Paper Reading

Neural Ordinary Differential Equations (NeurIPS 18)
Add a shortcut to overcome numerical issues to solve backprop

Make network deep enough to model complicated relation
RNN Decoder

\[ h_{t+1} = f(h_t) + h_t \]
ODE Recap

- First Order Initial Value Problem:

  \[
  \frac{dy}{dx} = f(x, y) \\
  y(x_0) = y_0
  \]

- f is y’s dynamics

- To solve \(y(x = x_t)\)
Euler Method

The problem [edit]

A first-order differential equation is an initial value problem (IVP) of the form,[1]

\[ y'(t) = f(t, y(t)), \quad y(t_0) = y_0, \]  

where \( f \) is a function that maps \((t, y) \mapsto \mathbb{R}^d\) and the initial condition \( y_0 \in \mathbb{R}^d \) is a given vector. First-order means that only the first derivative of \( y \) appears in the equation, and higher derivatives are absent.

Without loss of generality to higher-order systems, we restrict ourselves to first-order differential equations, because a higher-order ODE can be converted into a larger system of first-order equations by introducing extra variables. For example, the second-order equation \( y'' = -y \) can be rewritten as two first-order equations: \( y' = z \) and \( z' = -y \).

In this section, we describe numerical methods for IVPs, and remark that boundary value problems (BVPs) require a different set of tools. In a BVP, one defines values, or components of the solution \( y \) at more than one point. Because of this, different methods need to be used to solve BVPs. For example, the shooting method (and its variants) or global methods like finite differences, Galerkin methods, or collocation methods are appropriate for that class of problems.

The Picard–Lindelöf theorem states that there is a unique solution, provided \( f \) is Lipschitz-continuous.

Methods [edit]

Numerical methods for solving first-order IVPs often fall into one of two large categories: linear multistep methods, or Runge-Kutta methods. A further division can be realized by dividing methods into those that are explicit and those that are implicit. For example, implicit linear multistep methods include Adams-Moulton methods, and backward differentiation methods (BDF), whereas explicit Runge-Kutta methods[2] include diagonally implicit Runge-Kutta (DIRK), singly diagonally implicit runge kutta (SDIRK), and Gauss-Radau (based on Gaussian quadrature) numerical methods. Explicit examples from the linear multistep family include the Adams-Bashforth methods, and any Runge-Kutta method with a lower diagonal Butcher tableau is explicit. A loose rule of thumb dictates that stiff differential equations require the use of implicit schemes, whereas non-stiff problems can be solved more efficiently with explicit schemes.

The so-called general linear methods (GLMs) are a generalization of the above two large classes of methods.

Euler method [edit]

Further information: Euler method

From any point on a curve, you can find an approximation of a nearby point on the curve by moving a short distance along a line tangent to the curve.

Starting with the differential equation (1), we replace the derivative \( y' \) by the finite difference approximation

\[ y'(t) \approx \frac{y(t + h) - y(t)}{h}, \]  

which when re-arranged yields the following formula

\[ y(t + h) \approx y(t) + hy'(t) \]

and using (1) gives:

\[ y(t + h) \approx y(t) + hf(t, y(t)). \]  

This formula is usually applied in the following way. We choose a step size \( h \), and we construct the sequence \( t_0, t_1 = t_0 + h, t_2 = t_0 + 2h, \ldots \). We denote by \( y_n \) a numerical estimate of the exact solution \( y(\delta) \). Motivated by (3), we compute these estimates by the following recursive scheme

\[ y_{n+1} = y_n + hf(t_n, y_n). \]  

This is the Euler method (or forward Euler method, in contrast with the backward Euler method, to be described below). The method is named after Leonhard Euler who described it in 1768.

The Euler method is an example of an explicit method. This means that the new value \( y_{n+1} \) is defined in terms of things that are already known, like \( y_n \).
The Runge–Kutta method

The most widely known member of the Runge–Kutta family is generally referred to as "RK4", "classical Runge–Kutta method" or simply as "the Runge–Kutta method".

Let an initial value problem be specified as follows:

\[ \dot{y} = f(t, y), \quad y(t_0) = y_0. \]

Here \( y \) is an unknown function (scalar or vector) of time \( t \), which we would like to approximate; we are told that \( \dot{y} \), the rate at which \( y \) changes, is a function of \( t \) and of \( y \) itself. At the initial time \( t_0 \), the corresponding \( y \) value is \( y_0 \). The function \( f \) and the data \( t_0, y_0 \) are given.

