Diffusion for Imitation Learning

Devesh

January 4, 2025

Abstract

My notes for diffusion for imitation learning.

1 Neural Ordinary Differential Equations

Scalar Loss Function: $L(z(t_1)) = L\left(z(t_0) + \int_{t_0}^{t_1} f(z(t), t, \theta)\, dt\right) = L(\text{ODESolve}(z(t_0), f, t_0, t_1, \theta))$
State Dynamics: $\frac{dz(t)}{dt} = f(z(t), t, \theta)$
Lagrangian: $L(u, p, z, \lambda) = J(u, p) + \int_{t_0}^{t_1} a(t)^\top \left(\frac{dz(t)}{dt} - f(z(t), u, p)\right) dt$, for some cost J
Adjoint Dynamics: $a(t_N) = \frac{\partial L}{\partial z(t_N)}$, $\frac{da(t)}{dt} = -a(t)^\top \frac{\partial f(z(t),t,\theta)}{\partial z(t)}$
Parameter Gradient: $\frac{\partial L}{\partial \theta} = -\int_{t_0}^{t_N} a(t)^\top \frac{\partial f(z(t),t,\theta)}{\partial \theta}\, dt$

1.1 Comparison with Resnets

Resnets are a special case of Neural ODEs, where euler integration is used to model dynamical systems $x_{t+1} = x_t + f(x_t, u)$. Numerical integration requires to store the states of the system in memory and doesnt perform well for long sequences. Resnets also require data to be regularly spaced in time which is often constraining. Neural ODEs on the other hand, only require to store the initial state and the parameters of the ODE, and can be trained using reverse-mode differentiation.

1.2 Reverse-Mode Differentiation or Backpropagation

Neural ODEs model the dynamics of the continuous function directly. We have $\{t_0 ... t_N\}$ points to equally spaced points to discretize calculation of the Neural ODE. Data can be irregulary spaced and can occur anywhere in this interval. At t_N, $a(t_N) = \frac{\partial L}{\partial z(t_N)}$ is calculated. Then using $\frac{da(t)}{dt} = -a(t)^\top \frac{\partial f(z(t),t,\theta)}{\partial z(t)}$ the adjoint state is propagated back in time. To calculate the adjoint state, z(t) is required but unlike Resnets, $z(t)$ is not stored but is calculated by solving the ODE $\frac{dz(t)}{dt} = f(z(t), t, \theta)$ backwards in time until we reach a data point $z(t_i)$. At the data point, $a(t_i)$ is calculated similarly to $a(T_N)$. There is a discrepancy (shown in figure 2) between the propagated adjoint dynamics and the dynamics at this data point, which is minimzed to get continuous adjoint state dynamics. When we have the entire adjoint dynamics, we can calculate the gradient of the parameters θ using $\frac{\partial L}{\partial \theta} = -\int_{t_0}^{t_N} a(t)^\top \frac{\partial f(z(t),t,\theta)}{\partial \theta}\, dt$.

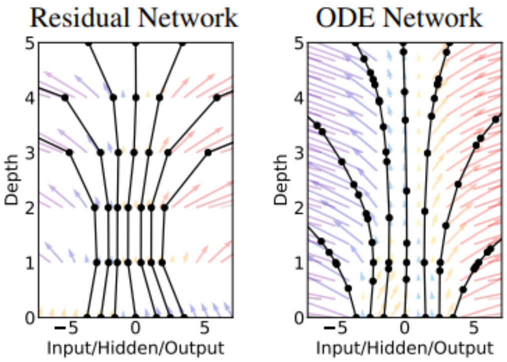

Figure 1: Resnets vs Neural ODEs

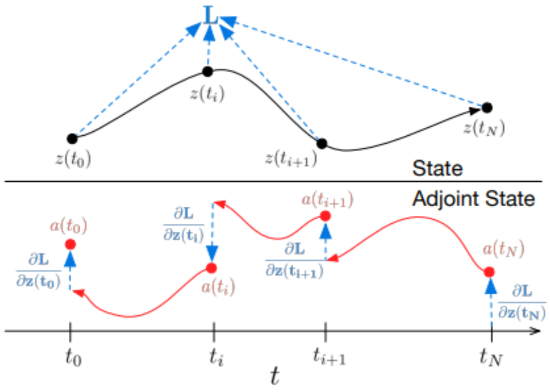
L
$z(t_0)$
$z(t_i)$
$z(t_{i+1})$
$z(t_N)$
State
Adjoint State
$a(t_0)$
$\frac{\partial L}{\partial z(t_0)}$
$\frac{\partial L}{\partial z(t_i)}$
$a(t_i)$
$a(t_{i+1})$
$\frac{\partial L}{\partial z(t_{i+1})}$
$a(t_N)$
$\frac{\partial L}{\partial z(t_N)}$
t_0
t_i
t_{i+1}
t_N
t

Figure 2: Adjoint and State Dynamics, Reverse-Mode Differentiation

1.3 Normalizing Flows

Normalizing flows use a sequence of transformations $f_1, f_2, \ldots, f_K$, where each transformation is invertible and differentiable, to map samples from a simple base distribution $p_X(x)$ to a target distribution $p_Z(z)$ (e.g., standard Gaussian). In Normalizing Flows, the transformation f is a learned bijective function:

$$z_1 = f(z_0) \Rightarrow \log p(z_1) = \log p(z_0) - \log \left| \det \frac{\partial f}{\partial z_0} \right|$$

Calculating the determinant is computationally expensive. Instead, continuous normalizinf flows are used. Planar Flow:

$$f(z(t+1)) = z(t) + uh(w^\top z(t) + b)$$

CNFs generalize normalizing flows by moving from discrete transformations (layers) to continuous transformations using differential equations. A CNF is defined by a time-dependent differential equation:

$$\frac{dz(t)}{dt} = f(z(t), t),$$

where f is a neural network parameterized by learnable weights.
The probability density evolves according to the instantaneous change of variables theorem:

$$\frac{\partial \log p(z(t))}{\partial t} = -\mathrm{tr}\left(\frac{\partial f}{\partial z(t)} \right),$$

where $\mathrm{tr}(\cdot)$ is the trace of the Jacobian matrix $\frac{\partial f}{\partial z(t)}$. In CNFs, the trace of the Jacobian $\mathrm{tr}\left(\frac{\partial f}{\partial z}\right)$ allows wide layers (many hidden units) to be evaluated efficiently, scaling linearly with the number of hidden units M. If the dynamics are defined as a sum of multiple functions:

$$\frac{dz(t)}{dt} = \sum_{n=1}^{M} f_n(z(t)),$$

the log-density evolves as:

$$\frac{\partial \log p(z(t))}{\partial t} = \sum_{n=1}^{M} \mathrm{tr}\left(\frac{\partial f_n}{\partial z}\right).$$

This decomposition makes the model highly expressive while keeping computations efficient. The dynamics $f(z(t), t)$ can depend explicitly on time t, allowing the transformation to vary continuously over time. Gating mechanisms $\sigma_n(t) \in (0, 1)$ control when specific dynamics $f_n(z)$ are active:

$$\frac{dz}{dt} = \sum_{n} \sigma_n(t) f_n(z).$$

This makes the model more interpretable and adaptable to the structure of the data. In both NFs and CNFs, hidden units correspond to neurons (i.e., individual computational units in the parameterizing neural network). The layers in discrete NFs correspond to discrete transformations, whereas in CNFs, the transformation is continuous, so there are no explicit ”layers”—only a time-evolving flow parameterized by the hidden units.