Understanding SDE-GAN

Jingyue Lu
February 11, 2022

1 Introduction

In this short report, we study a recently proposed method of using Wasserstein Gan (WGAN) to fit stochastic differential equations (SDE). The key underlying ideas of the method are introduced in [Kidger et al., 2021b] while implementation details with improved efficiency are provided in [Kidger et al., 2021a]. We refer to this method as SDE GAN from here onwards. In the following, we will mainly focus on providing explanations for the uncleared parts and details for the derivations that are omitted in the papers in the hope to paint a more comprehensive picture of SDE-GAN. In addition, we will conduct a study of applying SDE-GAN to real-world stock index data to evaluate its performance.

2 SDE-GAN: Model

SDE-GAN is a generative model specifically developed for SDEs. Like all other GAN-based models, it consists of a generator and a discriminator. The generator takes the form of Neural SDEs to model a path-valued random variable $Y_{true} : [0, T] \to \mathbb{R}^y$ while the discriminator adopts the formulation of a neural controlled differential equations (CDE) to distinguish whether a sample is real or generated. The generator and the discriminator are trained together in a zero-sum game until the generator has learnt to fool the discriminator. At this time, the generator is able to generate samples Y that have approximately the same distribution as Y_{true}. In terms of the training objective, SDE-GAN adopts the idea of WGAN by using Wasserstein distance instead of the original Jensen-Shannon divergence to measure the difference between the true data distribution and the generated data distribution. We give more details on WGAN, the generator, and the discriminator below.

2.1 WGAN

Given an optimal discriminator, vanilla GAN models encourage the learning of the generator by penalizing the Jensen-Shannon (JS) divergence between the real data distribution and the generated data distribution[1]. However, as demonstrated in [Arjovsky et al., 2017], JS divergence fails to provide a meaningful value when these two distributions are disjoint, which is likely to happen when both distributions are embedded in low-dimensional manifolds. To deal with this issue, [Arjovsky et al., 2017] proposed to replace JS divergence with Wasserstein distance.

Let p_r be the real data distribution and p_g the generated data distribution, Wasserstein distance is defined as

$$W(p_r, p_g) = \inf_{\gamma \sim \Pi(p_r, p_q)} \mathbb{E}_{(x,y)\sim\gamma}[\|x - y\|]. \tag{1}$$

Here, $\Pi(p_r, p_q)$ denotes the set of all possible joint distributions between p_r and p_q. It can be shown that Wasserstein distance manages to provide a smooth measure even when p_r and p_q are disjoint.

Solving for Eq. 1 directly is intractable as it is impossible to enumerate all joint distributions of p_r and p_g. However, with a delicate application of Kantorovich-Rubinstein duality arguments, we can solve the following maximization problem instead:

$$W(p_r, p_g) = \frac{1}{k} \sup_{\|F\|_L \le K} \mathbb{E}_{x\sim p_r}[F(x)] - \mathbb{E}_{x\sim p_g}[F(x)]. \tag{2}$$

In the context of WGAN, we can think the function F as the discriminator and $x \sim p_g$ is produced by the generator g such that $x = G(z)$ for $z \sim p_z$. The only extra requirement is that the discriminator F has to

[1] More details on this claim can be found in the Appendix.

be K-Lipschitz continuous. The final WGAN loss is

$$\min_{\theta\in\Theta}\max_{\phi\in\Phi}\mathbb{E}_{x\sim p_r}[F_\phi(x)] - \mathbb{E}_{z\sim p_z}[F_\phi(G_\theta(z))], \tag{3}$$

where p_z is the noise distribution.

In SDE-GAN, the discriminator has a recurrent nature. Depending on the number of steps T taken and the Lipschitz constant λ of the discriminator at each step, the overall discriminator has a Lipschitz constant $O(\lambda^T)$. For it to be bounded regardless the value of T, hard constraint $\lambda \leq 1$ has to be enforced. To fulfill this requirement, the authors of [Kidger et al., 2021a] have suggested weight clipping and LipSwish activation functions to replace the conventional gradient penalty.

We now give the exact forms for the generator G_θ and the discriminator F_ϕ

2.2 Generator

For the completeness of our discussion, we mention the minimum structure for constructing the generator as Neural SDEs. Recall that we want to model a path-valued random variable $Y_{true} : [0, T] \to \mathbb{R}^y$. The generator is modelled as follows:

$$X_0 = \xi_\theta(V) \tag{4}$$
$$dX_t = \mu_\theta(t, X_t)\, dt + \sigma_\theta(t, X_t) \circ dW_t \tag{5}$$
$$Y_t = \alpha_\theta X_t + \beta_\theta. \tag{6}$$

Here, V is a noise input drawn from $\mathcal{N}(0, I_v)$ and $W : [0, T] \to \mathbb{R}^w$ is a Brownian motion. The functions $\xi_\theta : \mathbb{R}^v \to \mathbb{R}^x$, $\mu_\theta : [0, T] \times \mathbb{R}^x \to \mathbb{R}^x$ and $\sigma_\theta : [0, T] \times \mathbb{R}^x \to \mathbb{R}^{x\times w}$ are neural networks. The strong solution $X : [0, T] \to \mathbb{R}^x$ to the SDE exists under mild conditions. Finally, we have $\alpha_\theta \in \mathbb{R}^{y\times x}$ and $\beta_\theta \in \mathbb{R}^y$. The modelled Y should have approximately the same distribution as Y_{true} when the generator is trained to its optimal.

