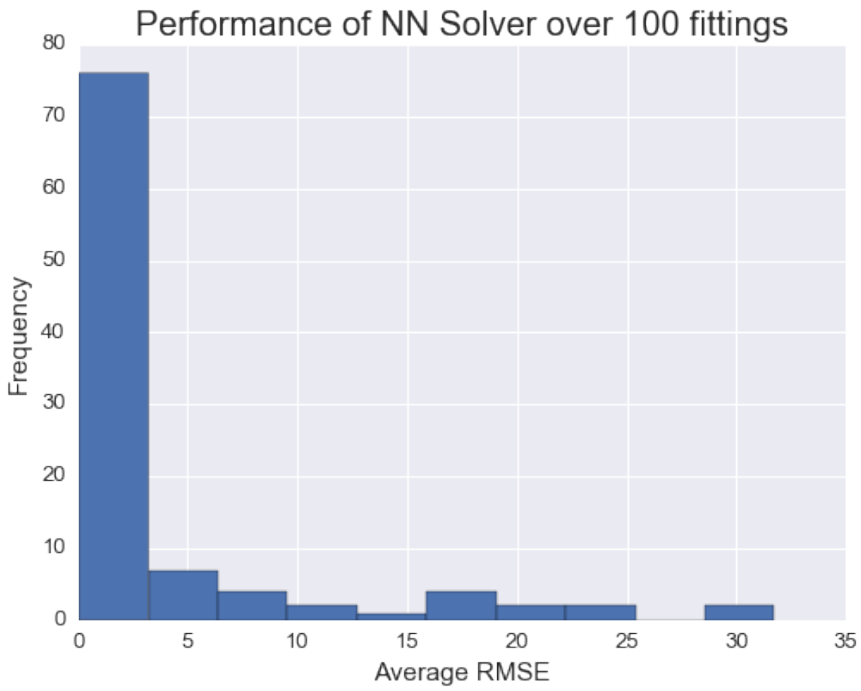


Figure 12: Average RMSE Distribution for 100 Repeated Fittings for Van der Pol oscillator.

Please refer to **AM205_final/notebook/CHC/Van_der_Pol_oscillator.ipynb** for the code and visualizations of this section.

3.3 Unstable, chaotic ODEs

3.3.1 Lorentz attractor

Overview

The Lorenz system, a system of three ordinary differential equations, was originally developed as a simplified mathematical model for atmospheric convection. It is non-linear, non-periodic, three-dimensional and deterministic. The Lorenz attractor is the set of chaotic solutions which, when plotted, resemble a butterfly [20].

The equations describe the rate of change of three quantities with respect to time: x is proportional to the rate of convection, y to the horizontal temperature variation, and z to the vertical temperature variation. The constants σ, ρ , and β are system parameters.

The equations are given by:

$$\frac{dx}{dt} = \sigma(y - x) \tag{32}$$

$$\frac{dy}{dt} = x(\rho - z) - y \tag{33}$$

$$\frac{dz}{dt} = xy - \beta z \tag{34}$$

Mathematics

We normally assumes that the parameters σ, ρ , and β are positive. Here we would use the same values as Lorenz did; $\sigma = 10$, $\rho = 8/3$, and $\beta = 28$. The system exhibits chaotic behavior for these values.

Results

From Fig. 13 we can see that our neural net solver attains a close fit to the ODE solver. Training was done with a grid of 40 equidistant points in $t = [0, 0.5]$ and 30 hidden units in the hidden layer. BFGS is used as the optimization method as we could see from Fig. 14 that the method attains the lowest log loss.

