Timestep Free Diffusion Model [Proposal]

Logic Q

Mobius AGI Lab

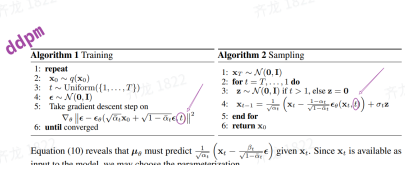
fenneishitiger@gmail.com

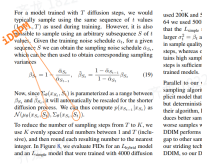
1. Abstract

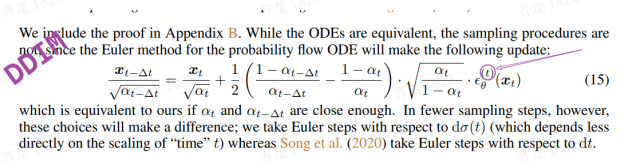
By converting $noise = model(x, t)$ to $noise, t = model(x)$, this research aims to train a more powerful, structured, flexible, and generalizable diffusion-based model. This modification has the potential to ensure a more structurally sound and efficient system, enhancing the model's overall performance and applicability across various tasks.

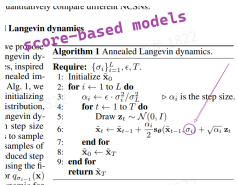
> https://github.com/fenneishi/timestep-free-diffusion-model.git
>
> github.com

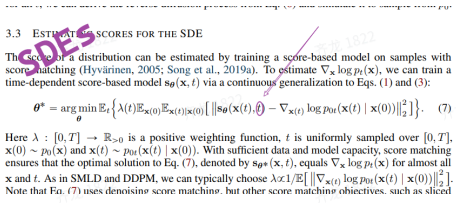
2. Introduction

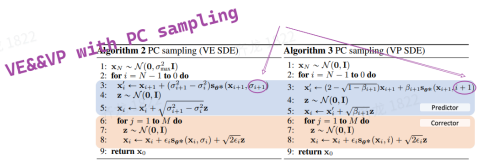


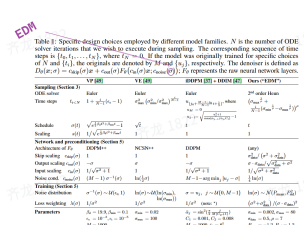












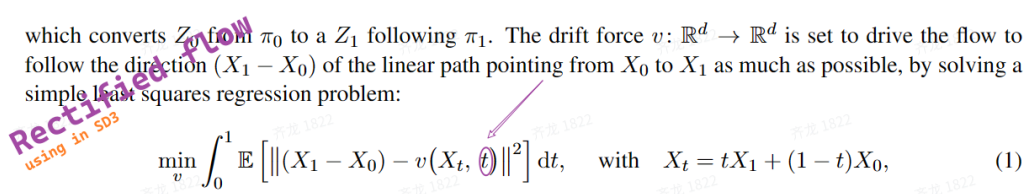


Figure 1:

Despite the vast and complex framework of diffusion-based(or score-based) models, the pattern $noise = model(x, t)$ is a common feature in almost all of them.This research begins by analyzing the role of t (or σ) within these models(#todo). By transforming the equation from $noise = model(x, t)$ to $noise, t = model(x)$, this research aims to develop a diffusion-based model that is more powerful, structured, flexible, and generalizable.