(Q1) When X is the strong solution, X satisfies the Markov property that is given the present state, the past and future of X are statistically independent. To accommodate general cases including non-Markov processes, we require a read out operation to Y. When α_θ is an invertible matrix, the filtrations of X and Y coincide so Y is a Markov process as well. However, when α_θ is non-invertible, Y is can be an non-Markov process.

(Q2) For a strong solution X to exist, it is required that the initial point X_0 is a random variable which is independent of the Brownian Motion W_t. Thus, an additional noise is required learning the distribution of X_0.

2.3 Discriminator

Unlike other GAN discriminators that take point-wise values as inputs, the discriminator of SDE-GAN has to operate on variables Y that have values over a path. To satisfy this requirement, the authors of [Kidger et al., 2021b] have employed Neural CDEs, proposed in [Kidger et al., 2020]. CDEs are typically used in modeling functions of time series and has the form:

$$dz(t) = h_\theta(t, z(t))dX(t), \tag{7}$$

where $z(t)$ is a continuous path. We note that when $dX(t)$ on the right hand side is replaced by dt, we have the standard ordinary differential equation. As mentioned in [Kidger et al., 2020], the benefit of using $dX(t)$ is that the model has a direct mechanism for incorporating incoming data: we can treat X as an interpolated function of data points $(x_0, x_1, \dots)$ to represent how data changes over time. Since the differential equation Eq. 7 now change in response to the input X, we say the differential question is controlled or driven by X. When the function $h_\theta(t, z(t))$ is a neural network, we call the model Neural CDE.

Following the same idea, SDE-GAN models its discriminator as a neural CDE driven by the path-valued variable Y. The minimum structure for the discriminator is:

$$H_0 = \xi_\phi(Y_0), \tag{8}$$
$$dH_t = f_\phi(t, H_t)\, dt + g_\phi(t, H_t) \circ dY_t, \tag{9}$$
$$D = m_\phi \cdot H_T, \tag{10}$$

where $\xi_\phi : \mathbb{R}^y \to \mathbb{R}^h$, $f_\phi : [0,T] \times \mathbb{R}^h \to \mathbb{R}^h$ and $g_\phi : [0,T] \times \mathbb{R}^h \to \mathbb{R}^{h \times y}$ are neural networks and $m_\phi \in \mathbb{R}^h$. Again, the strong solution $H : [0,T] \to \mathbb{R}^x$ to the SDE exists under mild conditions. Finally, we mention that, as pointed out in Section 2.1, the discriminator has to have a maximum Lipschitz constant one.

(Q3) Eq. 8-10 is the required minimum structure for the discriminator. As mentioned in the CDE paper [Kidger et al., 2020], the learnt initial condition ξ_ϕ is used to avoid translational invariance. Since the evolution of a CDE depends on the control Y only through $\frac{dY}{dt}$, H is invariant to translations of Y. To ensure H is sensitive to the translation of Y, the initial condition H_0 as a function of Y_0 should be learnt. The authors have mentioned other strategies to incorporate translation information of Y. For instance, replacing Y_t by $\tilde{Y} = (Y_t, tY_0)$ to incorporate translational information into $\frac{d\tilde{Y}}{dt}$. Finally, applying a linear map to H_T could be a learning consideration. A linear map can help with scaling, which could facilitate the learning process when H_T is too large or too small. We point out that in the original pytorch library, $f_\phi(t, H_t)dt$ term is removed.

Using CDE for the discriminator is a nice choice. With the help of CDE, all points Y_t generated along the path are considered in the loss in a natural way. Other statistics either lose the time information, such as mean, or lose the path information, such as focusing at one time point Y_T.

So far, we have introduced the model structure for SDE-GAN. In the next section, we will study how to carry out the learning procedure for it.

3 SDE-GAN: Learning

Given the model structure, the forward learning for SDE-GAN is straightforward. However, in terms of the backward pass, we note that both the generator and the discriminator have a recurrent nature (they have to solve SDEs numerically). This results in high memory cost for standard back-propagation, in which case, the memory cost grows linearly with the number of steps t taken. To deal with this issue, the authors of [Kidger et al., 2021a] have used the continuous adjoint method combined with their newly developed SDE solver: the reversible Heun method. We discuss this modified back-propagation below.