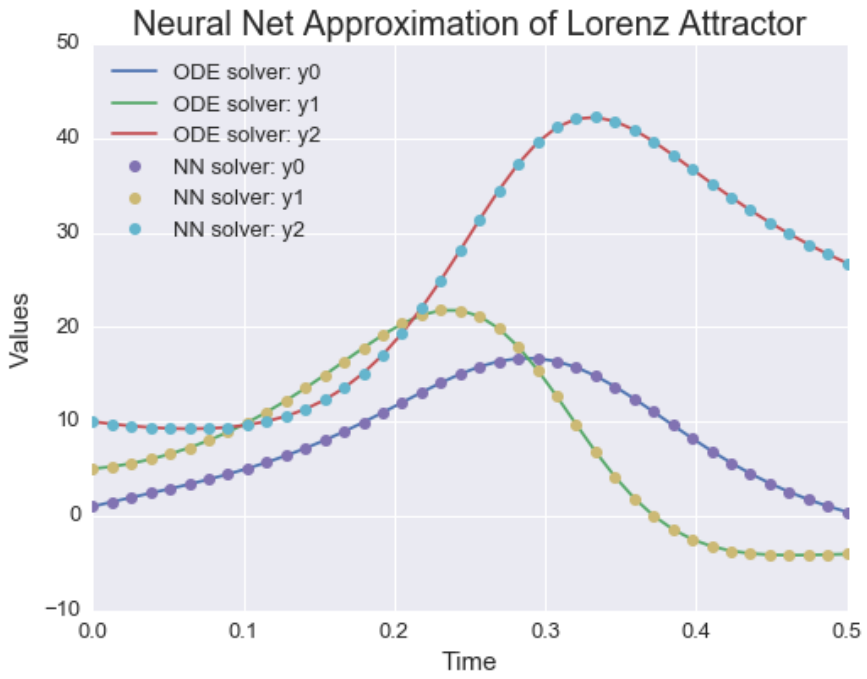


Figure 13: Comparing across results using standard ODE solver from Scipy library versus using neural net solver, at initial parameters of $\sigma = 10$, $\rho = 8/3$, and $\beta = 28$. Average RMSE = 5.93e-05.

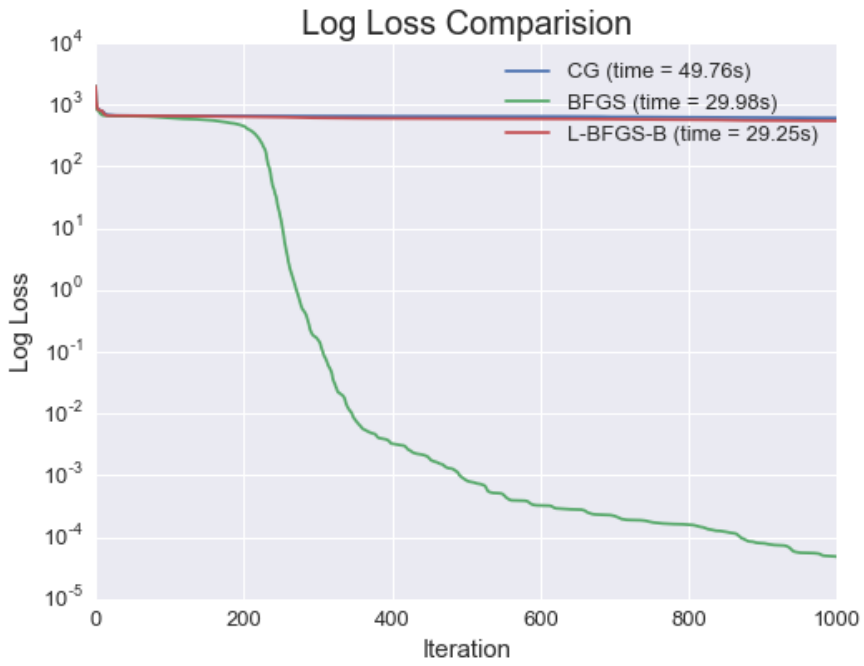


Figure 14: Comparison across different optimization methods for Lorentz Attractor.

Discussions

Similarly, in order to evaluate the fit of the three functions in this system of ODEs, we propose taking an average of the RMSE, in this case the overall average RMSE for our fit in Fig. 13 is at $5.93e-05$.

Since the fitting is satisfactory, we run the same fitting over 100 times to ensure reproducibility. From Fig. 15, 98% of the fittings results have relatively good fitting with average RMSE of less than 2.