3. Motivation

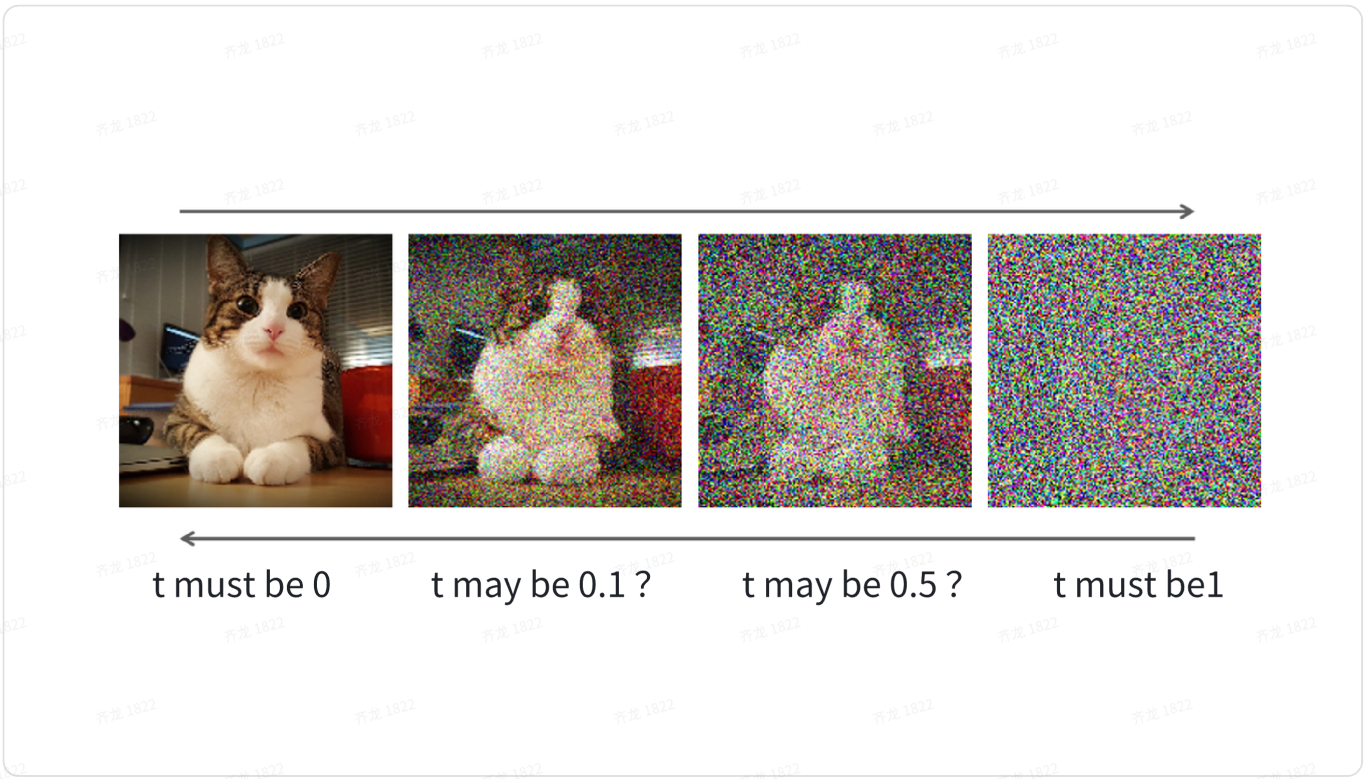


Figure 2

By simply observing with the eye, humans can roughly estimate the t for each image. So, why can't a neural network do the same? If an NN can estimate t , why do we still need to input t ? If we eliminate the need to input t , we can transform the left-side timestep-conditioned diffusion model(**TCDM**) into the right-side timestep-free diffusion model(**TFDM**). Intuitively, this transformation would make the entire system more powerful, structured, flexible, and generalizable.

current: TCDM

```
x = torch.randn()
for t in [1,0.9,0.7...,0]
    noise = model(x,t)
    x = x - noise
```

->

What we want: TFDM

```
x,t,epsilon = torch.randn(),1,1e-3
while epsilon < t :
    noise,t = model(x)
    x = x - noise
```

Figure 3

> If backpropagation used fixed training steps, AlexNet might not exist.

By transforming TCDM into TFDM, we may achieve the following potential benefits:

Speed Up:

TFDM can automatically determine the number of iterations based on the complexity of the generation task. For example, it uses fewer iterations for simple images and more iterations for complex images, ensuring that both simple and complex images ultimately achieve the same quality.

More Iterations, Higher Quality

Since denoising iterations are no longer bound to a fixed schedule, theoretically, we can iterate indefinitely until the desired quality is achieved.

Towards Better Local Optima

t essentially represents the inverse signal-to-noise ratio (NSR, i.e. $1/SNR$). Without inputting t , the model needs to infer the NSR based on x . NSR is a crucial signal deeply embedded in x and is highly related to the denoising process. When we push the model to learn this signal on its own, it likely guides the model's weight backpropagation towards better local optima. Overall, by removing the input of t , we construct a more difficult task. Generally speaking, more difficult tasks tend to push out more powerful models.

Decoupling and Schedule free

In the current TCDM model, t represents the inverse signal-to-noise ratio (NSR, i.e. $1/SNR$) and the input is (x, t) . This input method forces the coupling of "simple neural network input-output behavior" with the "complex diffusion process" during training, resulting in a tightly coupled system that increases complexity in both training and inference stages.

Adopting a timestep-free diffusion model can address this issue. If we extend t to the actual NSR and consider $-noise$ as $\Delta information$, allowing the model to predict SNR and $\Delta information$ during the training phase, we can achieve the following:

1. Train the model using any diffusion schedule.
2. Mix different diffusion schedules during training to enhance generalization.
3. Train using non-diffusion schedules, as long as SNR can be calculated (e.g., mixing multiple images, applying low-frequency and high-frequency filtering, random blur masks,random masked diffusion(https://arxiv.org/pdf/2406.17688) during training. This approach makes the model more flexible and generalizable).

4. Perform inference without relying on any specific schedule, starting from any point in the x space, not just limited to $torch.randn()$

```
x,nsr,epsilon= torch.randn(),1e3,1e-3
while epsilon < nsr
    delta_information,nsr= model(x)
    x = x + delta_information
```

Figure 4

In conclusion, by removing the timestep input t and introducing the inverse signal-to-noise ratio (NSR),we can simplify and decouple(More see APPENDIX.B) the model's training and inference processes. The model can self-adjust by predicting SNR and $\Delta information$, allowing it to train with any diffusion schedule and flexibly handle different input conditions during inference, thereby improving the model's generalization ability and flexibility.

