Math104B - Final Project Report

Ivan Gvozdanovic

June 2019

Introduction

The problem we present in the following report revolves around finding a way to model an ODE or a system of ODEs using finite number of input-output data pairs and a feed forward Neural Network. What we mean by modeling is that we want to find the law of an ODE, i.e. $y' = f(t, y)$, where f is the desired law. These kinds of problems emerge in application very often, and thus we want to find a way to fully unlock the information of an ODE using limited data. In our case, we are trying to model Zhao's dynamical system

$$\begin{aligned} x' &= a(y - x) \\ y' &= bx - xz \\ z' &= xy + cz \end{aligned}$$

with the initial starting point $(-5, -5, 15)$ and constants $a = 10,\ b = 10,\ c = 1$. Even though we know how the system of ODEs looks like, from this point on we assume that we do not have this information. Thus, we will only use a set of input-output pairs to model the system above. Moreover, note that we are dealing with a system that is chaotic and also the problem is not well-posed. However, the goal of the project, mentioned above, is to model an ODE system and so Zhao's system presented will work well for our purposes.

Approach and Method

In order to set up the problem, we assume that we are given a finite amount of data points. In a real world problem, this set would possibly come from a physical experiment. However, in order to show that the approach works, we use Euler's method to generate a set ranging from 10 to 3000 points as our input data and then produce the output pairs by plugging these into the ODE system itself. This set is not fixed because there are interesting results, that we will present later, that depend on number of points as well as the value of the step-size h in Euler's method.
Finally, we split this set into a training set (about 2/3 of total points) and into

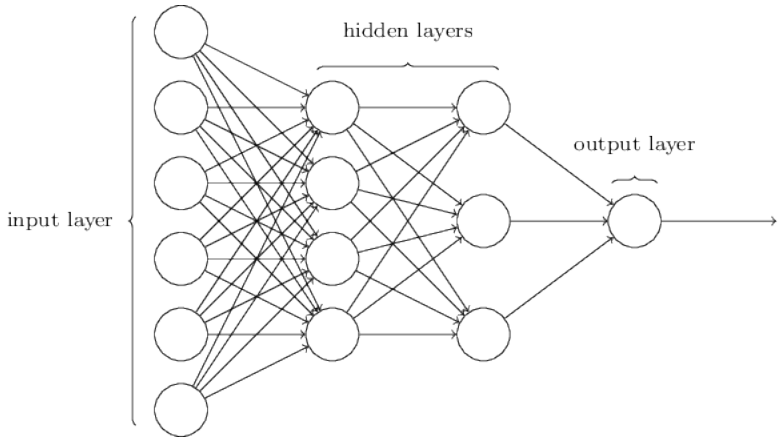


Figure 1: NN structure

a testing set which are both used in the method mentioned below.

As mentioned in the Introduction, the method we are using is a Neural Network. More specifically, a feed forward Neural Network represents a sequence of matrix operations that models how neurons in our brain work. The figure above models an arbitrary feed forward Neural Network with 1 input , 2 hidden and 1 output layer. In our case, we are using a similar structure. However, the NN we are using has an input layer that takes vectors $(x, y, z)^t$ and thus has 3 nodes for each coordinate. Moreover, we are using 2 hidden layers with around 100 nodes each and an output layer of 3 nodes that produces $(\hat{x'}, \hat{y'}, \hat{z'})^t$ approximations of the model.

Forward propagation: As mentioned earlier, a feed forward NN can be structured by using matrix operations. So, we have the following setup,

$$\begin{aligned}
&\mathbf{X} \cdot \mathbf{W_1} = \mathbf{Z_1} \\
&\mathbf{A_2} = \mathbf{ReLU}(\mathbf{Z_1}) \\
&\mathbf{Z_2} = \mathbf{A_2} \cdot \mathbf{W_2} \\
&\mathbf{A_3} = \mathbf{ReLU}(\mathbf{Z_2}) \\
&\mathbf{Z_3} = \mathbf{A_3} \cdot \mathbf{W_3} \\
&\hat{\mathbf{Y}} = \mathbf{ReLU}(\mathbf{Z_3})
\end{aligned}$$

In the above expression, $\mathbf{X}$ is the input matrix who's rows represent separate input data. So in our case, if we are using 1000 points, the matrix $\mathbf{X}$ will have 667x3 (around 2/3 of points for training) dimension where 3 comes from the $(x, y, z)^t$ input. Next, we matrix multiply the inputs with weights $\mathbf{W_1}$ connecting input nodes and nodes in the first hidden layer and then apply the

Rectified Linear Unit to obtain the activation matrix $\mathbf{A_2}$. The activation has dimension 667xC where C stands for the number of hidden nodes (columns) of $\mathbf{W_1}$ that we specify. Repeating the same process again, we get $\mathbf{A_3}$ with a certain dimension and finally we repeat it yet again to obtain $\hat{Y}$ which is a 667x3 matrix representing the approximation of the model for the given inputs.
Backwards propagation: In order to train the neural network, we need to somehow tell it the error in its approximation and then make it adjust the hidden weights so that in the next iteration, the value of the loss function is smaller. First, the loss function that we are using is the Mean Square Error defined as

$$\frac{1}{n}\sum_{i=1}^{n}(F(t,x_i,y_i,z_i) - N(w,x_i,y_i,z_i))^2$$

where $F(t,x_i,y_i,z_i)$ represents the target values we obtained by plugging Euler approximations into the ODE system and $N(w,x_i,y_i,z_i)$ represents the approximations computed by the NN. Next we apply Stochastic Gradient Descent in order to minimize the loss function. SGD is similar to regular gradient descent except the fact that we are not minimizing the loss function with respect to all input examples. On the contrary, we define a batch size and then minimize the loss based on this subset. The advantage of using SGD is that the gradient descent can escape local minima in the "energy" plane consisted of mountains and valleys which can be troublesome for convergence of the loss function. After computing the correct gradient direction, we update the weights by the following formula,