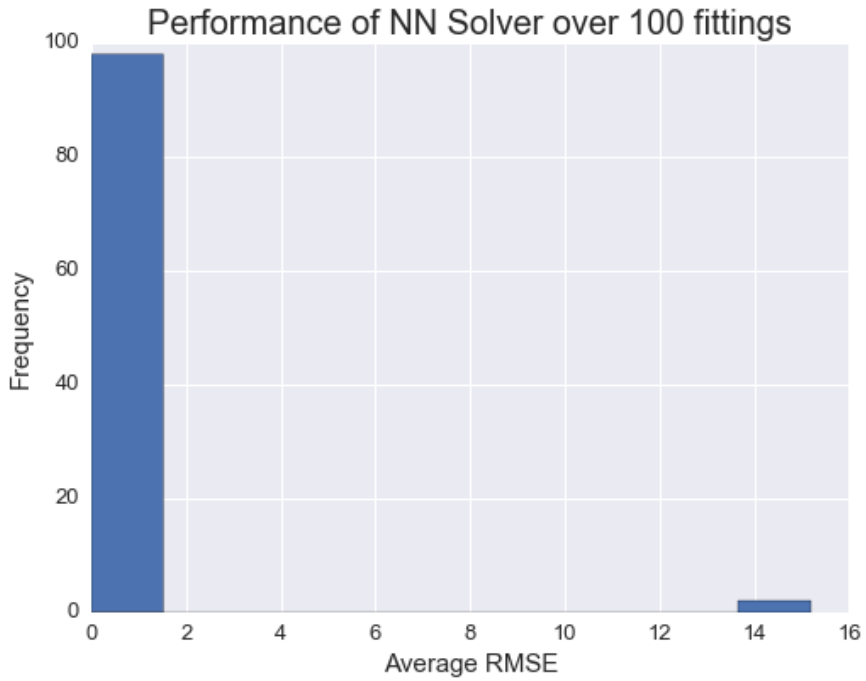


Figure 15: Average RMSE Distribution for 100 Repeated Fittings for Lorentz Attractor.

Please refer to **AM205_final/notebook/RZ/Lorenz_Attractor.ipynb** for the code and visualizations of this section.

3.3.2 Rossler attractor

Overview

The Rossler system is a system of three non-linear ordinary differential equations, which define a continuous-time dynamical system that exhibits chaotic dynamics associated with the fractal properties of the attractor.

Rossler attractor is the attractor for the system. This attractor is similar to the Lorenz attractor, but is simpler. An orbit within the attractor follows an outward spiral close to the x,y plane around an unstable fixed point. Once the graph spirals out enough, a second fixed point influences the graph, causing a rise and twist in the z-dimension. The oscillations are chaotic although each variable is oscillating within a fixed range [21].

The equations are given by:

$$\frac{dx}{dt} = -y - z \tag{35}$$

$$\frac{dy}{dt} = x + ay \tag{36}$$

$$\frac{dz}{dt} = b + z(x - c) \tag{37}$$

Mathematics

Rossler system has two fixed points. The first fixed point is located in the middle of the attractor and is a saddle-focus with an unstable 2D manifold - an unstable spiral mainly in the xy plane - when the trajectory settles down onto a chaotic attractor. The second fixed point is outside of the region of the attractor. The non-linearity z(x-c) becomes active when the trajectory leaves the xy plane. The trajectory thus visits the neighborhood of second fixed point whose 1D unstable manifold sends the trajectory along the 1D stable manifold of the first fixed point. A new cycle can then occur. With appropriate parameter values, the trajectory thus describes a chaotic attractor.

Results

From Fig. 16 we can see that our neural net solver attains a close fit to the ODE solver. Training was done with a grid of 40 equidistant points in $t = [0, 1]$ and 50 hidden units in the hidden layer. BFGS is used as the optimization method as we could see from Fig. 17 that the method attains the lowest log loss.