Model as First-class citizen:

In TCDM, the neural network (NN) has always been a second-class citizen, while the diffusion schedule is the first-class citizen. The diffusion schedule is responsible for the overall structure, architecture, and framework , and the neural network is just one of the components (albeit the most critical one). We attempt to solve all problems using the diffusion schedule, only resorting to the neural network when symbolic mathematics cannot resolve the issue.

Imagine an extremely simple dataset where all images are completely black (all pixel values are 0). We use $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon$ to construct the forward diffusion process to train the neural network. Since x_0 is completely black, the task of the neural network is very simple: regardless of what x_t is (no matter how noisy), it only needs to output a completely black image. Theoretically, this model is very easy to train and will converge quickly. A well-trained model will be able to output the expected completely black image even if the input is pure noise.

However, because the neural network is a second-class citizen, even if the first iteration (t=T) of denoising has already produced the final result, it will still be regarded as the result of (t=T-1) instead of t=0, and rigid iterations will continue.

> This is like a company planning to earn $10 million annually, reach $100 million in 10 years, and then go public. If a very smart employee generates $100 million in the first year, but the company only records $10 million and continues with the original plan, it will take 10 years to go public instead of one.

The neural network is the most powerful part of the entire denoising process, while the diffusion schedule is the weakest part. We should make the powerful part the first-class citizen, giving it

higher authority, thus making the whole system stronger and more flexible.

In TFDM, the situation is exactly the opposite. The TFDM approach aims to resolve this imbalance by removing the timestep input t and allowing the model to self-adjust by predicting the inverse signal-to-noise ratio (nsr). These changes make the NN the core, while the diffusion schedule becomes auxiliary. If the NN predicts that the nsr is sufficiently small, the entire denoising process can stop at any time, rather than rigidly continuing the iterations. Conversely, if the NN predicts that the nsr is not small enough, the entire denoising process can continue rather than being forcibly stopped after reaching a certain number of iterations.

> **Modern human**
>
> You can totally run on just two legs, so why use all four?
>
> **Ape ancestor**
>
> I've got four legs, why should I only use two?
>
> **Modern human**
>
> But you can run with just two legs.
>
> **Ape ancestor**
>
> But I've got four legs!

4. Method

TCDM

```
t = self.time_mlp(time)

x = unet(x,t)
```

->

TFDM

```
t = self.time_mlp(time)
t = t_signal(t).detach() * t + (1 - t_signal(t).detach()) * torch.randn_like(t)
x,t = unet(x,t)
```

Figure 5：The left figure shows the general paradigm of TCDM. The right figure demonstrates a plug-and-play method that can be easily integrated into any diffusion model, regardless of its architecture.

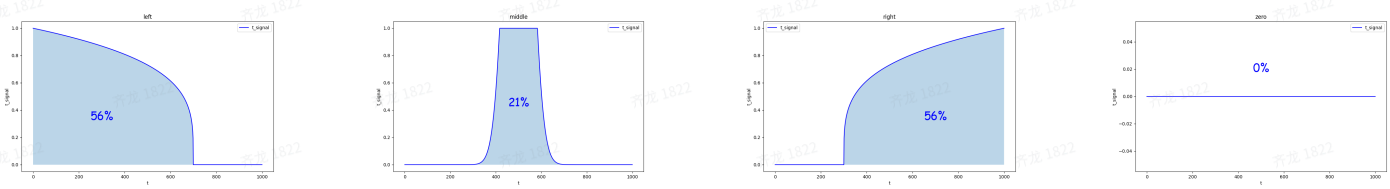


Figure 6：$Left$, $Middle$, $Right$, and $Zero$, all curves were heuristically determined in a one-off manner, without using hyperparameter search.

Controlling the Input Proportion of t :

Controlling the input proportion of t is not a simple "yes or no" issue. To thoroughly investigate the impact of t input on model performance, we need to propose a method that is applicable to any neural network architecture and modifies the neural network as little as possible (This allows us to better perform a controlled experiment). On the other hand, we need a flexible way to control the strength of t input throughout the entire diffusion (and denoising) process, rather than a binary choice of whether to input t or not.

To address this, we introduce the concept of t_signal and use the formula

$$t = t\_signal(t) * t + (1 - t\_signal(t)) * torch.randn\_like(t)$$

to flexibly control the strength of t input at different stages of the schedule (see Figure 5). Here, $t\_signal(\cdot)$ is a simple function mapping that can theoretically be any one-dimensional function to control the strength of t input.

In this study, we specifically investigate four cases— $Left$, $Middle$, $Right$, and $Zero$ (see Figure 6) —to study their effects on model performance (see Chapter 5 for Experimental Results).

For the rest:

For the rest, we essentially follow the original DDPM paper, except:

- We use conditional instance normalization to integrate the information of t into the UNet.

