Neural ODE meeting reading

For meeting in Data Analytics

Yingzi Bu

August 6, 2024

- neural ODE simple explanation
- more explanations for neural networks as ODE, adjoint method
- github for neural ODE, pytorch
- vector jacobian in torch, math explanation
- adjoint sensitivity analysis, math explanation
- adjoint math explanation
- physics informed neural networks

1 Neural networks as ordinary differential equations

The core idea is that certain types of neural networks are analogous to a discretized differential equation, so maybe using off-the-shelf differential equations solvers will help get better results. Typically, we think about neural networks as a series of discrete layers, each one taking in a previous state vector $\mathbf{h}_n$, and producing a new state vector $\mathbf{h}_{n+1} = F(\mathbf{h}_n)$. Here, let's assume that each layer is the same width. Note that we don't particularly care about what F looks like, but typically it's something like $F(x) = \sigma(\Sigma_i \theta_i x_i)$ where σ is an activation function (e.g. relu or a sigmoid), and θ is a vector of parameters we are learning. This core formulation has some problems - notably, adding more layers, while theoretically increasing the ability of the network to learn, can actually decrease accuracy of it, both in training and testing results.
This problem was addressed by Deep Residual Learning [1]:

"Driven by the significance of depth, a question arises *Is learning better networks as easy as stacking more layers?* An obstacle to answering this question was the notorious problem of vanishing/exploding gradients, which hamper convergence from the beginning. This problem, however, has been largely addressed by normalized initialization and intermediate normalization layers, which enable networks with tens of layers to start converging for stochastic gradient descent (SGD) with backpropagation.
When deeper networks are able to start converging, a *degradation* problem has been exposed: with the network depth increasing, accuracy gets saturated (which might be unsurprising) and then degrades rapidly. Unexpectedly, such degradtation is *not caused by overfitting*, and adding more layers to a suitably deep model leads to *higher training error*.
The degradation (of training accuracy) indicates that not all systems are similarly easy to optimize. Let us consider a shallower architecture and its deeper counterpart that adds more layers onto it. There exists a solution *by construction* to the deeper model: the added layers are *identity* mapping, and the other layers are copied from the learned shallower model. The existence of this constructed solution indicates that a deeper model should produce no higher training error than its shallower counterpart. But experiments show that our current solvers on hand are unable to find solutions that are comparably good or better than the constructed solution (or unable to do so in feasible time)."

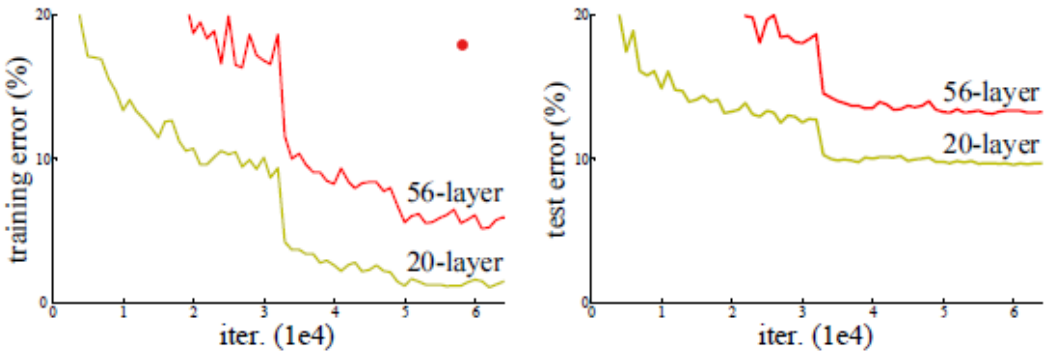

Figure 1: Deep does not mean better performance [1]

The idea in a nutshell is to learn a function of the difference between layers $\mathbf{h}_{n+1} = F(\mathbf{h}_n) + \mathbf{h}_n$. In this paper, they show this simple transformation in what you're learning allows the networks to keep improving as then add more layers.

Euler's method and residual networks

Suppose we have some constant that we'll call $\Delta t \in \mathbb{R}$. Then we can write the state update of our neurla network as

$$\begin{aligned}\mathbf{h}_{t+1} &= F(\mathbf{h}_t) + \mathbf{h}_t \\ &= \frac{\Delta t}{\Delta t} F(\mathbf{h}_t) + \mathbf{h}_t \\ &= \Delta t G(\mathbf{h}_t) + \mathbf{h}_t\end{aligned}$$

This formulation looks very familiar - it is a single step of Euler's method for solving ordinary differential equations.