Now pick a step-size \( h > 0 \) and define

\[ y_{n+1} = y_n + \frac{h}{6} \left( k_1 + 2k_2 + 2k_3 + k_4 \right), \]
\[ t_{n+1} = t_n + h \]

for \( n = 0, 1, 2, 3, \ldots \), using\(^\[2\]

\[ k_1 = h f(t_n, y_n), \]
\[ k_2 = h f \left( t_n + \frac{h}{2}, y_n + \frac{k_1}{2} \right), \]
\[ k_3 = h f \left( t_n + \frac{h}{2}, y_n + \frac{k_2}{2} \right), \]
\[ k_4 = h f \left( t_n + h, y_n + k_3 \right). \]

(Note: the above equations have different but equivalent definitions in different texts\(^[3]\)).

Here \( y_{n+1} \) is the RK4 approximation of \( y(t_{n+1}) \), and the next value \( (y_{n+1}) \) is determined by the present value \( (y_n) \) plus the weighted average of four increments, where each increment is the product of the size of the interval, \( h \), and an estimated slope specified function \( f \) on the right-hand side of the differential equation.

- \( k_1 \) is the increment based on the slope at the beginning of the interval, using \( y \) (Euler's method);
- \( k_2 \) is the increment based on the slope at the midpoint of the interval, using \( y \) and \( k_1 \);
- \( k_3 \) is again the increment based on the slope at the midpoint, but now using \( y \) and \( k_2 \);
- \( k_4 \) is the increment based on the slope at the end of the interval, using \( y \) and \( k_3 \).

In averaging the four increments, greater weight is given to the increments at the midpoint. If \( f \) is independent of \( y \), so that the differential equation is equivalent to a simple integral, then RK4 is Simpson's rule\(^[4]\).

The RK4 method is a fourth-order method, meaning that the local truncation error is on the order of \( O(h^5) \), while the total accumulated error is on the order of \( O(h^4) \).
Intuition

\[ h_{t+1} = f(h_t) + h_t \]

\[ \frac{dh(t)}{dt} = f(h(t), t, \theta_t) \]

- A lot of modern deep networks’ forwarding pass is like euler/rk4 to solve IVP
- The layer definition is like a discretization of a dynamics
- Why not thinking in the limit world and model the continuous dynamics of hidden units?
- Dynamics f can be specified by a neural network
The Idea

- Forward step is equal to solving a IVP using a ODESolver

\[
\frac{dh(t)}{dt} = f(h(t), t)
\]

\[
h(t_0) = x
\]
The Idea

• What about backward step: calculate gradient?

• Operator based (chain rule)?

• Some math solution?
Potential Benefits

- Memory Efficiency
- Adaptive Computation
- Parameter Efficiency
- Scalable and invertible normalizing flow
- Continuous time-series models
Memory Efficiency

- **Traditional Backprop**
  
  - I. Forward step should save state $z$ for gradient computation
  
  - II. Backward step is operator based (chain rule)

  - I. is memory costly, II is numerical unstable (dropout, etc)

Summary: the equations of backpropagation

1. $\delta^L = \nabla_a C \circ \sigma'(z^L)$ (BP1)
2. $\delta^l = ((w^{l+1})^T \delta^{l+1}) \circ \sigma'(z^l)$ (BP2)
3. $\frac{\partial C}{\partial b^l_j} = \delta^l_j$ (BP3)
4. $\frac{\partial C}{\partial w^{l-1}_{jk}} = a^{l-1}_k \delta^l_j$ (BP4)
How to Calculate Gradient

Ultimately want to optimize some loss

\[
L(z(t_1)) = L\left(\int_{t_0}^{t_1} f(z(t), t, \theta) dt\right)
\]

\[
= L(\text{ODESolve}(z(t_0), f, t_0, t_1, \theta))
\]
Reverse-time autodiff

• Define adjoint: \[ a(t) = -\frac{\partial L}{\partial z(t)} \]

• Which has dynamics: \[ \frac{da(t)}{dt} = -a(t)^T \frac{\partial f(z(t), t, \theta)}{\partial z} \]