3.1 Adjoint Method

Using the adjoint method for back-propagation is first proposed by Chen et al. [2018] in solving neural ordinary differential equations (ODE). Unlike standard back-propagation, the adjoint method treats the original ODE as a black-box and computes gradients by solving another ODE backward in time. Since there is no need for the adjoint method to store any intermediate quantities of forward pass for gradient computation, the adjoint method has a constant memory cost. Li et al. [2020] later generalised this idea for computing gradients of SDEs.

Mathematically, consider some Stratonovich SDE

$$dZ_t = \mu(t, Z_t)\, dt + \sigma(t, Z_t) \circ dW_t \qquad \text{for } t \in [0,T],$$

and a loss $L : \mathbb{R} \to \mathbb{R}$ on the terminal value Z_T. The adjoint process is defined as $A_t := \frac{dL(Z_T)}{dZ_t} \in \mathbb{R}^z$, which is a solution to the SDE

$$dA_t^i = -A_t^j \frac{\partial \mu^j}{\partial Z^i}(t, Z_t)dt - A_t^j \frac{\partial \sigma^{j,k}}{\partial Z^i}(t, Z_t) \circ dW_t^k. \tag{11}$$

Provided with the initial condition $A_T = \frac{dL(Z_T)}{dZ_T}$, we solve the SDE Eq. 11 numerically to get the desired gradient $A_0 = \frac{dL(Z_T)}{Z_0}$.

3.2 The Reversible Heun Method

Kidger et al. [2021a] introduced Reversible Heun method to solve the SDE Eq. 11. The key advantage of the method is that it is algebraically reversible. Normally, Z_t computed along the way of solving the SDE Eq. 11 are different from their values in the forward pass. Reversible Heun method allows the values of Z_t in forward and backward passes to match exactly, leading to precise gradient computation using the adjoint method.

Algorithms of Reversible Heun method can be found in [Kidger et al., 2021a]. Below, we give further details on gradients computation carried out in Algorithm 2 in [Kidger et al., 2021a].

Assume $\frac{\partial L(Z_T)}{\partial z_T}$ is provided. We start by calculating $\frac{\partial L(Z_T)}{\partial \hat{z}_T}$ as follows:

$$\frac{\partial L(Z_T)}{\partial \mu_T} = \frac{\partial L(Z_T)}{\partial z_T} \cdot \frac{1}{2}\Delta t, \tag{12}$$

$$\frac{\partial L(Z_T)}{\partial \sigma_T} = \frac{\partial L(Z_T)}{\partial z_T} (\frac{1}{2}\Delta W), \tag{13}$$

$$\frac{\partial L(Z_T)}{\partial \hat{z}_T} = \frac{\partial L(Z_T)}{\partial \mu_T}\frac{\partial \mu_T}{\partial \hat{z}_T} + \frac{\partial L(Z_T)}{\partial \sigma_T}\frac{\partial \sigma_T}{\partial \hat{z}_T}. \tag{14}$$

Here, $\frac{\partial \mu_T}{\partial \hat{z}_T}$ and $\frac{\partial \sigma_T}{\partial \hat{z}_T}$ are computed through standard network propagation. After that, we conduct an iteration procedure for $0 \leq n < T$. At each time step n, we are provided with the inputs:

$$\frac{\partial L(Z_T)}{\partial z_{n+1}}, \quad \frac{\partial L(Z_T)}{\partial \hat{z}_{n+1}}.$$

We first make some preliminary computations:

$$\begin{aligned}
\frac{\partial L(Z_T)}{\partial z_{n+1}}\frac{\partial z_{n+1}}{\partial \mu_n} &= \frac{\partial L(Z_T)}{\partial z_{n+1}} \cdot \frac{1}{2}\Delta t, \\
\frac{\partial L(Z_T)}{\partial z_{n+1}}\frac{\partial z_{n+1}}{\partial \sigma_n} &= \frac{\partial L(Z_T)}{\partial z_{n+1}} (\frac{1}{2}\Delta W), \\
\frac{\partial L(Z_T)}{\partial \hat{z}_{n+1}}\frac{\partial \hat{z}_{n+1}}{\partial \mu_n} &= \frac{\partial L(Z_T)}{\partial \hat{z}_{n+1}} \cdot \Delta t, \\
\frac{\partial L(Z_T)}{\partial \hat{z}_{n+1}}\frac{\partial \hat{z}_{n+1}}{\partial \sigma_n} &= \frac{\partial L(Z_T)}{\partial \hat{z}_{n+1}} (\Delta W).
\end{aligned}$$