Evaluating ODEs

If we consider a layer of our neurla network to be doing a step of Euler's method, then we can model our system by the differential equation

$$\frac{\mathrm{d}\mathbf{h}(t)}{\mathrm{d}t} = G(\mathbf{h}(t), t, \theta)$$

Here we've made explicit G's dependency on t, as well as some parameters θ which we will train on. In this formulation, the output of our "network" is the state $\mathbf{h}(t_1)$ at some time t_1. Therefore, if we know how to describe the function G, we can use any number of off-the-shelf ODE solvers to evaluate the neural network.

$$\mathbf{h}(t_1) = \text{ODESolve}(\mathbf{h}(t_0), G, t_0, t_1, \theta)$$

There is a ton of research on different methods than can be used as our ODESolve function, but for now we'll treat it as a black box. What matters is that if you substitute in Euler's method, you get exactly the residual state update from above, with

$$\mathbf{h}(t_1) = \text{ODESolve}(\mathbf{h}(t_0), G, t_0, t_1, \theta) = \mathbf{h}(t_0) + (t_1 - t_0) G(\mathbf{h}(t_0), t_0, \theta)$$

However, we do not need to limit ourselves to Euler's method, and in fact will do much better if we use more modern approaches.

Training

Suppose we have a loss function

$$L(\mathbf{h}(t_1)) = L(\text{ODESolve}(\mathbf{h}(t_0), G, t_0, t_1, \theta))$$

To optimize L, we require gradients with respect to its parameters

- $\mathbf{h}(t)$ (the state of our system at time t),
- t (our "time" variable, which is sort of a continuous analog to depth)
- θ, our training parameters.

The adjoint method describes a way to come up with this. The adjoint method is a neat trick which uses a simple substitution of variables to make solving certain linear systems easier. Suppose we have 2 known matrices A and C, and an unknown vector $\mathbf{u}$ and we would like to compute a product $\mathbf{u}^\top B$ such that $AB = C$. We could first solve the linear system to find the unknown matrix B, then compute the product, but solving the linear system could be expensive. Instead, let's find a vector $\mathbf{v}$ and compute $\mathbf{v}^\top C$ such that $A^\top \mathbf{v} = \mathbf{u}$. We can show that these are in fact the same problem

$$\mathbf{v}^\top C = \mathbf{v}\top AB = (A^\top v)^\top B = \mathbf{u}^\top B$$

Through this transformation, we've reduced the problem from solving a matrix, and reduced it to solving for a vector.
In brief, we define the adjoint state as

$$a(t) = -\partial L/\partial \mathbf{h}(t)$$

And we can describe its dynamics via

$$\frac{\mathrm{d}a(t)}{\mathrm{d}t} = -a(t)^\top \frac{\partial G(\mathbf{h}(t), t, \theta)}{\partial \mathbf{h}}$$

We can compute the derivative of G with respect to $\mathbf{h}$ already - we compute this gradient during backpropagation of traditional neural networks. With this, we can then compute $a(t)$ by using another call to an ODE solver. There is one other derivative, $\mathrm{d}L/\mathrm{d}\theta$, that can be computed similarly. The paper shows that we can wrap up all of these ODE solves into a single call to an ODE solver, which computes all the necessary gradients for training the system.
We're able to train a model with much less memory, with fewer parameters, and we are able to backpropagate more efficiently. The most interesting aspect is that treating our system like a continuous time model, allows us to predict continuous time systems. They show a way to take data which arrives at arbitrary times, rather than at fixed intervals, and they can predict the output at arbitrary future times.

2 My understanding

Instead of traditional DL which predicts

$$\hat{y} = F(x, \theta)$$

and minimizes $\mathcal{L} = (y_{\text{obs}} - \hat{y})^2$ for instance (loss function could definitely be different), Kaiming's paper demonstrates that DL cannot improve accuracy performance by simply stacking more layers, and this has nothing to do with overfitting. Instead, they consider deep residual learning (and RNN is also similar) in which we would consider

$$\mathbf{z}_{t+1} = \mathbf{z}_t + f(\mathbf{z}_t, \theta_t)$$

This could be seen as Euler discretization of a continuous transformation. If to take smaller steps, and rewrite the equation above

$$\begin{aligned} \text{ODE}: &\frac{\mathrm{d}z}{\mathrm{d}t} = f(\mathbf{z}_t, t, \theta) \\ \text{IC}: &\mathbf{z}_t \end{aligned}$$

in which θ is the model parameters that we could optimize, and t could be seen as the depth of layers.
We could use NN to approximate function $f(\mathbf{z}_t, t, \theta)$ instead of the traditional $\hat{y} = F(x, \theta)$. Approximate the "slope" of the original function instead of approximation of the original function itself. After the approximation of the slope, we calculate $\mathbf{z}_{t+1}$.
So for $\mathbf{z}_{t_1}$, given $\mathbf{z}_{t_0}$, we could use the ODESolver to solve the ODE to obtain $\mathbf{z}_{t_1}$.

