neuralODE

December 16, 2021

1 Data Loading and Preprocessing

```
[ ]: import numpy as np
     import matplotlib.pyplot as plt
     import MDAnalysis as mda
     import torch
     import torchsde
     import torchdiffeq

     from neuralSDE import NeuralSDE
     from utils import plot
```

2 Review of Typical Neural Network Framework

In any supervised or semi-supervised setting, we are given a set of inputs and outputs $\{(x_i, y_i)\}_i^n$. The goal is learn a function $F : x \to y$ that maps each input x_i to its corresponding output y_i. Typically, f is modeled as a neural network f_θ where the parameters θ are learned through stochastic gradient descent with respect to a user-defined loss function $\mathcal{L}(x_i, y_i)$ (e.g. squared loss).

3 Neural Ordinary Differential Equations (Neural ODEs)

Chen et al. showed that instead of modeling $F : x \to y$ as a neural network, we can reformulate the problem in a continuous setting. Namely, let us call $h(t)$ the **hidden state** which depends on t (t can but does not have to represent time). The initial hidden state at some time t_0 will be $h(t_0) = x$. The final hidden state at some time t_1 after t_0 will be $h(t_1) = y$.

The derivative of $h(t)$ with respect to t is modeled as a neural network f which receives $h(t)$ and t as input and is parameterized by parameters θ. Notice that the values of the initial time t_0 and final time t_1 are not well-defined. Like the parameters of the neural network θ, t_0 and t_1 are free parameters and are likewise optimized through gradient descent with respect to the loss $\mathcal{L}$.

In summary, we are learning a function $f(t, h(t), \theta)$ such that

$$f(t, h(t), \theta) = \frac{dh(t)}{dt}$$

$$h(t_0) = x$$

$$h(t_1) = y$$

Our prediction $\hat{y}$ is thus

$$\widehat{y} = h(t_1) = ODESolve\big(h(t_0), t_0, t_1, \theta, f\big)$$

where $ODESolve$ is any numerical ODE solver (e.g. Runge-Kutta) which solves for the hidden state at t_1, $h(t_1)$. To optimize, the free parameters t_0, t_1, and θ, the adjoint sensitivity method as described by Chen et al. is first used to find the gradients $\frac{d\mathcal{L}}{dh(t)}$ and $\frac{d\mathcal{L}}{d\theta}$. The free parameters are then optimized using gradient descent.

3.1 Advantages of Neural ODEs

There are several advantage of Neural ODEs over normal neural networks.

First, Neural ODEs ensure that **the learned mapping $F : x \to y$ is smooth**.

Second, the Neural ODE formulation is **well-suited for time series data**. For instance, in the framework given above, the only hidden states that had any meaning for our problem were $h(t_0$ which was our input x and $h(t_1)$ which was our output y. All other hidden states in between t_0 and t_1 did carry any meaning for our problem. Imagine now that we are dealing with time-series data $x_0, x_1, x_2, ...x_n)$ and our goal is to either interpolate in between two given data points or predict new data points. Now each hidden state has meaning $h(t) = x_t$.

Third, we can **take advantage of the rich differential equations theory and numerical solvers** developed over the past 200 years.

4 Neural ODE Demo

In the demo below, we will attempt to use a Neural ODE to fit time-series data from a molecular simulation of alanine dipeptide.

[ ]:
```
# Neural ODE
```

5 Limitations of Neural ODEs

One issue is that normal ODEs struggle with modeling functions that sudden jumps or spikes. Some dynamical systems found in nature may also be noisy (i.e. molecular dynamics, EEG, ...). Stochastic differential equations insert an additive noise term to the ordinary differential equation to model the noise in a system. Neural ODEs can be extended to Neural SDEs by modeling the noise term as a neural network.

5.1 Ito Stochastic Differential Equations

Formally, assume that we have a stochastic process $\{Z_T\}_{t\in\mathbb{T}}$ with Gaussian noise. We can represent this stochastic with a Ito stochastic differential equation. Ito SDEs are all of the form

$$Z_T = z_0 + \int_0^T b(Z_t, t)dt + \sum_{i=1}^{m} \int_0^T \sigma(Z_t, t)dW_t$$