Then, we make the following calculations:

$$\frac{\partial L(Z_T)}{\partial z_n} = \frac{\partial L(Z_T)}{\partial z_{n+1}} + 2 \cdot \frac{\partial L(Z_T)}{\partial \hat{z}_{n+1}}, \tag{15}$$

$$\frac{\partial L(Z_T)}{\partial z_n}\frac{\partial z_n}{\partial \mu_n} = \frac{\partial L(Z_T)}{\partial z_n} \cdot \frac{1}{2}\Delta t, \tag{16}$$

$$\frac{\partial L(Z_T)}{\partial \mu_n} = \frac{\partial L(Z_T)}{\partial z_{n+1}}\frac{\partial z_{n+1}}{\partial \mu_n} + \frac{\partial L(Z_T)}{\partial \hat{z}_{n+1}}\frac{\partial \hat{z}_{n+1}}{\partial \mu_n} + \frac{\partial L(Z_T)}{\partial z_n}\frac{\partial z_n}{\partial \mu_n}, \tag{17}$$

$$\frac{\partial L(Z_T)}{\partial z_n}\frac{\partial z_n}{\partial \sigma_n} = \frac{\partial L(Z_T)}{\partial \hat{z}_n}(\frac{1}{2}\Delta W), \tag{18}$$

$$\frac{\partial L(Z_T)}{\partial \sigma_n} = \frac{\partial L(Z_T)}{\partial z_{n+1}}\frac{\partial z_{n+1}}{\partial \sigma_n} + \frac{\partial L(Z_T)}{\partial \hat{z}_{n+1}}\frac{\partial \hat{z}_{n+1}}{\partial \sigma_n} + \frac{\partial L(Z_T)}{\partial z_n}\frac{\partial z_n}{\partial \sigma_n}, \tag{19}$$

$$\frac{\partial L(Z_T)}{\partial \hat{z}_n} = -\frac{\partial L(Z_T)}{\partial \hat{z}_{n+1}} + \frac{\partial L(Z_T)}{\partial \mu_n}\frac{\partial \mu_n}{\partial \hat{z}_n} + \frac{\partial L(Z_T)}{\partial \sigma_n}\frac{\partial \sigma_n}{\partial \hat{z}_n}. \tag{20}$$

Similarly, $\frac{\partial \mu_n}{\partial \hat{z}_n}$ and $\frac{\partial \sigma_n}{\partial \hat{z}_n}$ are evaluated through standard network back-propagation. We treat $\frac{\partial L(Z_T)}{\partial z_n}$, $\frac{\partial L(Z_T)}{\partial \hat{z}_n}$ as the input values for the next step.

In terms of parameters θ for $\mu(\cdot;\theta)$ and $\sigma(\cdot;\theta)$, we note that

$$\frac{\partial dL(Z_T)}{\partial \theta} = \sum_{n=0}^{T} \frac{\partial dL(Z_T)}{\partial \mu_n}\frac{\partial \mu_n}{\partial \theta} + \frac{\partial dL(Z_T)}{\partial \sigma_n}\frac{\partial \sigma_n}{\partial \theta}. \tag{21}$$

Thus, we calculate $\frac{\partial \mu_n}{\partial \theta}$, $\frac{\partial \sigma_n}{\partial \theta}$ at the same time as computing $\frac{\partial \mu_n}{\partial \hat{z}_n}$, $\frac{\partial \sigma_n}{\partial \hat{z}_n}$. Then we evaluate

$$\frac{\partial dL(Z_T)}{\partial \mu_n}\frac{\partial \mu_n}{\partial \theta} + \frac{\partial dL(Z_T)}{\partial \sigma_n}\frac{\partial \sigma_n}{\partial \theta}$$

and add it to the sum of previous summands. Only the updated partial sum is recorded and carried into the next step $n-1$.

4 Experiments (NEW)

(Q4) Before we go into the experiments, we make some comments regarding section 4.1 in [Kidger et al., 2021b]. If two processes have the same marginal distributions at each given single t value, the two processes are not guaranteed to be the same. For instance, consider the example of an AR(1) process,

$$y(t) = a \cdot y(t-1) + e, \text{ where } e \sim N(0, \sigma^2), \quad y(0) \sim N(0, \frac{\sigma^2}{1-a^2}). \tag{22}$$

We observe that the distribution of $y(t)$ is $N(0, \frac{\sigma^2}{1-a^2})$. Thus, it is possible for a process $y'(t)$ with a', σ', such that $a' \neq a$, $\sigma' \neq \sigma$, to have the same marginal distribution with $y(t)$ as long as $\frac{\sigma'^2}{1-a'^2} = \frac{\sigma^2}{1-a^2}$.

Two processes $y(t)$ and $y'(t)$ are equivalent if they have the same finite dimensional distributions. We thus should compare the finite dimensional distributions of $y(t)$ and $y'(t)$ for an arbitrary set of times $\{t_1, \ldots, t_k\}$ where k is arbitrary and each t_i satisfies $t_i \in T$. In this case, we could simply choose the set $\{t_i - 1, t_i\}$. By noting that given $a \neq a'$,

$$\text{Cov}(y(t_i), y(t_i - 1)) \neq \text{Cov}(y'(t_i), y'(t_i - 1),$$

we conclude $(y(t_i), y(t_i - 1))$ does not have the same distribution as $(y'(t_i), y'(t_i - 1))$. Consequently, $y(t)$ and $y'(t)$ are different processes.

