Solving ODE by artificial neural networks with Knet.jl and Optim.jl

Jiawei Zhuang
(jiaweizhuang@g.harvard.edu)

2017 Fall

Contents

1 Introduction

Ordinary differential equations (ODEs) are generally solved by finite-differencing 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.

The artificial neural network (ANN) method [2] provides a way to obtain ODE solutions 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.

2 Mathematical theory

2.1 Fitting functions by artificial neural networks

The universal approximation theorem [1] 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 a single ODE by artificial neural networks

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

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

If we use a standard neural network

$$N(t; w) = W_2\sigma(W_1t + b_1) + b_2 \tag{6}$$

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 $[t_0, t_1]$:

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

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 a system of two ODEs

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

We can use two separates 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 Implementation in Julia

3.1 Design idea

To implement this neural network solver, we combine Optim.jl[6] and Knet.jl[3] (more specifically, its AutoGrad part[5]). Unlike implementing normal ANNs with Knet, here we only use Knet to compute $\partial\hat{y}'(t_i)/\partial t$, but use Optim.jl for ANN training. The reason is explained in the "Use second-order optimization" section below.

We use standard ODE solvers in DifferentialEquations.jl[4] to benchmark our ANN solution.

3.2 Technical challenges

3.2.1 Vectorizing autograd

Unlike in ordinary ANNs where we only need the gradient of the loss $L(w)$ w.r.t weights w, in this ODE-ANN theory we need to take derivative w.r.t.to the input variable t, as suggested by Eq. (9). This derivative was hand-coded in the original paper [2], but here we would like to utilize the automatic differentiation in Julia.

However, a challenge is both AutoGrad.jl[5] and ForwardDiiff.jl[7] assume scalar-valued functions, but our ANN prediction function is vectorized over the input parameter x:

```
function predict(params, x; act=tanh)
    W1, b1, W2, b2 = params
    a = act.(W1*x .+ b1)
    y = W2*a .+ b2
    return y
end
```

There are two ways to vectorize its gradient over x. One is to assume x is a scalar and use "dot" to vectorize over an array. Another is to take gradient w.r.t to the scalar $y_{sum} = y_1 + y_2 + ... + y_n$, where y_i is the vectorized output from input x_i. Each component of the gradient would be

$$\frac{\partial y_{sum}}{\partial x_i} = \frac{\partial y_i}{\partial x_i} \tag{18}$$

since y_i only depends on x_i, not on x_j $(j \neq i)$.

The code is implemented as

```
sum_predict(x) = sum(predict(params, x))
predict_grad = grad(sum_predict)
```

We found the second method (gradient-of-sum) is 10 times faster than the first method ("dot" vectorize). See **prototype/benchmark_NNpredict.ipynb** for the timing comparison. The gradient-of-sum method is also how the Python version of Autograd [8] vectorize over input parameters. This functionality is not in Knet yet, and having this as a convenient function might be helpful.

3.2.2 Use second-order optimization

This ODE-ANN theory was already implemented in the experimental NeuralNetDiffEq.jl [9], but that implement was clearly not working, according to the results they have shown. The problem is NeuralNetDiffEq.jl uses stochastic gradient descent (SGD) methods in Knet. Althouth SGD methods and its variants like the Adam method are widely used in deep learning [10], they are not suited for fitting smooth functions or approximate ODE solutions. SGD methods are only able to get near the minimum, but to land exactly at the minimum we need second-order optimization like the BFGS method [11]. This is illustrated in more details in the numerical results section.

Knet doesn't have second-order optimization methods, so we need to resort to Optim.jl. An annoying thing here is Optim.jl expects the input parameter to be an 1-D vector, but the neural network parameters $[W_1, W_2, b_1, b_2]$ are typically nested arrays. The nesting gets deeper for ODE systems where we need multiple ANNs and thus many groups of parameters.