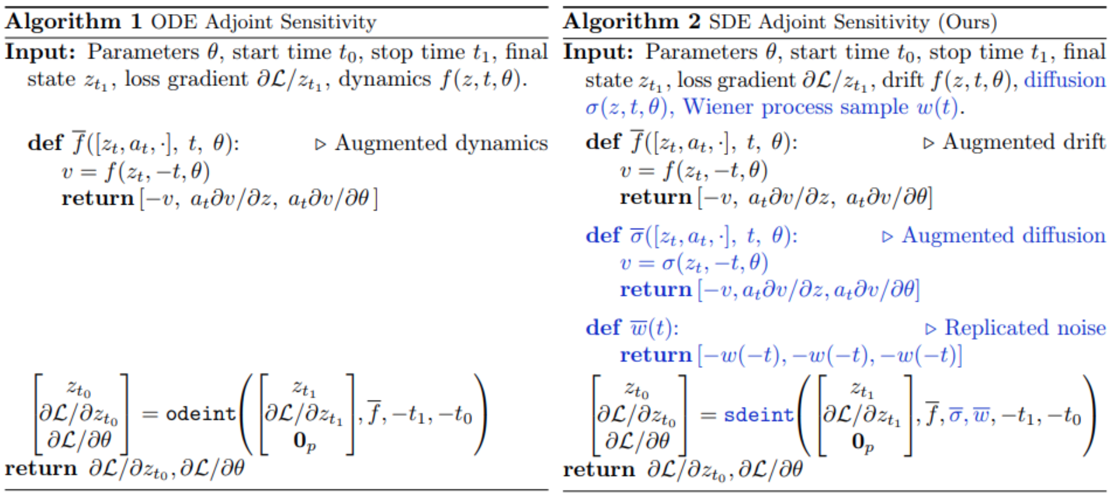
- z_0 is the initial state.
- W_t is a Wiener Process (or Brownian Motion)
- $b : \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}^d$ is called the **drift function**.
- $\sigma_i : \mathbb{R}^d \times \mathbb{R} \to \mathbb{R}^d$ is called a **diffusion function** and represents the noise in our system (for simplicity m is often set to 1 so there is only one diffusion function σ)

5.2 Neural SDEs

In the neural SDE formulation, we will model the drift function b as a neural network $f(z, t, \theta)$ and the diffusion functio σ as another neural network $\sigma(z, t, \theta)$. Li et al. address **three problems for neural SDEs**.

1. **Defining the Adjoint method for SDEs**
2. **Querying the Wiener Process W during the backward pass***
3. **Formulating a loss function for SDEs**