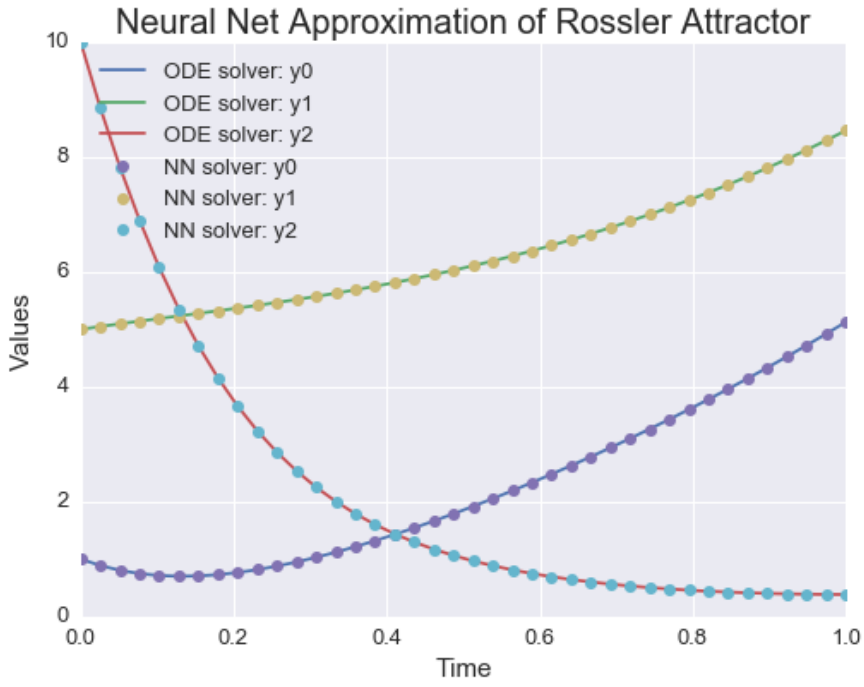


Figure 16: Comparing across results using standard ODE solver from Scipy library versus using neural net solver, at initial parameters of $a = 0.2$, $b = 0.2$, and $c = 5.7$. Average RMSE = 6.50e-05.

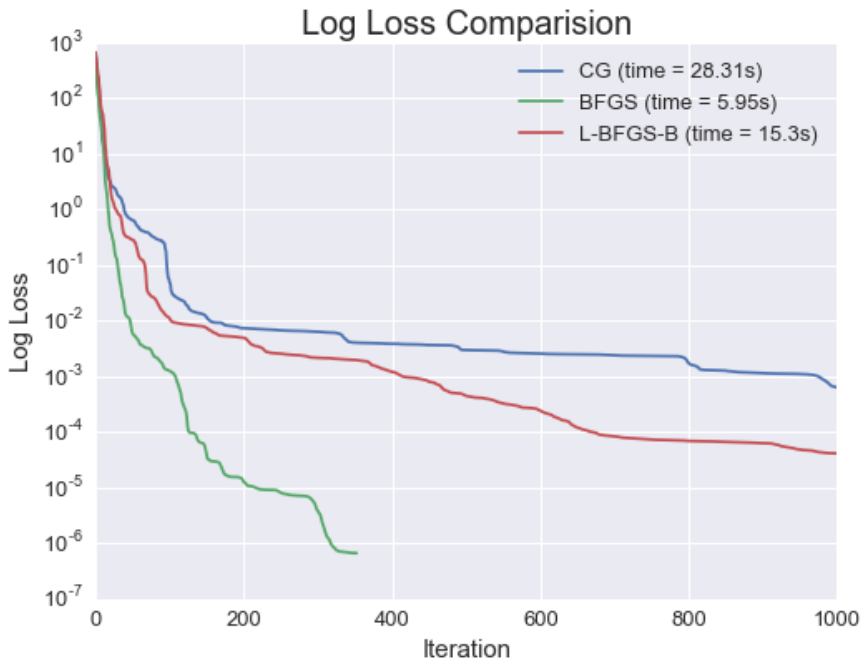


Figure 17: Comparison across different optimization methods for Rossler Attractor.

Discussions

Similarly, in order to evaluate the fit of the three functions in this system of ODEs, we propose taking an average of the RMSE, in this case the overall average RMSE for our fit in Fig. 16 is at $6.50e-05$.

Since the fitting is satisfactory, we run the same fitting over 100 times to ensure reproducibility. From Fig. 18, 99% of the fitting results is relatively good fitting with average RMSE of less than 0.00006.