Statistically, we can test two processes by choosing multiple arbitrary sets of times $\{t_1, \ldots, t_k\}$ and then compare the mean and variance between $(y(t_1), \ldots, y(t_k))$ and $(y'(t_1), \ldots, y'(t_k))$ for each set of times.

4.1 (Q6) Synthetic Data

We use synthetic data to simulate the stock index. Assume the synthetic data follows the process

$$dS/S = a dt + b dW, \tag{23}$$

By setting $a = 0.1$ and $b = 0.2$, we generate 2000 data points from $t = 0$ to $t = 2000$. We attempt to recover the constants 0.1 and 0.2 by fitting the SDE-GAN to the generated synthetic data points.

To facilitate the learning process, instead of directly fitting S, we study $\log(S)$. A direct application of Itô's Lemma shows that

$$d\log(S) = a'\, dt + b\, dW, \tag{24}$$

where $a' = a - \frac{b^2}{2}$. We note that once a' is known, we can easily compute $a = a' + \frac{b^2}{2}$. We focus on fitting a' from here onwards. As SDE-GAN is developed for Stratonovich SDEs, we also need to transform the above Itô equation into its corresponding Stratonovich form. In this case, given that a', b are constants, both Itô equation and Stratonovich equation have the same drift and diffusion terms.

Since $\log(S)$ has the Markov property, the next state only depends on the current state. We thus set $t = 2$. Each sample consists of two data points $(\log(s_t), \log(s_{t+1}))$. The learnt initial condition should recover the

starting point of each sample $\log(s_t)$ in $(\log(s_t), \log(s_{t+1}))$. When all sample points are considered, the learnt function ξ_θ needs to be able to simulate $\log(s_0), \ldots, \log(s_{1999})$. As a result, the function ξ_θ has to depend on t, which complicates the learning process. We note that when the drift a' and the diffusion b are constants, it is easy to translate the sample points to start from 0. To be more specific, the equation

$$\log(S_{t+1}) = \log(S_t) + \int_t^{t+1} (a - \frac{b^2}{2})dt + \int_t^{t+1} bdW \tag{25}$$

can be equivalently written as

$$\log(S_{t+1}) - \log(S_t) = 0 + \int_0^1 (a - \frac{b^2}{2})dt + \int_0^1 bdW. \tag{26}$$

We thus transform all samples to $(0, \log(s_{t+1}) - \log(s_t))$. The initial point is always set as 0 and the function ξ_θ is removed.

Additional modifications are made to the SDE-GAN architecture. We summarise all changes below:
Generator

- We replace $X_0 = \xi_\theta(V)$ by $X_0 = 0$ in Eq. 4.
- Both the drift function $\mu_\theta(t, X_t)$ and the diffusion function $\sigma_\theta(t, X_t)$ are changed to a trainable Variable respectively to represent constants.
- The readout function in Eq. 6 is removed. That is, $Y_t = X_t$ is used.
- hidden size, initial noise size and noise size are all set to 1 to present the fact that we are directly simulating the process with X, instead of treating X as the hidden process.

Discriminator

- all functions $\xi_\phi(Y_0)$, $g_\phi(t, H_t)$ and m_ϕ are set as a simple fully connected layer. The function f_ϕ is kept as 0 as in the original library.

4.1.1 Results

Since the modified data samples all start from 0, we can random shuffle the data set and split it into training, validation, and test datasets. We use Adadelta optimizer with learning rate 0.1 for both the generator and the discriminator. We reran the experiments couple times. A typical loss plot is provided in Figure 1. It is consistent with the often observed GAN behaviour (The loss oscillates before it converges). As a side note,

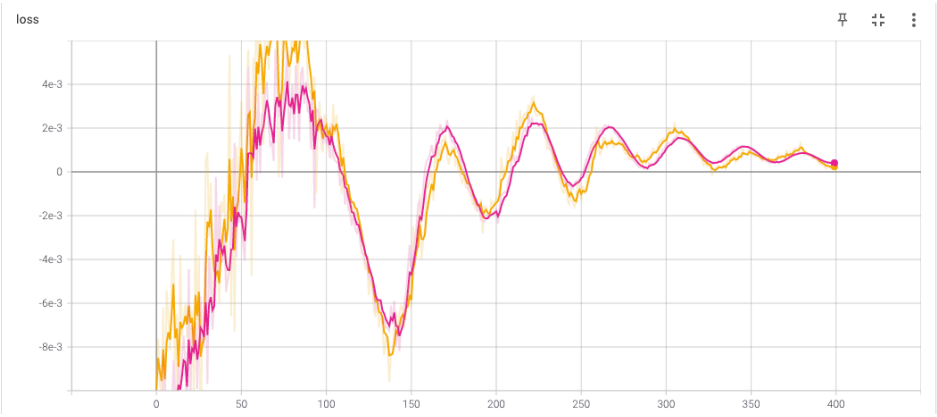

Figure 1: A loss plot for the synthetic experiment. The red line is for the training data and the yellow line is for the validation data.

we mention that the starting point generally has a heavy influence on the convergence of GAN models. We observed divergent behaviour in some runs, as shown in Figure 2.