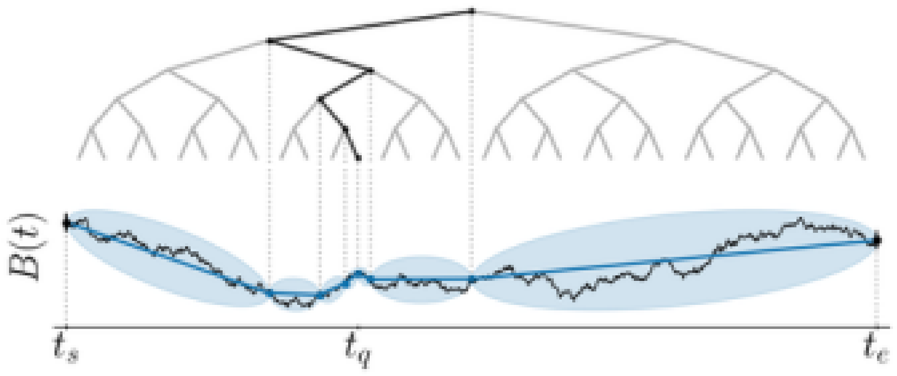
Recently, Li et al. extended the adjoint sensitivity method for ODEs to SDEs. Therefore, the gradients can be computed by solving a Stratonovich SDE that runs backward in time, and intermediate gradients do not have to be stored. The proof that a backward SDE exists can be found in the Appendix of Li et al. The pseudocode comparison of Neural ODEs and Neural SDEs is shown below. Notice that the adjoint method is similar to the adjoint method for ODEs. However, **we must compute the adjoints of two functions: the drift and the diffusion functions** and use an **SDE solver rather than an ODE Solver**.



```
Algorithm 1 ODE Adjoint Sensitivity
Input: Parameters $\theta$, start time $t_0$, stop time $t_1$, final
  state $z_{t_1}$, loss gradient $\partial\mathcal{L}/z_{t_1}$, dynamics $f(z,t,\theta)$.

  def $\overline{f}([z_t, a_t, \cdot],\ t,\ \theta)$:          ▷ Augmented dynamics
      $v = f(z_t, -t, \theta)$
      return $[-v,\ a_t\partial v/\partial z,\ a_t\partial v/\partial\theta]$

$\begin{bmatrix} z_{t_0} \\ \partial\mathcal{L}/\partial z_{t_0} \\ \partial\mathcal{L}/\partial\theta \end{bmatrix} = \texttt{odeint}\left(\begin{bmatrix} z_{t_1} \\ \partial\mathcal{L}/\partial z_{t_1} \\ \mathbf{0}_p \end{bmatrix}, \overline{f}, -t_1, -t_0\right)$
return $\partial\mathcal{L}/\partial z_{t_0}, \partial\mathcal{L}/\partial\theta$
```

```
Algorithm 2 SDE Adjoint Sensitivity (Ours)
Input: Parameters $\theta$, start time $t_0$, stop time $t_1$, final
  state $z_{t_1}$, loss gradient $\partial\mathcal{L}/z_{t_1}$, drift $f(z,t,\theta)$, diffusion
  $\sigma(z,t,\theta)$, Wiener process sample $w(t)$.

  def $\overline{f}([z_t, a_t, \cdot],\ t,\ \theta)$:          ▷ Augmented drift
      $v = f(z_t, -t, \theta)$
      return $[-v,\ a_t\partial v/\partial z,\ a_t\partial v/\partial\theta]$

  def $\overline{\sigma}([z_t, a_t, \cdot],\ t,\ \theta)$:          ▷ Augmented diffusion
      $v = \sigma(z_t, -t, \theta)$
      return $[-v, a_t\partial v/\partial z, a_t\partial v/\partial\theta]$

  def $\overline{w}(t)$:          ▷ Replicated noise
      return $[-w(-t), -w(-t), -w(-t)]$

$\begin{bmatrix} z_{t_0} \\ \partial\mathcal{L}/\partial z_{t_0} \\ \partial\mathcal{L}/\partial\theta \end{bmatrix} = \texttt{sdeint}\left(\begin{bmatrix} z_{t_1} \\ \partial\mathcal{L}/\partial z_{t_1} \\ \mathbf{0}_p \end{bmatrix}, \overline{f}, \overline{\sigma}, \overline{w}, -t_1, -t_0\right)$
return $\partial\mathcal{L}/\partial z_{t_0}, \partial\mathcal{L}/\partial\theta$
```

Notice in the pseudocode above, we need some way to query the Wiener process $w(t)$ for different values of t while we are performing the backward pass. The authors accomplish this with a **Brownian Tree** in logarithmic time. The Brownian tree allows you to recursively query for the value of Wiener process at time t without having to store these noise values in the forward pass.



Another limitation of Neural SDEs is that maxmizing the likelihood will cause the diffusion function to go to 0. Li et al. propose using variational inference to optimize Neural SDEs. Namely, a prior over functions and an approximate posterior are each parameterized using an SDE.

Prior: $dz_p = f_\theta(z(t))\ dt + \sigma_\theta(z(t))\ dW(t)$

Approximate Posterior: $dz_q = f_\phi(z(t))\ dt + \sigma_\theta(z(t))\ dW'(t)$

Note that both the prior and approximate posterior must have the same diffusion function. We can then optimize the evidenced lower bound (ELBO) as derived below. The first term is the KL term and the second term is the log likelihood.

$$\mathcal{L} = \mathbb{E}\left[\frac{1}{2}\int_0^T |u(Z_t,t)|^2 dt + \sum_{i=1}^{N} logp(y_{t_i}|z_{t_i})\right]$$

where u is

$$u(Z_t,t) = |\frac{f_\theta(z(t)) - f_\phi(z(t))}{\sigma_\theta(z(t))}|_2^2$$

5.3 Limitations of Current Neural SDE Methods

- SDEs with non-diagonal noise are very costly to compute. In Li et al., they restrict themselves to SDEs with diagonal noise.
- Maximum Likelihood maximization results in overfitting and diffusion function σ goes to 0