This parameter shape problem is an existing issue [12] in Optim.jl and there are tools being developed to address it [13]. But at this stage I have to hand-code the functions to flatten (for Optim.jl) and unflatten (for prediction) ANN parameters. The functions are implemented in **NN_util.jl**. The overhead of flatten&unflatten should be minimal as shown by **prototype/benchmark_NNpredict.ipynb**, but this solution is by-no-means elegant and doesn't generalize well. A much better solution in the future would be allowing Optim.jl to take nested arrays of any shapes.

3.3 Neural-solver API

Finally, I built an API (see **NN_solver.jl** and **Use_API.ipynb**) similar to DifferentialEquations.jl, which is able to solve general ODE systems.

The core computation part is not large, but there are many boilerplate codes to track the shapes of ANN weights for flatten&unflatten operations. The API internal could be much clearer and better optimized if Knet has second-order optimization methods or if Optim.jl can take arbitrary shapes.

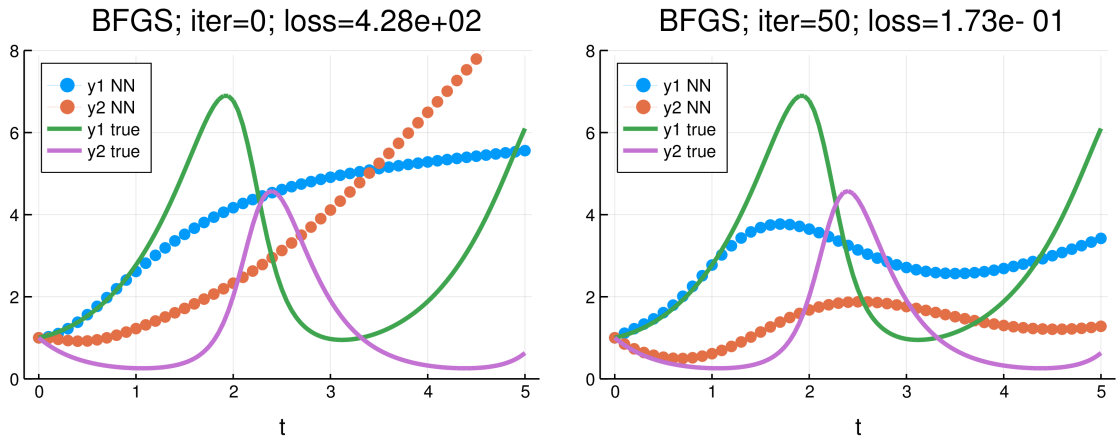
4 Results and discussions

Our API is able to solve general ODE systems, but in this report we focus on a single problem: the Lotka-Volterra problem that the original NeuralNetDiffEq.jl [9] totally failed to solve. The ODE is defined as

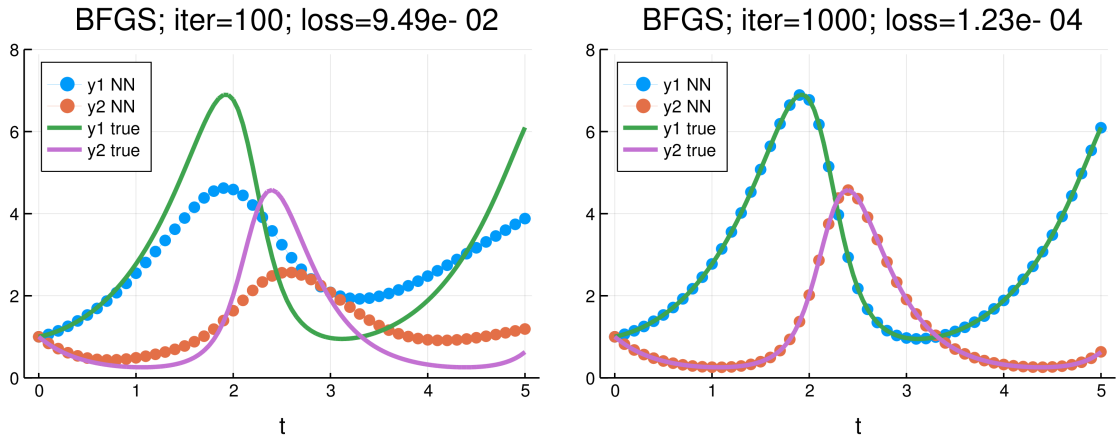
$$y_1'(t) = 1.5y_1 - y_1 * y_2, \quad y_1(0) = 1 \tag{19}$$
$$y_2'(t) = -3y_2 + y_1 * y_2, \quad y_2(0) = 1 \tag{20}$$