Detail

For detailed content, refer to the source code.

https://github.com/fenneishi/timestep-free-diffusion-model.git
github.com

5. Experimental result

		Min FID		
		training steps < 3k	training steps < 9k	training steps < 14k
noise = *model(x,t)*		15.861	# todo	# todo
noise = *model(x)*	left	18.638	# todo	# todo
	middle	# todo	# todo	# todo
	right	# todo	# todo	# todo

	zero	111	# todo	# todo
noise,t = model(x)	left	15.765	**15.072**	**11.543**
	middle	17.432	15.278	# todo
	right	**15.671**	15.116	# todo
	zero	44	44	# todo

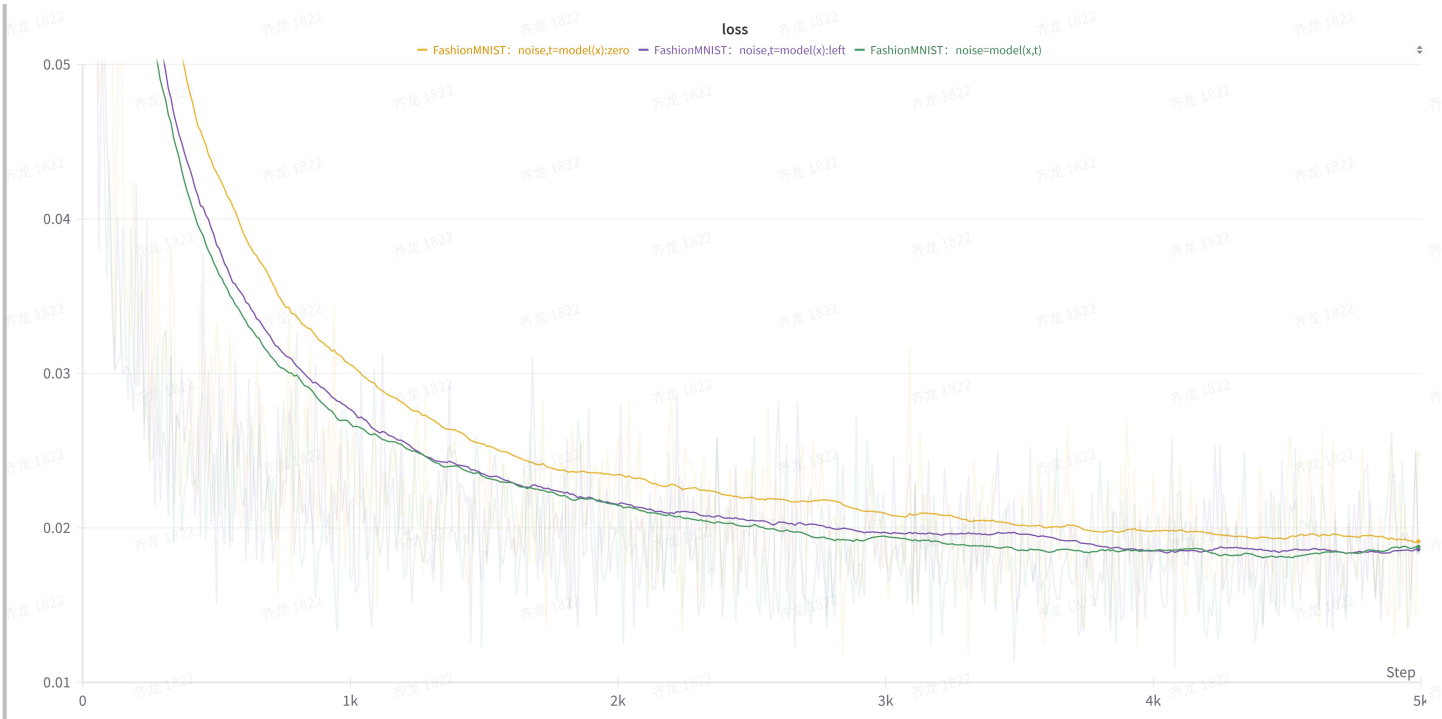
Figure 7

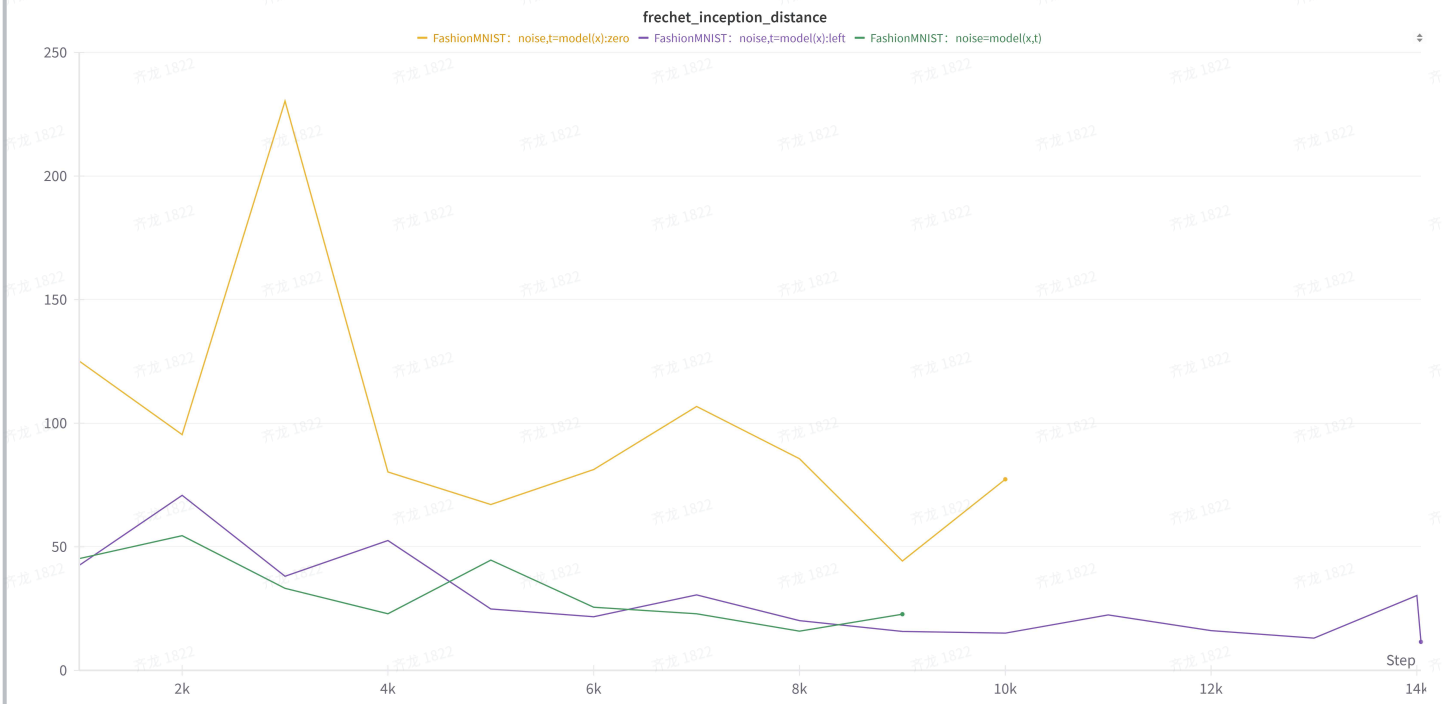
```
dataset: Fashion-MNIST
diffusion_schedule:"ddpm with linear_beta and timesteps is 1000"