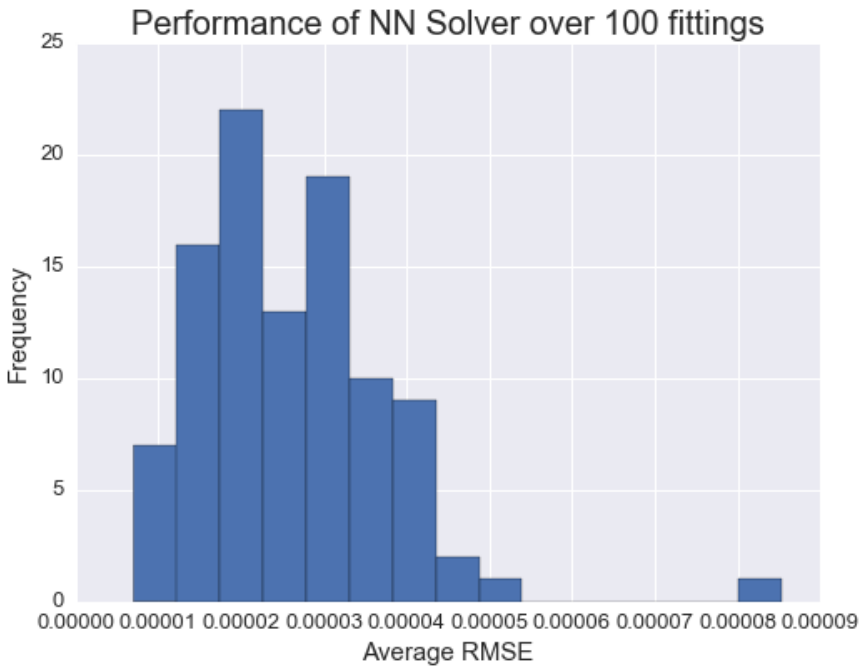


Figure 18: Average RMSE Distribution for 100 Repeated Fittings for Rossler Attractor.

Please refer to **AM205_final/notebook/RZ/Rossler_Attractor.ipynb** for the code and visualizations of this section.

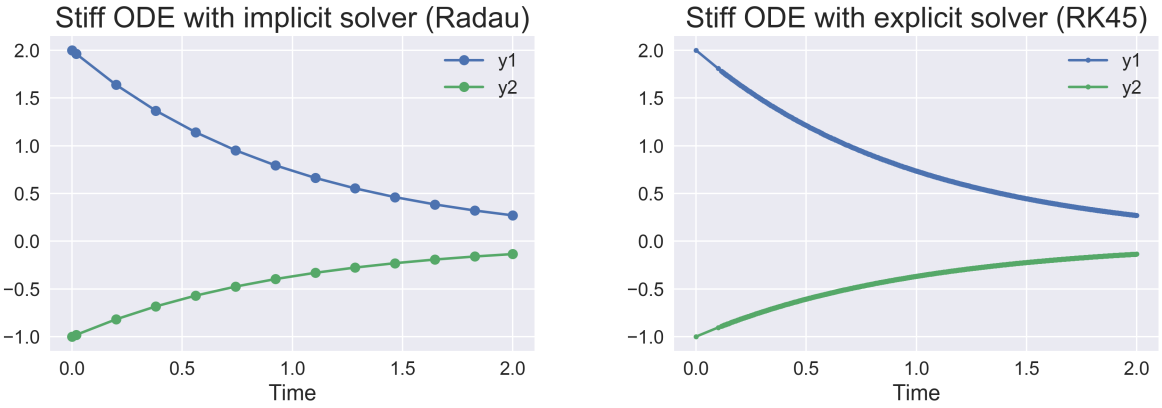
3.4 Stiff ODE system

One advantage of neural network method is it does not care if an ODE system is stiff or not, since the issue of stiffness only exists for iterative, finite-differencing method. Here we consider a typical stiff system:

$$\frac{dy_1}{dt} = 998y_1 + 1998y_2 \tag{38}$$

$$\frac{dy_2}{dt} = -999y_1 - 1999y_2 \tag{39}$$

The two eigenvalues of this system are -1 and -1000, thus the stiffness ratio is 1000. The initial condition is set to $[y_1, y_2] = [2, -1]$. This kind of system is generally to be solved by implicit methods, since explicit methods require very short time steps. Solutions by Scipy's implicit and explicit solvers are shown in Fig. 19. The explicit Runge-Kutta method needs more than 500 time steps to achieve stable results while the implicit Radau method only needs approximately 10 time steps. However, with neural network method, we can still use only approximately 10 training points without creating instability, as shown by Fig. 20.



(a) Implicit

(b) Explicit

Figure 19: Stiff ODE with standard solvers. Dots are actual time steps used by the solver.