The training process is shown Fig.1. The solid lines are the reference solution given by DifferentialEquations.jl, and the dotted lines show the neural network prediction during the

training process, with the BFGS optimizer. We can see the loss drops from 400 to 10^{-4} and the ANN prediction is converging to the reference solution. The initial guess is only correct at the initial point ($t = 0$) as forced by Eq. (9), but is incorrect elsewhere. After 1000 training steps, the final prediction is visually indistinguishable from the reference "true" solution. Animation for the entire training process is available as an MP4 file **BFGS.mp4**.



(a) initial guess (b) train for 50 steps

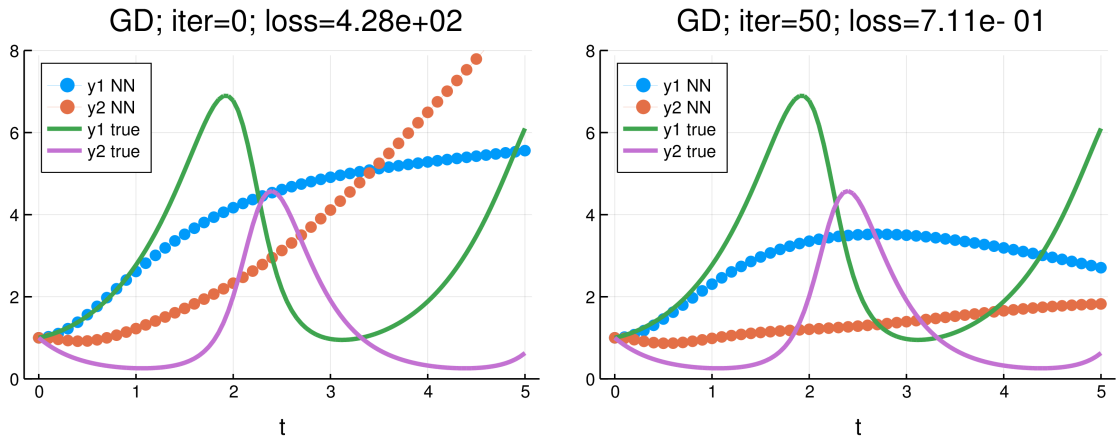


(c) train for 100 steps (d) final result (1000 steps)

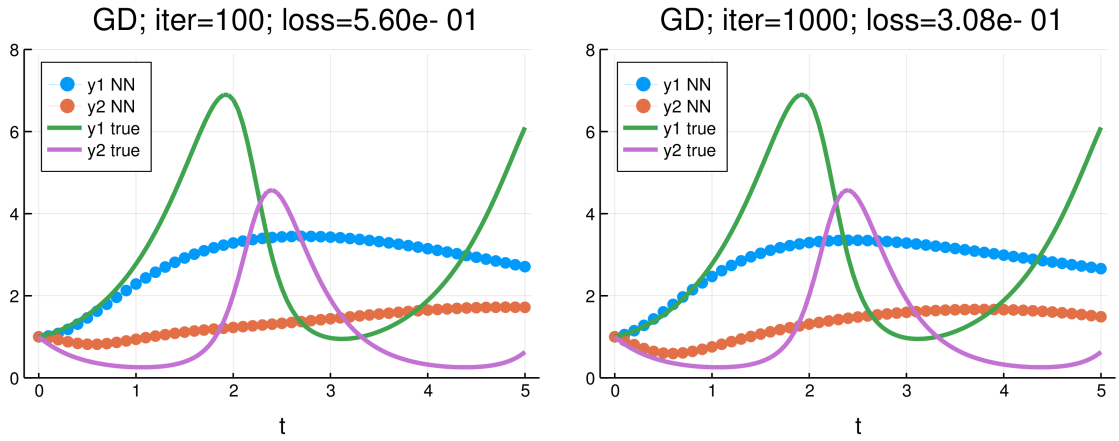
Figure 1: Training with BFGS

Then we demonstrate in Fig.2 how this ANN method fails with first-order optimizers like the basic Gradient Descend. The loss drops from 400 to 10^{-1} quickly but cannot decrease further. Animation for the entire training process is available as an MP4 file **GD.mp4**.