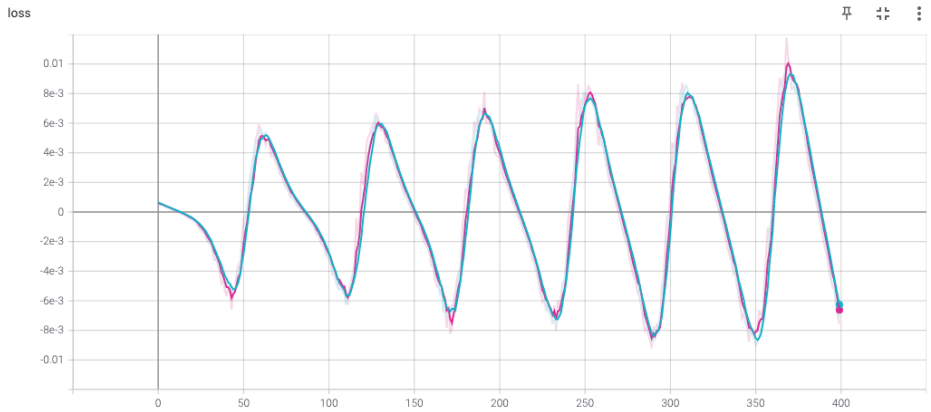

Figure 2: A loss plot for the divergent behaviour of SDE-GAN on the synthetic data. The blue line is for the training data and the red line is for the validation data.

In the experiment shown in Figure 1, when the model converges, the fitted value for a' is 0.0513 and the fitted value for b is 0.007, which are a bit far away from the ground truth values $a' = 0.08$, $b = 0.2$, especially for the diffusion value.

We mention some difficulties in fitting values with SDE-GAN. Firstly, we found that the SDE-GAN model converges to different values of a' and b when the experiment is run multiple times. For instance, in another converged run, the fitted value for a' is 0.067 and for b is 0.993. Secondly, a convergence measure should be designed to control when to terminate the training. In our experiments, we have used fixed 400 epochs for simplicity. In some cases, it is unclear whether the SDE-GAN converges or not. An example is given in Figure 3.

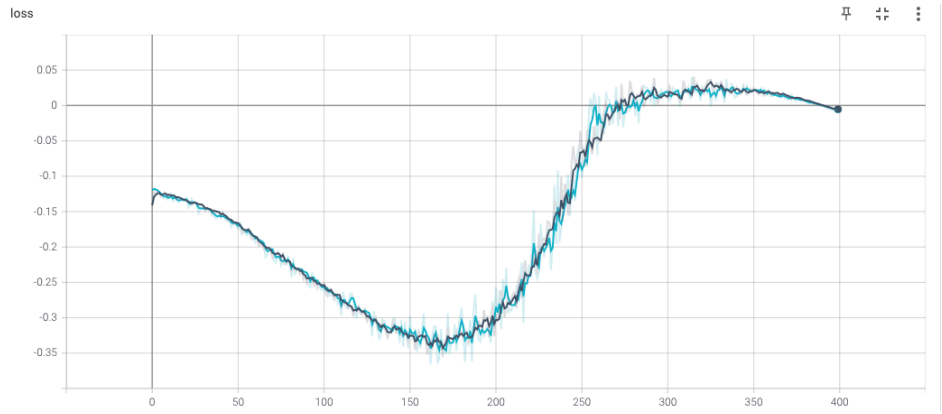

Figure 3: An example where we are unclear whether the SDE-GAN model converges. Although the loss is around zero when training stops. The loss is likely to oscillate further when the model gets trained longer.

4.2 (Q5, Q7) HSI Data

We fit the HSI data in a similar way. Denote HSI data as s. We first transform the data into process sample points $(0, \log(s_{t+1}) - \log(s_t))$. To get a rough approximation of a' and b, we compute the mean and standard deviation of $\log(s_{t+1}) - \log(s_t)$, which are 0.00 and 0.0117.

Similar observations to those of synthetic data are made. For a converged model, the final fitted value for a' is 0.021 and for b is 0.017.

5 Comments on Tensorflow Implementation

For this project, we have also provided a re-implementation of the original pytorch library **torchsde** in tensorlfow, titled **tfsde**. We make some comments regarding our tensorflow implementation.