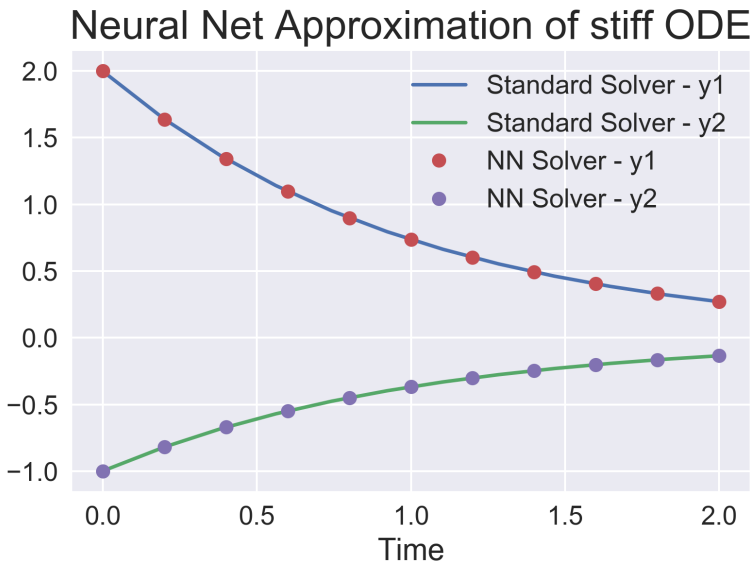


Figure 20: Stiff ODE with neural net solver.

Please refer to **AM205_final/notebook/JWZ/stiff_ODE_system.ipynb** for the code and visualizations of this section.

3.5 Extension: Flexible initial conditions

Finally, we extend the original theory [11] to take the initial condition also as an input parameter, so we can train the ANN only once and use it to retrieve the solution for different initial conditions.

Now our ANN has two inputs, t and y_0, and the ANN solution is rewritten as

$$\hat{y}(t, y_0; w) = y_0 + (t - t_0)N(t, y_0; w) \tag{40}$$

Notice that y_0 appears both inside the standard ANN term $N(t, y_0; w)$ and outside that term as the forced initial condition.

The training data is a grid mesh of different initial conditions y_0 and different t. Thus the loss function becomes

$$L(w) \approx \sum_j \sum_i [\hat{y}'(t_i, y_{0,j}; w) - F(\hat{y}(t_i, y_{0,j}; w), t_i)]^2 \tag{41}$$

To test this new method, We use a simple ODE with the initial condition y_0 ranging from 0 to 10.

$$\frac{dy}{dt} = -\sin(t)y \tag{42}$$

The solution is shown in Fig. 21. We can see this ANN method is able to approximate ODE solutions with multiple initial conditions, although the accuracy goes down at the last several time steps.

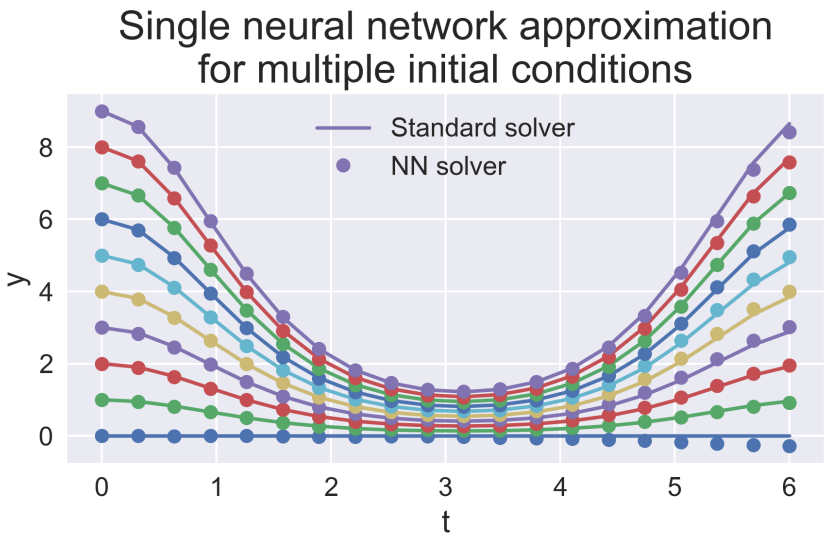


Figure 21: A single neural net for multiple initial conditions

Please refer to **AM205_final/notebook/JWZ/Flexible_initial_cond.ipynb** for the code and visualizations of this section.

4 Conclusions and Future Work

From the results and discussions above, we find that ANNs with one hidden layer in general show great capacity in approximating the solutions of ODEs.

However, this method still has some limitations. Firstly, the range of t cannot be very large, presumably because ANNs are sensitive to the scale of the input data. This means that ANN can only have good results for smaller values of t (starting from $t = 0$ to $t \approx 10$). Meanwhile, ANNs might not fit highly-oscillating solutions well. ANN may be unstable in yielding a good fit when 1) weights are not initialized properly, 2) function values change dramatically, while the training data points are evenly distributed within the range.