batch_size:128
fixed_learning_rate:1e-3
number_of_images_for_FID:10000
```

Figure 8

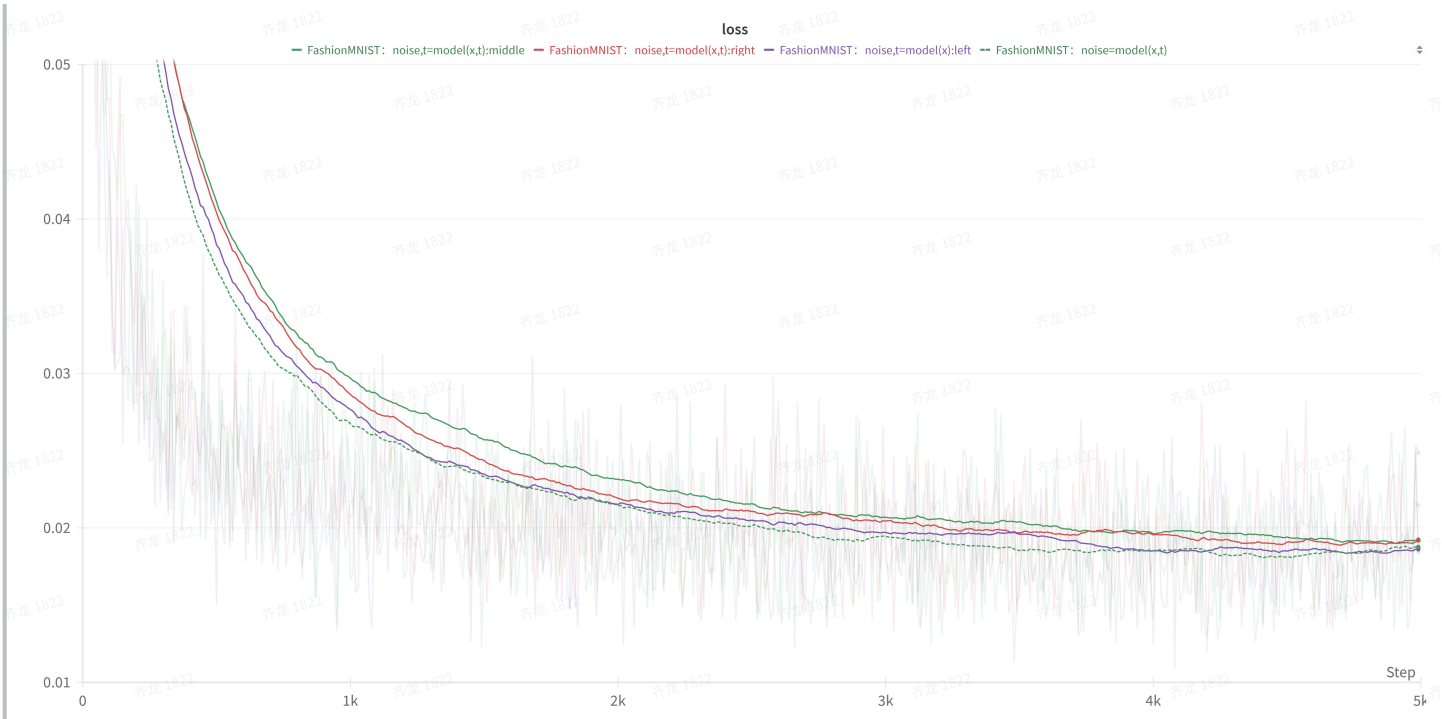
- A constant and relatively large learning rate （1e-3） can help mitigate performance differences caused by falling into different random local optima.
- In the $Zero$ case, ***random escape collapse*** phenomenon occurs, see Appendix A for details
- The $Left$, $Middle$ and $Right$ cases all work, indicating that the input of t is not necessary in the early and late stages, only in the middle stage as per the current study progress.
- For comparison, if T is reduced from 1000 to 300, both FID and training loss will show significant differences （#todo）.