• Start adjoint with \[ \frac{\partial L}{\partial z(t_1)} \]

• And solve a combined ODE backwards in time:

\[
\frac{dL}{d\theta} = \int_{t_1}^{t_0} a(t)^T \frac{\partial f(z(t), t, \theta)}{\partial \theta} dt
\]

[Scalable Inference of Ordinary Differential Equation Models of Biochemical Processes”, Froehlich, Loos, Hasenauer, 2017]
Reverse-time autodiff

- In english: Solve the original ODE and the accumulated gradients backwards through time.
Can ask for multiple measurement times

Reverse pass breaks solution into N-1 chunks
Memory Efficiency

- Traditional Backprop
  - I. Forward step should save state $z$ for gradient computation
  - II. Backward step is operator based (chain rule)
  - I. is memory costly, II is numerical unstable (dropout, etc)

- ODENet
  - $O(1)$ memory cost: don’t need to store layer activations for reverse pass (just follow the dynamics in reverse since it’s reversible)
  - numerical more stable (numerical methods from math)
Adaptive Computation

• Old ODE Solvers
  • Euler’s method (1870)
  • Runge-Kutta (1901)
  • Hairer (1987)

• Modern ODE Solvers
  • Achieve specified approximation error
  • Adaptive steps to minimize effort
Adaptive Computation

- Human defined network architecture
  - Similar to define artificial steps in RK4/Euler
- Continuous Dynamics
  - Smart
  - No wasted layers
Fewer Parameters

- Neighboring layers automatically similar
- Dynamics space is lower than the original space
  - Define dynamics structure needs fewer parameters
Continuous time-series models

- RNNS: need to discretize time, only classify/predict/generate data in the future step (fix time interval)
- Classify/predict/generate data which arrives at arbitrary times
ODENet Method

• Define the continuous dynamics of the hidden states (natural reversible)

• For each training sample, train theta using SGD/Adam/Momentum/AdaGrad, to calculate gradients:
  • ODESolver round to solve forward pass
  • Another ODESolver round to compute gradients
  • Apply gradient update

• Theta is bundled with the optimization objective function
ODENet Method

• How to define the dynamics of the hidden state?
  • Also a NN which may have good generality
    • e.g. ResBlock
  • Time dependent vs time independent?
    • Tricky
Applications

- Supervised Learning
- Generative latent function time-series model
- Poisson Process Likelihood
- Continuous Normalizing Flows
Supervised Learning
Table 1: Performance on MNIST. †From [23].

<table>
<thead>
<tr>
<th>Model</th>
<th>Test Error</th>
<th># Params</th>
<th>Memory</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-Layer MLP†</td>
<td>1.60%</td>
<td>0.24 M</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ResNet</td>
<td>0.41%</td>
<td>0.60 M</td>
<td>$\mathcal{O}(L)$</td>
<td>$\mathcal{O}(L)$</td>
</tr>
<tr>
<td>RK-Net</td>
<td>0.47%</td>
<td>0.22 M</td>
<td>$\mathcal{O}(\tilde{L})$</td>
<td>$\mathcal{O}(\tilde{L})$</td>
</tr>
<tr>
<td>ODE-Net</td>
<td>0.42%</td>
<td>0.22 M</td>
<td>$\mathcal{O}(1)$</td>
<td>$\mathcal{O}(\tilde{L})$</td>
</tr>
</tbody>
</table>
Explicit Error Control

- More fine-grained control than low-precision floats
- Cost scales with instance difficulty
Speed-Accuracy Tradeoff

- Time cost is dominated by evaluation of dynamics
- Roughly linear with number of forward evaluations
Reverse vs Forward Cost

- Empirically, reverse pass roughly half as expensive as forward pass
- Again, adapts to instance difficulty
- Num evaluations comparable to number of layers in modern nets
How complex are the dynamics?