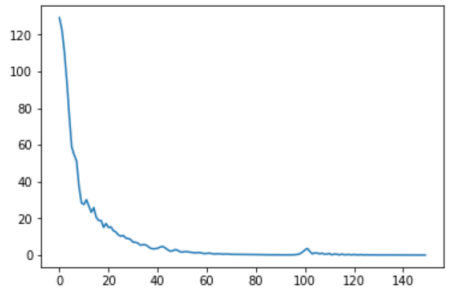
$$\mathbf{W} = \mathbf{W} - \mathbf{lr}\cdot\nabla\mathbf{E}$$

where $\mathbf{W}$ represents an arbitrary matrix containing weights, lr represents the learning rate (step-size) of the SGD and ∇E represents the gradient of the loss function defined above. Besides, weights we also update the biases in a similar manner.
Once we finish the training process, we save the last updated weights and use the testing set (around 1/3 of total input points) to test the accuracy of our NN.

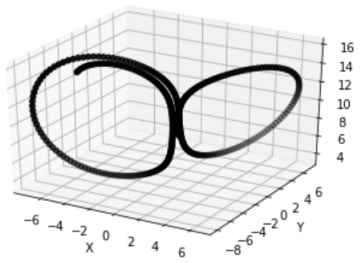
Results

In this section we present the results of our NN. As mentioned earlier, the accuracy of the NN strongly depends on the interval we are modeling the system on. What we mean by this is that if we use a lot of points with a large step-size h in the Euler' method, the network has a hard time fitting the data due to the complexity of the system. Even if we increase the number of hidden layers and even the number of hidden nodes, the NN still has trouble

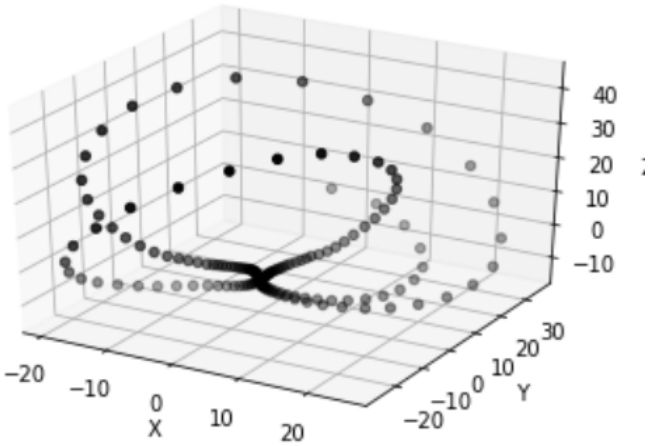
capturing the trajectories of the system. Using trial and error, we deduced that an optimal number of points that we use in the training and testing set is around 500 with the Euler step-size $h = 0.001$. When using this setup with the learning rate 0.01, number of hidden nodes 100 and number of epochs 200, the average error of the testing set is about 0.4 which is not that bad considering that we started with Euler approximated inputs which had an error built into them to begin with. Moreover, in the testing we noticed that the NN has trouble fitting the first point in the system from which the biggest error comes from. Therefore, one solution is to extend the testing data set by couple of points and then consider all the points without the first point (first n points). This approach is very effective and lowers the max error by around 0.9 which is a lot. Thus, the max error we get is 0.8 and the minimum error of the test set is around 10^{-4}. Visually, the results look as follows,



(a) Loss



(b) Euler Solution to the system



(c) Model of the ODE system

As the reader can see, we picked just enough points to cover the full geometry of the ODE system. Moreover Figure (a) shows the minimization of the Loss function. Even though it seems that we over fitted the data, this is not the

case. Increasing the number of epochs and the number of hidden nodes and leaving all other parameters untouched, we conclude that over fit occurs around 350 epochs and 100 hidden nodes (at this threshold the loss starts to increase again). Lowering the epochs to 200, we get the result above which has the average error = 0.4, maximum error = 0.85 and minimum error = 0.0001. Therefore, we obtained the approximation of $S' = F$ and therefore we have that $S' \approx N(w, y)$, where S' represents the ODE system, F is the law we modeled and $N(w, \vec{v})$ represents our neural network with weights w and inputs $\vec{v}$.

Conclusion

Our goal for this project was to verify that we can use NNs to model ODEs. Although we got the accuracy of the test set as mentioned above, we have to take into consideration that we used Euler to obtain the inputs. It is highly likely that if we used methods such as Runge-Kutta we would start with better approximations and thus get better outputs. Nevertheless, we showed that NNs have great potential in this field of study. The true power of this method comes from its application to real world problems. More often than not, we cannot obtain explicit forms of ODEs and ODE systems and thus we have to rely on approximations. However, using only a small number of points, our NN modeled a complex system of ODEs with a relatively good accuracy. From this point, we can use Euler's method with our NN to get approximations of solutions that look like

$$\vec{v}_{n+1} = \vec{v}_n + hN(w, \vec{v}_n).$$

However, we are not constricted to Euler's method only. We can use any iterative method from this point on in order to obtain solutions to the ODE system.

The next step in this method would be to try to use different NN architecture in order to obtain better accuracy of the ODE system as well as to expend the application to PDEs.