$$\mathbf{z}_{t_1} = \mathbf{z}_{t_0} + \int_{t_0}^{t_1} f(\mathbf{z}(t), t, \theta)\mathrm{d}t = \text{ODESolve}(\mathbf{z}_{t_0}, f, t_0, t_1, \theta)$$

We are interested to minimize $\mathcal{L}$ using gradients with respect to θ (training the model and optimize the parameters).

$$\text{Find } \arg\min_{\theta} \mathcal{L}(z(t_1)) \text{ subject to} \tag{1}$$

$$F(\dot{z}(t), z(t), \theta, t) = \dot{z}(t) - f(z_t, \theta, t) = 0 \tag{2}$$

$$z(t_0) = z_{t_0}, t_0 < t_1 \tag{3}$$

- f is our neural network with parameters θ
- $z(t_0)$ is our input, $z(t_1)$ is the output, $z(t)$ is the state reached from $z(t_0)$ at time $t \in [t_0, t_1]$
- $\dot{z}(t) = \frac{\mathrm{d}z(t)}{\mathrm{d}t}$ is the time derivative of our state $z(t)$
- $\mathcal{L}$ is our loss and it is a function of the output $z(t_1)$

To tackle Eq. 1 with gradient descent, we need the gradient of the loss $\mathcal{L}(z(t_1))$ with respect to the network parameter θ. But we saw that simple backprop can be memory inefficient because of the potentially huge computation graph (e.g. $\frac{\partial f}{\partial z}$ and $\frac{\partial z}{\partial \theta}$ calculation can be expensive). We want an efficient calculation of

$$\frac{\mathrm{d}\mathcal{L}(z(t_1))}{\mathrm{d}\theta}$$

New objective using Lagrangian:

$$\psi = \mathcal{L}(z(t_1)) - \int_{t_0}^{t_1} \lambda(t) F(\dot{z}(t), z(t), \theta, t)\mathrm{d}t$$

$\lambda(t)$ is called a Lagrange mutiplier. We know that F = 0 so the additional term does not change our gradient. That is

$$\frac{\mathrm{d}\psi}{\mathrm{d}\theta} = \frac{\mathrm{d}\mathcal{L}(z(t_1))}{\mathrm{d}\theta}$$

Try to choose $\lambda(t)$ so that we can hopefully eliminate hard to compute derivatives in $\frac{\mathrm{d}\mathcal{L}(z(t_1))}{\mathrm{d}\theta}$, which then helps for minimizing $\mathcal{L}$ with respect to θ.

$$\frac{\mathrm{d}\mathcal{L}(z(t_1))}{\mathrm{d}\theta} = \frac{\mathrm{d}\psi}{\mathrm{d}\theta} = \frac{\partial \mathcal{L}}{\partial z(t_1)}\frac{\mathrm{d}z(t_1)}{\mathrm{d}\theta} - \frac{\mathrm{d}}{\mathrm{d}\theta}\left[\int_{t_0}^{t_1} \lambda F \mathrm{d}t\right]$$

We are interested in selecting λ so that we could avoid calculation of some derivatives.

$$\begin{aligned}
\int_{t_0}^{t_1} \lambda F \mathrm{d}t &= \int_{t_0}^{t_1} \lambda(\dot{z} - f)\mathrm{d}t \\
&= \lambda(t)z(t)|_{t_0}^{t_1} - \int_{t_0}^{t_1} \dot{\lambda}(t)z(t)\mathrm{d}t - \int_{t_0}^{t_1} \lambda(t) f \mathrm{d}t \\
&= \lambda(t_1)z(t_1) - \lambda(t_0)z(t_0) - \int_{t_0}^{t_1} \left[\dot{\lambda}(t)z(t) + \lambda(t)f(z(t),\theta,t)\right]\mathrm{d}t
\end{aligned}$$

Because $z(t_0)$ is input, thus $\frac{\mathrm{d}z(t_0)}{\mathrm{d}\theta} = 0$, then