- Dynamics become more demanding to compute during training
Generative Time-series Model
The Problem

- Given observation times $t_0, t_1, \ldots, t_N$, generate new data $t_M$ in the observation space.
• Given observation times $t_0, t_1, \ldots, t_N$, and the initial state $Z_{t_0}$

• ODE Solver produces $Z_{t_1}, Z_{t_2}, \ldots, Z_{t_N}$ and generate data through a sampling procedure as follows:

$$z_{t_0} \sim p(z_{t_0})$$

$$z_{t_1}, z_{t_2}, \ldots, z_{t_N} = \text{ODESolve}(z_{t_0}, f, \theta_f, t_0, \ldots, t_N)$$

each $x_{t_i} \sim p(x|z_{t_i}, \theta_x)$
To obtain the latent representation $z_{t_0}$, we traverse the sequence using RNN and obtain parameters of distribution $q(z_{t_0}|\{x_{t_i}, t_i\}_i, \theta_{enc})$. The algorithm follows a standard VAE algorithm with an RNN variational posterior and an ODESolve model:

1. Run an RNN encoder through the time series and infer the parameters for a posterior over $z_{t_0}$:

$$q(z_{t_0}|\{x_{t_i}, t_i\}_i, \phi) = \mathcal{N}(z_{t_0}|\mu_{z_{t_0}}, \sigma_{z_0}),$$  \hspace{1cm} (53)

where $\mu_{z_0}, \sigma_{z_0}$ comes from hidden state of RNN($\{x_{t_i}, t_i\}_i, \phi$)

2. Sample $z_{t_0} \sim q(z_{t_0}|\{x_{t_i}, t_i\}_i)$

3. Obtain $z_{t_1}, z_{t_2}, \ldots, z_{t_M}$ by solving ODE ODESolve$(z_{t_0}, f, \theta_f, t_0, \ldots, t_M)$, where $f$ is the function defining the gradient $dz/dt$ as a function of $z$

4. Maximize ELBO $= \sum_{i=1}^{M} \log p(x_{t_i}|z_{t_i}, \theta_x) + \log p(z_{t_0}) - \log q(z_{t_0}|\{x_{t_i}, t_i\}_i, \phi)$, where $p(z_{t_0}) = \mathcal{N}(0, 1)$
Experiment

- 1000 2d spirals each starting at a different point
- Extrapolate hidden state and then generate data at $t_M$

Baseline Model
- VAE like model
  - Recognition network is RNN with 25 hidden units (encoder)
  - decoder network with 25 hidden units

Latent ODE Model
- Recognition network is RNN with 25 hidden units (encoder)
- Z is 4d latent space
- $p(x_t | z_t)$ is another 20 hidden units one layer NN
- Dynamics f is one hidden layer network with 20 hidden units
Table 2: Predictive RMSE on test set

<table>
<thead>
<tr>
<th># Observations</th>
<th>30/100</th>
<th>50/100</th>
<th>100/100</th>
</tr>
</thead>
<tbody>
<tr>
<td>RNN</td>
<td>0.3937</td>
<td>0.3202</td>
<td>0.1813</td>
</tr>
<tr>
<td>Latent ODE</td>
<td><strong>0.1642</strong></td>
<td><strong>0.1502</strong></td>
<td><strong>0.1346</strong></td>
</tr>
</tbody>
</table>

Figure 9: Data-space trajectories decoded from varying one dimension of $z_{t_0}$. Color indicates progression through time, starting at purple and ending at red. Note that the trajectories on the left are counter-clockwise, while the trajectories on the right are clockwise.
Figure 10: Spiral reconstructions using a latent ODE with a variable number of noisy observations.
Poisson Process Likelihoods

- Can condition on observation times
- Define rate function as a function of latent state
- Poisson likelihood is just another integral, can be solved along with latent state

\[
\log p(t_1, \ldots, t_N | t_{\text{start}}, t_{\text{end}}) = \sum_{i=1}^{N} \log \lambda(z(t_i)) - \int_{t_{\text{start}}}^{t_{\text{end}}} \lambda(z(t)) \, dt
\]
Normalizing Flows

\[ x_1 = f(x_0) \implies p(x_1) = p(x_0) \left| \det \frac{\partial f}{\partial x_0} \right|^{-1} \]

- Determinant of Jacobian has cost \( O(D^3) \).
- Matrix determinant lemma gives \( O(DH^3) \) cost.
- Normalizing flows use 1 hidden unit. Deep & skinny

\[ x(t + 1) = x(t) + uh(w^T x(t) + b) \]
\[ \log p(x(t + 1)) = \log p(x(t)) - \log \left| 1 + u^T \frac{\partial h}{\partial x} \right| \]
Paper Revisit

Code Revisit

Thank You!