Although such a magnitude of loss might be small enough for regression or classification problems, it is not enough for accurately fitting a smooth function at every point. First-order methods are only able to get near the minimum, but to land exactly at the minimum we need second-order methods. This explains why NeuralNetDiffEq.jl [9] with the Adam optimizer (SGD with momentum and adaptive learning rate) cannot solve this problem.



(a) initial guess (b) train for 50 steps



(c) train for 100 steps (d) final result (1000 steps)

Figure 2: Training with Gradient Descend (GD)

Codes for this section are in **train_Lotka_Volterra.ipynb** and **Plot_Lotka_Volterra.ipynb**.

5 Conclusions and suggestions

5.1 Advantages and limitations of neural network method

We have developed a neural network ODE solver with Knet.jl and Optim.jl. The solution is analytically differentiable and requires less storage than traditional finite-differencing methods.

However, a major limitation of this method is it cannot deal with large t. Although our method is successful in solving the Lotka-Volterra for $t \in [0, 5]$, it usually fails for $t \in [0, 20]$ or larger. The major reason is ANNs are scale-aware – the input data often need to be scaled to [-1,1], but there is no easy way to scale an ODE problem. Only scaling t will increase the gradient dy/dt, making the function oscillating more drastically and harder to fit.

5.2 Suggestions for package interoperability in Julia

This project suggests that Optim.jl might be quite useful for deep learning. Besides in our use case, the BFGS method is also useful in more standard deep learning applications like style transfer [14].

Second-order methods (e.g. BFGS, L-BFGS, Conjugate Gradient) are not very well supported in current deep learning frameworks, because they are harder to code than first-order methods (e.g. SGD, Adam):

- Tensorflow's second-order optimizer (L-BFGS) is a wrapper around scipy, which is again a wrapper around the Fortran minpack;
- Pytorch's L-BFGS optimizer is ported from Lua torch and has an API different from other methods;
- MXNet does not have second-order methods at all;
- Julia machine learning packages such as Knet.jl and Flux.jl [15] also do not have second-order methods at all. But fortunately we are inside Julia framework so we can resort to Optim.jl

One remained problem, though, is Optim.jl cannot run on GPUs. If GPU libraries can interoperate with native Julia libraries like Optim.jl, they will open a lot of new opportunities.

References

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

[2] 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.

[3] Yuret, Deniz. "Knet: beginning deep learning with 100 lines of julia." Machine Learning Systems Workshop at NIPS. Vol. 2016. 2016.

[4] Rackauckas, Christopher, and Qing Nie. "DifferentialEquations. jl A Performant and Feature-Rich Ecosystem for Solving Differential Equations in Julia." Journal of Open Research Software 5.1 (2017).

[5] https://github.com/denizyuret/AutoGrad.jl

[6] https://github.com/JuliaNLSolvers/Optim.jl

[7] https://github.com/JuliaDiff/ForwardDiff.jl

[8] https://github.com/HIPS/autograd

[9] https://julialang.org/blog/2017/10/gsoc-NeuralNetDiffEq

[10] Ruder, Sebastian. "An overview of gradient descent optimization algorithms." arXiv preprint arXiv:1609.04747 (2016).

[11] Fletcher, Roger. Practical methods of optimization. John Wiley & Sons, 2013.

[12] https://github.com/JuliaNLSolvers/Optim.jl/issues/399

[13] https://github.com/JuliaDiffEq/RecursiveArrayTools.jl

[14] https://blog.slavv.com/picking-an-optimizer-for-style-transfer-86e7b8cba84b

[15] https://github.com/FluxML/Flux.jl