6 Neural SDE Examples

6.1 Plotting the mean and noise separately

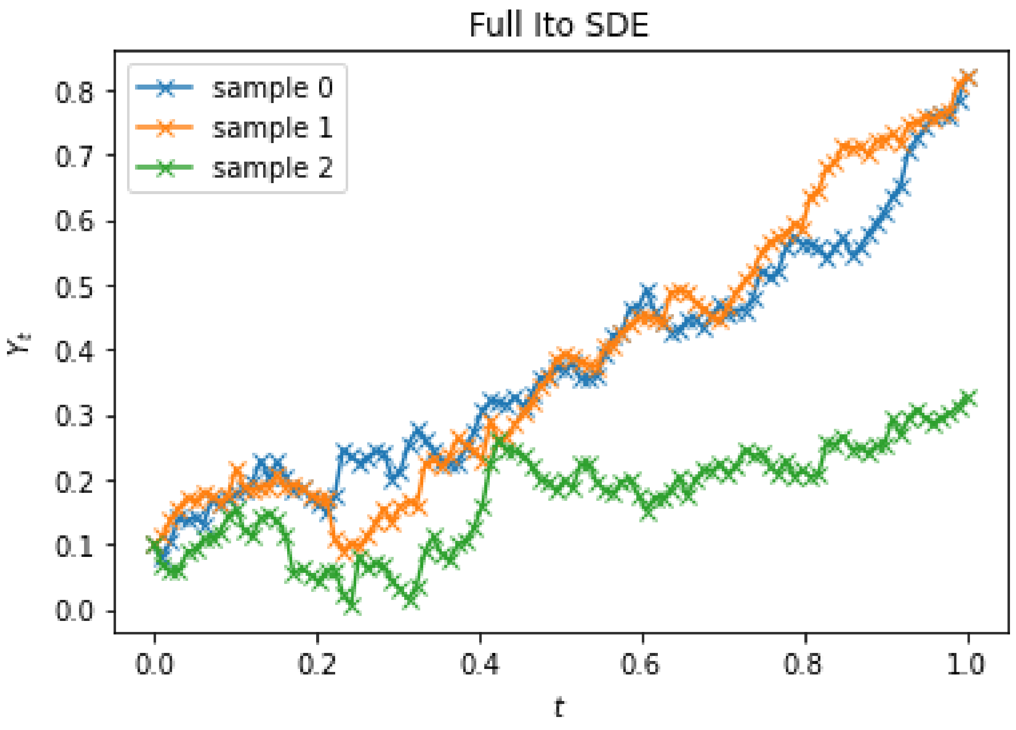
For simplicity, let us consider an SDE with the drift and diffusion, f and g, defined below