In the future, we plan to explore more about neural net architectures such as the number of layers and hidden nodes, the choice of activation function and the connectivity between layers, to see if there are some better choices for each different type of ODEs. We would also lay some emphasis on the optimization part, including efficiency and stability of different methods. Finally, we would like to explore some recent applications of deep ANNs on ODE problems, such as using recurrent neural networks to identify unknown parameters in ODEs, as presented on the Conference and Workshop on Neural Information Processing Systems (NIPS) 2017 [13].

5 Code Availability

We have made a Python package "neuralsolver" for this project (`https://github.com/JiaweiZhuang/AM205_final`). It has an API similar to the ODE solver in Scipy [14] and is able to solve general ODE initial value problems. Please refer to **AM205_final/README.rst** for installation instructions and **AM205_final/example/** for how to use the package. Our package depends on Scipy [14] for minimizing the loss function and autograd [22] for computing the derivative of numpy functions.

References

[1] H. Lee and I. S. Kang, Neural algorithm for solving differential equations, Journal of Computational Physics, vol. 91, no. 1, pp. 110131, 1990.

[2] Hornik, Kurt, Maxwell Stinchcombe, and Halbert White. "Multilayer feedforward networks are universal approximators." Neural networks 2.5 (1989): 359-366.

[3] Baymani, Modjtaba, Asghar Kerayechian, and Sohrab Effati. "Artificial neural networks approach for solving stokes problem." Applied Mathematics 1.04 (2010): 288.

[4] I. E. Lagaris, A. C. Likas, and D. G. Papageorgiou, Neural-network methods for boundary value problems with irregular boundaries, IEEE Transactions on Neural Networks, vol. 11, no. 5, pp. 10411049, 2000.

[5] K. S. McFall and J. R. Mahan, Artificial neural network method for solution of boundary value problems with exact satisfaction of arbitrary boundary conditions, IEEE Transactions on Neural Networks, vol. 20, no. 8, pp. 12211233, 2009.

[6] S. A. Hoda and H. A. Nagla, Neural network methods for mixed boundary value problems, International Journal of Nonlinear Science, vol. 11, pp. 312316, 2011.

[7] S. He, K. Reif, and R. Unbehauen, Multilayer neural networks for solving a class of partial differential equations, Neural Networks, vol. 13, no. 3, pp. 385396, 2000.

[8] L. Jianyu, L. Siwei, Q. Yingjian, and H. Yaping, Numerical solution of elliptic partial differential equation using radial basis function neural networks, Neural Networks, vol. 16, no. 5-6, pp. 729734, 2003.

[9] Y. Shirvany, M. Hayati, and R. Moradian, Multilayer perceptron neural networks with novel unsupervised training method for numerical solution of the partial differential equations, Applied Soft Computing Journal, vol. 9, no. 1, pp. 2029, 2009.

[10] C. Franke and R. Schaback, Solving partial differential equations by collocation using radial basis functions, Applied Mathematics and Computation, vol. 93, no. 1, pp. 7382, 1998.

[11] Lagaris, Isaac E., Aristidis Likas, and Dimitrios I. Fotiadis. "Artificial neural networks for solving ordinary and partial differential equations." IEEE Transactions on Neural Networks 9.5 (1998): 987-1000.

[12] Chiaramonte, M. M., and M. Kiener. "Solving differential equations using neural networks." Machine Learning Project.

[13] Hagge, Tobias, et al. "Solving differential equations with unknown constitutive relations as recurrent neural networks." arXiv preprint arXiv:1710.02242 (2017).

[14] Jones, Eric, Travis Oliphant, and Pearu Peterson. "SciPy: open source scientific tools for Python." (2014).

[15] https://en.wikipedia.org/wiki/Lotka-Volterra_equations

[16] http://www.math.harvard.edu/library/sternberg/slides/11809LV.pdf

[17] http://people.kzoo.edu/barth/math280/articles/3speciesLV.pdf

[18] http://www2.me.rochester.edu/courses/ME406/webexamp5/vanpol.pdf

[19] Kaplan, D. and Glass, L., Understanding Nonlinear Dynamics, Springer, 240244, (1995).

[20] http://mathworld.wolfram.com/LorenzAttractor.html

[21] http://www.scholarpedia.org/article/Rossler_attractor

[22] Maclaurin, Dougal, David Duvenaud, and Ryan P. Adams. "Autograd: Effortless gradients in numpy." ICML 2015 AutoML Workshop. 2015.