The tfsde is a direct reimplementation of the original library in tensorflow. We followed the original pytorch library closely when constructing the whole pipeline.

In terms of accuracy, a direct comparison of the forward pass and backward gradient computation between tfsde and torchsde is performed. To be more specific, when randomness parts and model parameters are set the same for both tfsde and torchsde, we verified that these two packages perform exactly same in the forward pass (same outputs) and the backward pass (same gradients are computed). You can find more information under the Tests section on README.md. All scripts used for testing can be found in the directory: tests. In terms of computational efficiency, we find tfsde is slower than torchsde. Since tfsde is a direct reimplementation, the computational difference could be mainly caused by the underlying execution of tensorflow and pytorch.

There are several aspects we would like to improve. Firstly and most importantly, we would like optimize the computational graph to better accommodate to the underlying tensorflow library. So the performance speed of tfsde can match up with torchsde. Secondly, for the purpose of this project, we have only reimplemented the reversible heun method. The original library contains many other solvers, including Euler Heun, Midpoint, SRK and Milstein. It would be nice to include these solvers in the tfsde as well.

In addition, we mention some minor technique details that might cause some confusions.

- In the paper [Kidger et al., 2021b], the authors mentioned that they used gradient penalty for Lipschtz regularisation. However, as commented in [Kidger et al., 2021a]: "Gradient penalty requires calculating second derivatives (a double backward). This complicates the use of continuous adjoint methods: the double continuous adjoint introduces substantial truncation error; sufficient to obstruct training and requiring small step sizes to resolve." To deal with this, [Kidger et al., 2021a] suggested to use weight clipping and LipSwish activation functions. The current torchsde library has adopted these changes. We implemented tfsde to include these modifications accordingly.
- We note that tfcde.cdeint is exactly the same as the torchcde.cdeint. The reason we removed the "tuple(self._func.parameters())" is because the different ways tensorflow and pytorch handle self-defined backward propagation. More in detail, in tensorflow, when maintaining a list of parameter gradients, the order is not guaranteed to be consistent with the layer number in the model. Thus, when doing self-defined backward propagation, we want to use a separate list to keep track of the gradients of ys_coeffs. As a result, in our implementation, we check whether adjoint params is an empty list or not to know whether a separate list for ys_coeffs should be generated or not[2]. We then add self.func.parameters[3].
- Weight regularisation is an optimisation technique. It is a regulariser to help with the overfitting issue. In pytorch, L_2 weight regularisation is directly handled by the optimiser: weight decay parameter is an input for the optimiser. In tensorflow, this is not supported. Thus, to be consistent with the torchsde library, we performed the weight decay explicitly by adding the L_2 norm of all parameters to the loss.

6 Discussions

In this report, we have studied and evaluated the method SDE-GAN. Before we end, we list several limitations of our study. Firstly, the authors of [Kidger et al., 2021a] have developed new ways for speeding up the Brownian motion sampling. Since the explanation in the paper is clear and straightforward, we decide not to discuss these improvements. Secondly, in terms of the experiments, we have only used a simple measure of test loss to evaluate the models' performance.

[2]see https://github.com/JingyueLu/tfsde/blob/main/tfcde/solver.py#L185
[3]shown https://github.com/JingyueLu/tfsde/blob/main/tfcde/solver.py#L186

7 Appendix

In this section, we collect some basic definitions and facts that might be useful for readers.

KL(Kullback-Leibler) Divergence

KL divergence is a statistical distance that measures how one probability distribution q diverges is different from a second reference probability distribution p.

$$D_{KL}(p||q) = \int_x p(x) \log \frac{p(x)}{q(x)}\, dx.$$

D_{KL} achieves the minimum zero when $p(x) == q(x)$ everywhere.

Remark. KL divergence is asymmetric. In cases where $p(x)$ is close to zero, but $q(x)$ is significantly non-zero, the q's effect is disregarded.

Jensen-Shannon Divergence

Jenson-Shannon Divergence is a symmetrized and smoothed version of the KL divergence. It is defined as

$$D_{JS}(p||q) = \frac{1}{2} D_{KL}(p||\frac{p+q}{2}) + \frac{1}{2} D_{KL}(q||\frac{p+q}{2}).$$

Generative Adversarial Network (GAN)

Denote the generator as G and the discriminator as D. Moreover, let p_z denote the data distribution over noise input z; p_g the generator's distribution over x; and p_r the data distribution over real samples x.