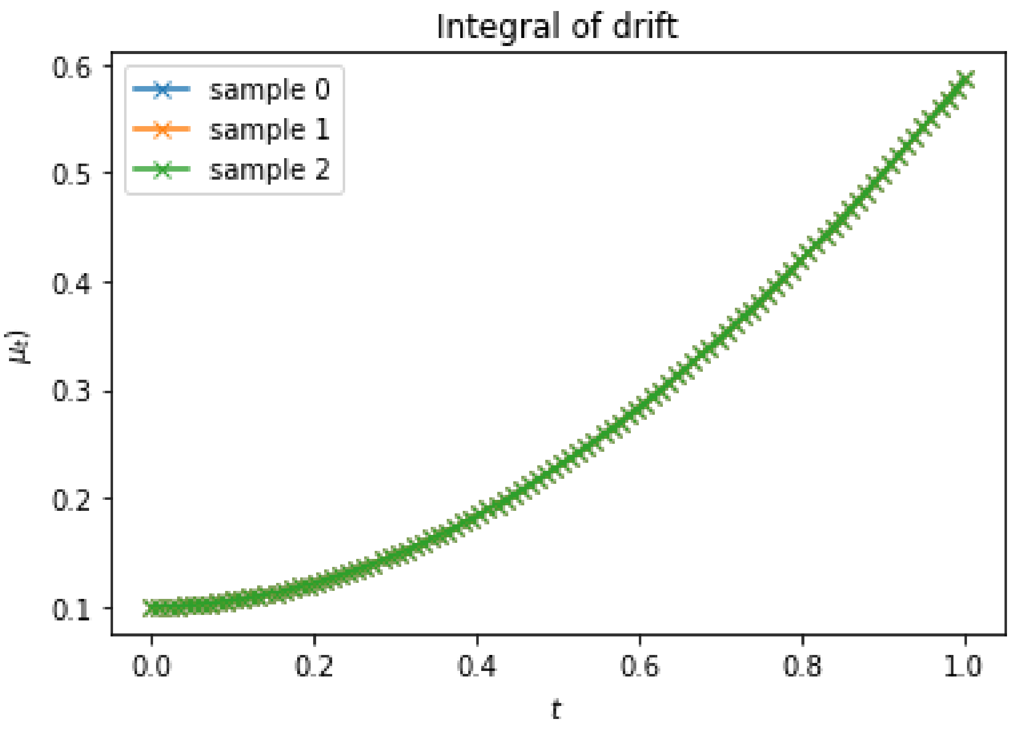
$$f(t,y) = sin(t) + \theta \cdot y$$

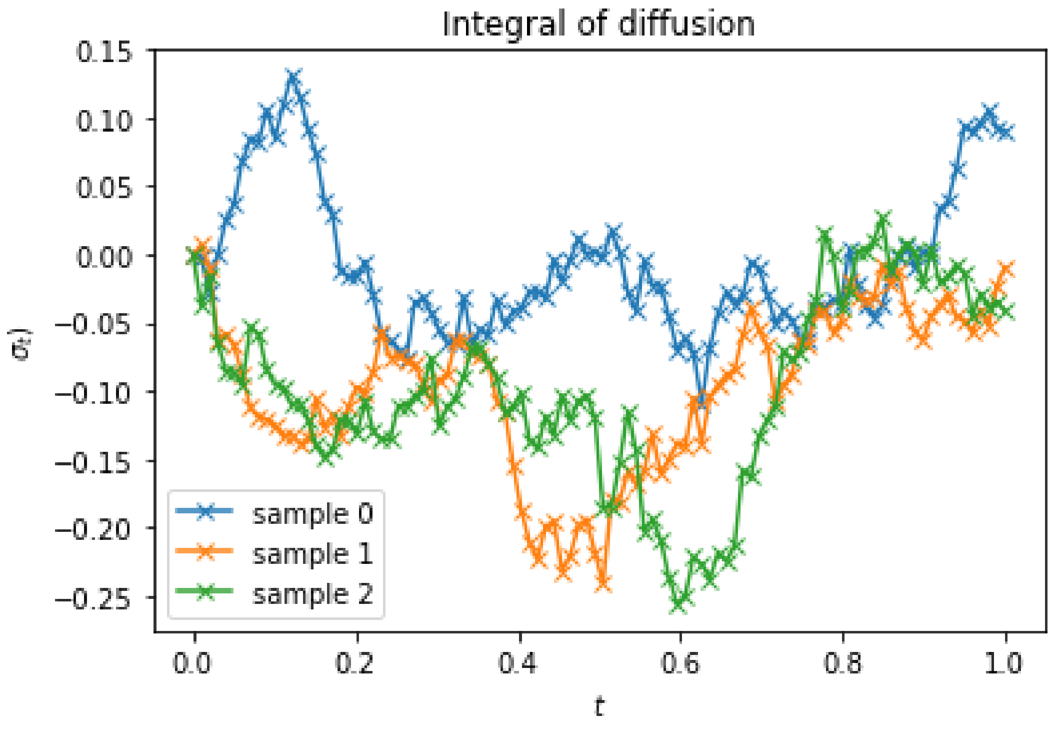
$$g(t, y) = 0.3 * \frac{1}{1 + e^{(cos(t) \cdot e^{-y}))}}$$

where θ is a scalar equal to 0.1. We can plot samples from the SDE, its mean, and its noise shown below.

```
[ ]: # Set data
batch_size, state_size, t_size = 3, 1, 100
ts = torch.linspace(0, 1, t_size)
y0 = torch.full(size=(batch_size, state_size), fill_value=0.1)

sde = NeuralSDE(state_size, batch_size)
with torch.no_grad():
    ys = torchsde.sdeint(sde, y0, ts, method='euler')  # (t_size, batch_size,␣
 ↪state_size) = (100, 3, 1)
    mean = sde.compute_mean(ts, y0=y0)
    noise = sde.compute_noise(ts)


plot(ts, ys, xlabel='$t$', ylabel='$Y_t$', title="Full Ito SDE")
plot(ts, mean, xlabel='$t$', ylabel='$\mu_t)$', title="Drift Term (mean)")
plot(ts, noise, xlabel='$t$', ylabel='$\sigma_t)$', title="Diffusion Term␣
 ↪(noise)")
```




Integral of drift
sample 0
sample 1
sample 2
$\mu_t)$
t
0.1
0.2
0.3
0.4
0.5
0.6
0.0
0.2
0.4
0.6
0.8
1.0


Integral of diffusion
sample 0
sample 1
sample 2
$\sigma_t)$
t
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
−0.20
−0.25
0.0
0.2
0.4
0.6
0.8
1.0