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}\theta}\left[\int_{t_0}^{t_1} \lambda F \mathrm{d}t\right] &= \frac{\mathrm{d}}{\mathrm{d}\theta}\left\{\lambda(t_1)z(t_1) - \lambda(t_0)z(t_0) - \int_{t_0}^{t_1} \left[\dot{\lambda}(t)z(t) + \lambda(t)f(z(t),\theta,t)\right]\mathrm{d}t\right\} \\
&= \lambda(t_1)\frac{\mathrm{d}z(t_1)}{\mathrm{d}\theta} - \int_{t_0}^{t_1} \left[\dot{\lambda}\frac{\mathrm{d}z}{\mathrm{d}\theta} + \lambda(\frac{\partial f}{\partial z}\frac{\mathrm{d}z}{\mathrm{d}\theta} + \frac{\partial f}{\partial \theta})\right]\mathrm{d}t
\end{aligned}$$

Then

$$\begin{aligned}
\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\theta} &= \frac{\partial \mathcal{L}}{\partial z(t_1)}\frac{\mathrm{d}z(t_1)}{\mathrm{d}\theta} - \frac{\mathrm{d}}{\mathrm{d}\theta}\left[\int_{t_0}^{t_1} \lambda F \mathrm{d}t\right] \\
&= \left[\frac{\partial \mathcal{L}}{\partial z(t_1)} - \lambda(t_1)\right]\frac{\mathrm{d}z(t_1)}{\mathrm{d}\theta} + \int_{t_0}^{t_1} \left[\dot{\lambda}\frac{\mathrm{d}z}{\mathrm{d}\theta} + \lambda(\frac{\partial f}{\partial z}\frac{\mathrm{d}z}{\mathrm{d}\theta} + \frac{\partial f}{\partial \theta})\right]\mathrm{d}t \\
&= \left[\frac{\partial \mathcal{L}}{\partial z(t_1)} - \lambda(t_1)\right]\frac{\mathrm{d}z(t_1)}{\mathrm{d}\theta} + \int_{t_0}^{t_1} \left[\dot{\lambda} + \lambda\frac{\partial f}{\partial z}\right]\frac{\mathrm{d}z}{\mathrm{d}\theta}\mathrm{d}t + \int_{t_0}^{t_1} \lambda(t)\frac{\partial f}{\partial \theta}\mathrm{d}t
\end{aligned}$$

We want to get rid of $\frac{\mathrm{d}z}{\mathrm{d}\theta}$, so we could choose $\lambda(t)$ such that $\frac{\partial \mathcal{L}}{\partial z(t_1)} - \lambda(t_1) = 0$, $\dot{\lambda} + \lambda\frac{\partial f}{\partial z} = 0$, so we solve and obtain

$$\lambda(t_0) = \lambda(t_1) - \int_{t_1}^{t_0} \lambda(t)\frac{\partial f}{\partial z}\mathrm{d}t, \qquad \lambda(t_1) = \frac{\partial \mathcal{L}}{\partial z(t_1)} \tag{4}$$

Then

$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\theta} = \int_{t_0}^{t_1} \lambda(t)\frac{\partial f}{\partial \theta}\mathrm{d}t \tag{5}$$

Notice that Eq. 5 is a new ODE system that does not require us to preserve activations from the forward pass. So the new method trades off computation for memory $\mathcal{O}(1)$ with respect to the number of layers.

Summary of steps

1. Forward pass, Solve the ODE Eq. 2 and 3 from time t_0 to t_1, and get the output $z(t_1)$
2. Loss calculation: calculate $\mathcal{L}(z(t_1))$
3. Backward pass: Solve ODEs 4 and 5 from reverse time t_1 to t_0 to get the gradient of the loss $\frac{\mathrm{d}\mathcal{L}(z(t_1))}{\mathrm{d}\theta}$
4. Use the gradient to update the network parameters θ

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[2] R. W. Erickson and D. Maksimović, *Fundamentals of Power Electronics*. Springer Cham, 2020.

[3] A. Gole, A. Keri, C. Kwankpa, E. Gunther, H. Dommel, I. Hassan, J. Marti, J. Martinez, K. Fehrle, L. Tang, M. McGranaghan, O. Nayak, P. Ribeiro, R. Iravani, and R. Lasseter, "Guidelines for modeling power electronics in electric power engineering applications," *IEEE Transactions on Power Delivery*, vol. 12, no. 1, pp. 505–514, 1997.