The standard training objective for GAN is

$$\begin{aligned}\min_G \max_D L(D,G) &= \mathbb{E}_{x\sim p_r(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1-D(G(z)))] \\ &= \mathbb{E}_{x\sim p_r(x)}[\log D(x)] + \mathbb{E}_{x\sim p_g(x)}[\log(1-D(x))].\end{aligned}$$

It can be computed that for the well-defined loss function

$$L(G,D) = \int_x \left(p_r(x)\log(D(x)) + p_g(x)\log(1-D(x))\right),$$

the optimal value for D is $D^*(x) = \frac{p_r(x)}{p_r(x)+p_g(x)}$. When the generator is trained to optimal, the distribution p_g is very close to p_r, which leads to $D^*(x) = \frac{1}{2}$. As a result, when both G and D are at their optimal, we have

$$\begin{aligned}L(G,D^*) &= \log\frac{1}{2}\int_x p_r(x)\, dx + \log\frac{1}{2}\int_x p_g(x)\, dx \\ &= -2\log 2.\end{aligned}$$

To connect the loss with JS divergence, we note that

$$\begin{aligned}D_{JS}(p_r||p_g) &= \frac{1}{2}D_{KL}(p_r||\frac{p_r+p_g}{2}) + \frac{1}{2}D_{KL}(p_g||\frac{p_r+p_g}{2}) \\ &= \frac{1}{2}\left(\log 2 + \int_x p_r(x)\log\frac{p_r(x)}{p_r(x)+p_g(x)}\, dx\right) + \frac{1}{2}\left(\log 2 + \int_x p_r(x)\log\frac{p_g(x)}{p_r(x)+p_g(x)}\, dx\right) \\ &= \frac{1}{2}\left(\log 4 + L(G,D^*)\right).\end{aligned}$$

It follows,

$$L(G,D^*) = 2D_{JS}(p_r||p_g) - 2\log 2.$$

The loss of GAN quantifies the similarity between p_r and p_q with JS Divergence. The optimal G^* gives zero JS divergence, resulting $L(G^*,D^*) = -2\log 2$, which is consistent with above analyses.

Itô's Process

An Itô process or stochastic integral is a stochastic process on $(\Sigma, \mathcal{F}, \mathbb{P})$ adopted to $\mathcal{F}_t$ which can be written in the form

$$X_t = X_0 + \int_0^t \mu_s \, ds + \int_0^t \sigma_s \, dW_s,$$

where $\mu, \sigma \in \mathcal{L}_2$. Equivalently, we can write it as

$$dX_t = \mu_t \, dt + \sigma_t \, dW.$$

Itô's Lemma

Let X_t be an Itô process $dX_t = \mu_t \, dt + \sigma_t \, dW_t$. Suppose $g(x) \in C^2(\mathbb{R})$ is a twice continuously differentiable function. Suppose $g(X_t) \in \mathcal{L}_2$. Then $Y_t = g(X_t)$ is again an Itô process and

$$dY_t = \frac{\partial g(X_t)}{\partial x} dX_t + \frac{1}{2} \frac{\partial^2 g(X_t)}{\partial x^2} (dX_t)^2.$$

That is,

$$dY_t = \left(\frac{\partial g(X_t)}{\partial x} \mu_t + \frac{1}{2} \frac{\partial^2 g(X_t)}{\partial x^2} (X_t) \sigma_t^2 \right) dt + \frac{\partial g(X_t)}{\partial x} \sigma_t dW_t.$$

Itô integral vs Stratonovich Integral

Suppose X_t solves the Stratonovich equation

$$dX_t = \mu_t \, dt + \sigma_t \circ dW_t.$$

Then X_t solves the Itô equation

$$dX_t = \left(\mu_t + \frac{1}{2} \sigma_t \frac{\partial \sigma_t}{\partial x} \right) dt + \sigma_t \, dW_t.$$

Conversely, suppose X_t solves the Itô equation

$$dX_t = \mu_t \, dt + \sigma_t \, dW_t.$$

Then X_t also solves the Stratonovich equation

$$dX_t = \left(\mu_t - \frac{1}{2} \sigma_t \frac{\partial \sigma_t}{\partial x} \right) dt + \sigma_t \circ dW_t.$$

References

Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein generative adversarial networks. In International conference on machine learning, pages 214–223. PMLR, 2017.

Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David Duvenaud. Neural ordinary differential equations. arXiv preprint arXiv:1806.07366, 2018.

Patrick Kidger, James Morrill, James Foster, and Terry Lyons. Neural controlled differential equations for irregular time series. arXiv preprint arXiv:2005.08926, 2020.

Patrick Kidger, James Foster, Xuechen Li, and Terry Lyons. Efficient and accurate gradients for neural sdes. arXiv preprint arXiv:2105.13493, 2021a.

Patrick Kidger, James Foster, Xuechen Li, Harald Oberhauser, and Terry Lyons. Neural sdes as infinite-dimensional gans. arXiv preprint arXiv:2102.03657, 2021b.

Xuechen Li, Ting-Kam Leonard Wong, Ricky TQ Chen, and David Duvenaud. Scalable gradients for stochastic differential equations. In International Conference on Artificial Intelligence and Statistics, pages 3870–3882. PMLR, 2020.