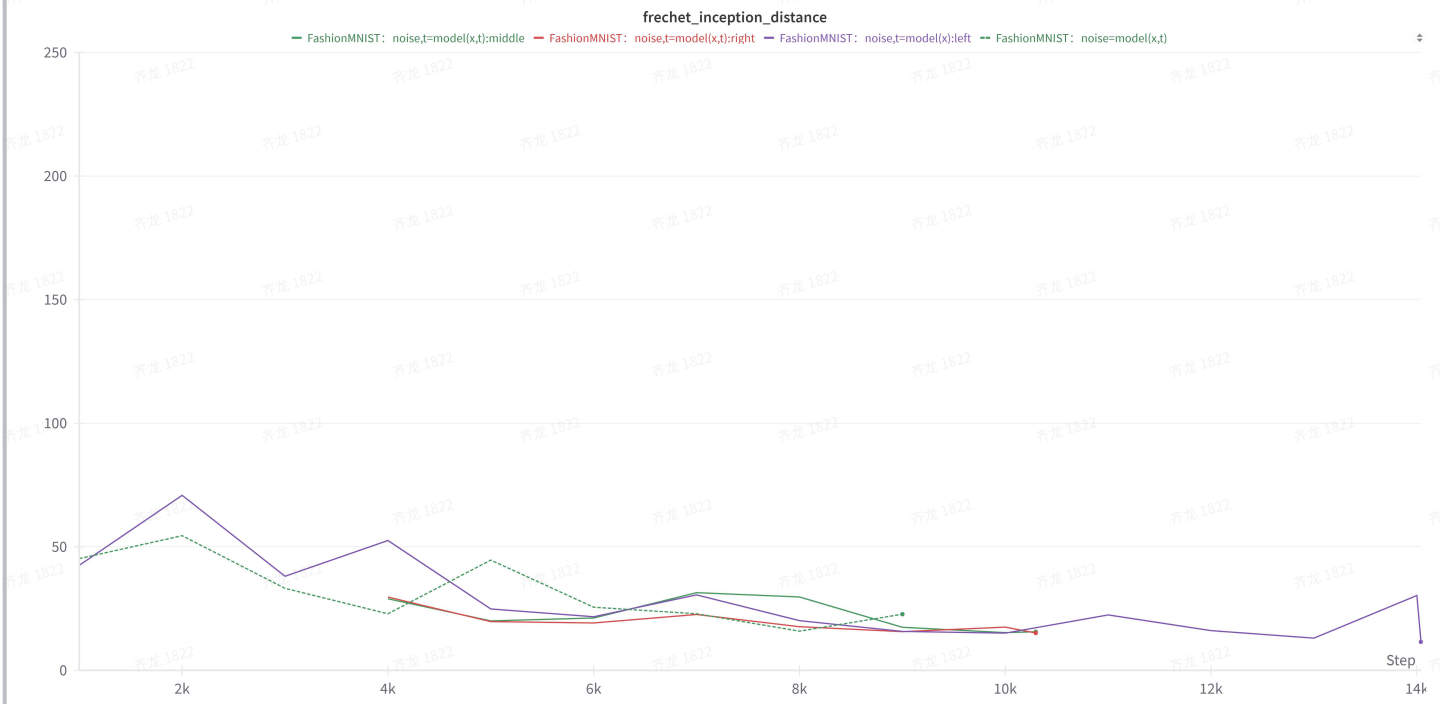
> Left vs Zero



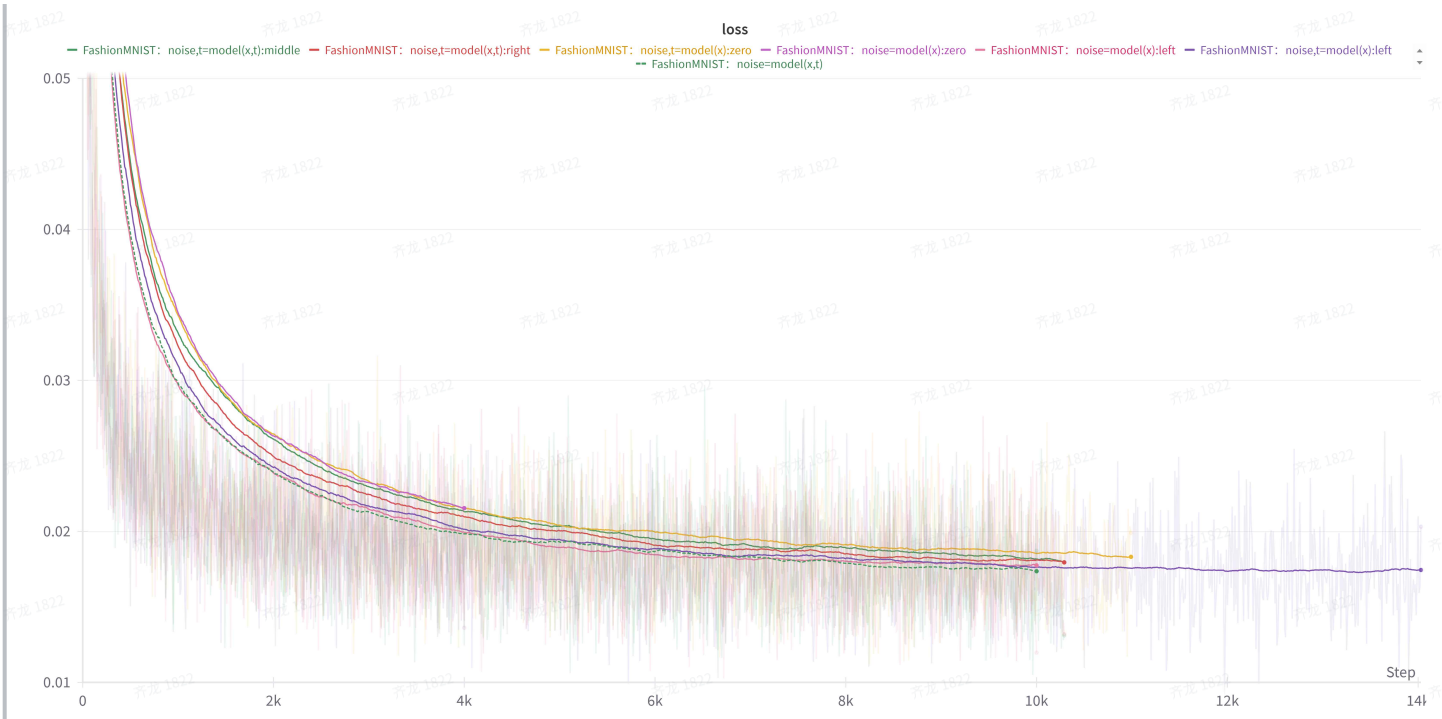


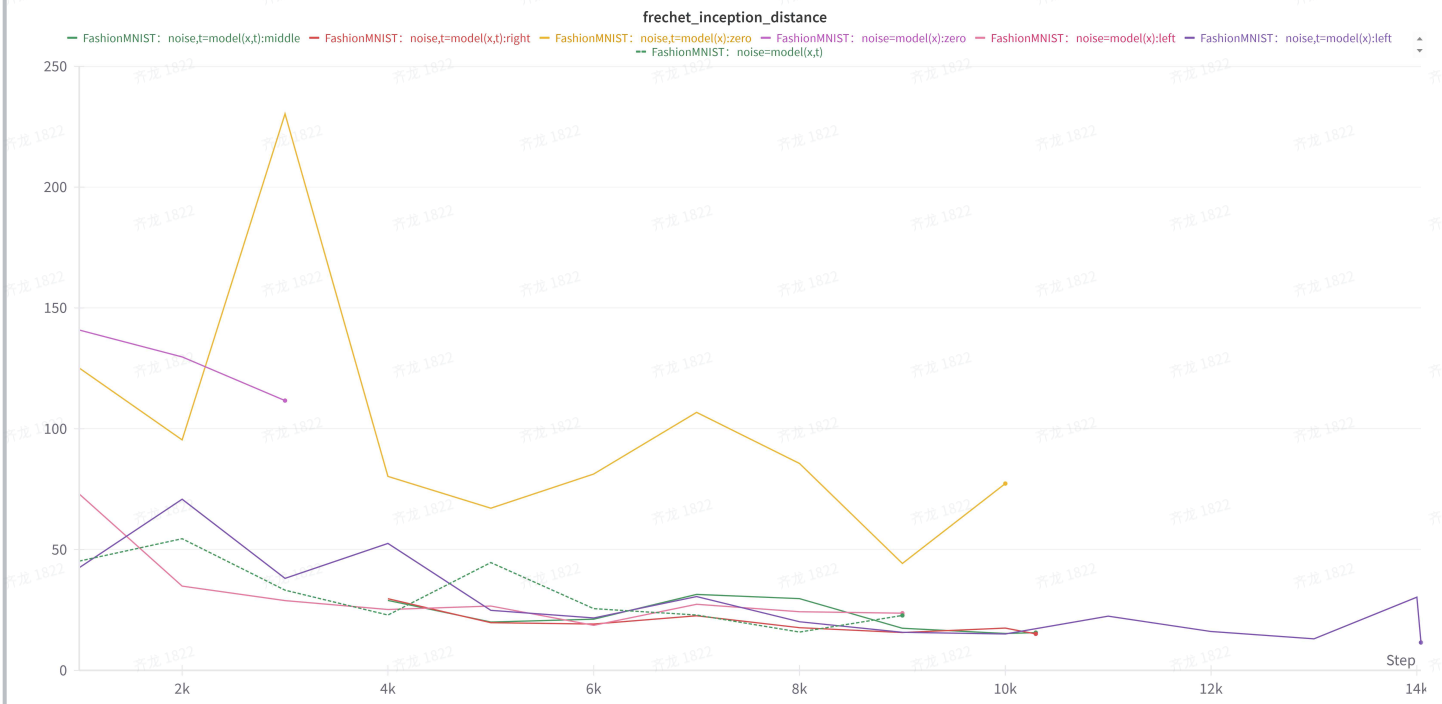
Left vs Middle vs Right

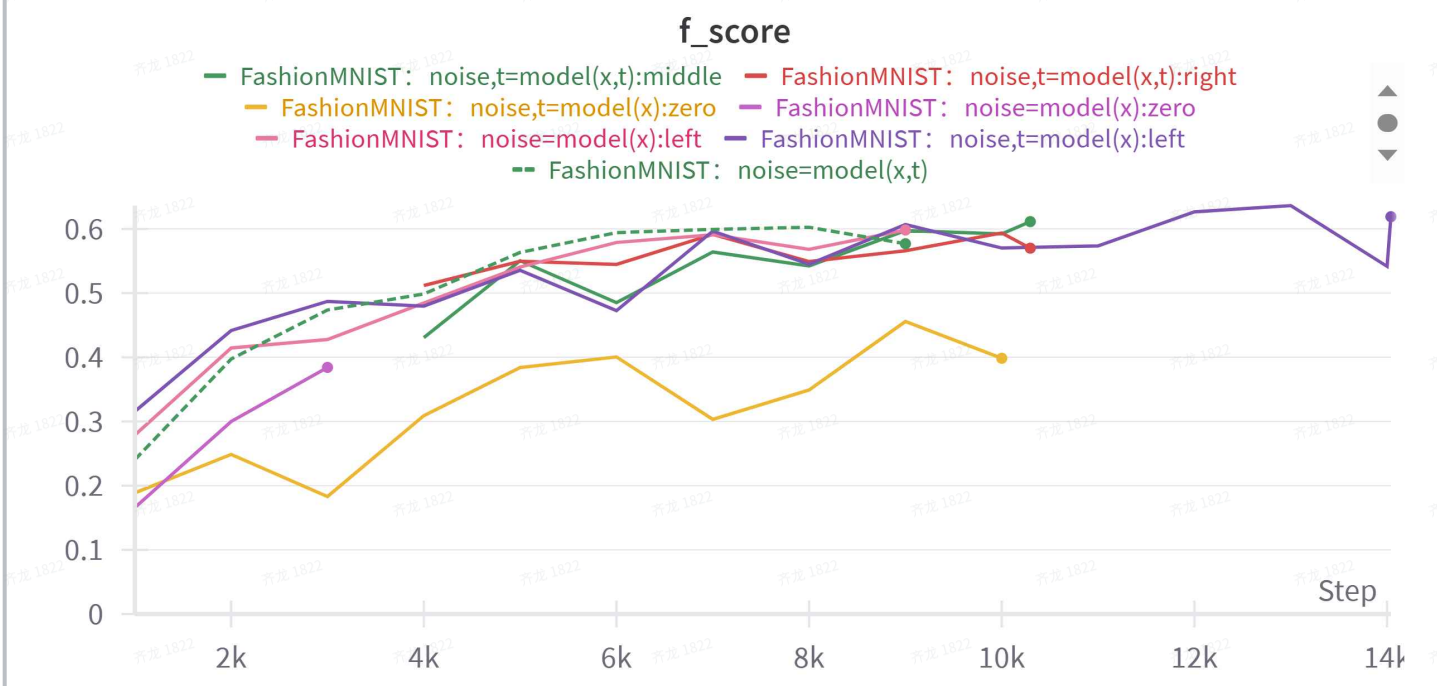

loss
FashionMNIST：noise,t=model(x,t):middle
FashionMNIST：noise,t=model(x,t):right
FashionMNIST：noise,t=model(x):left
FashionMNIST：noise=model(x,t)
0.05
0.04
0.03
0.02
0.01
0
1k
2k
3k
4k
5k
Step


frechet_inception_distance
FashionMNIST：noise,t=model(x,t):middle
FashionMNIST：noise,t=model(x,t):right
FashionMNIST：noise,t=model(x):left
FashionMNIST：noise=model(x,t)
250
200
150
100
50
0
2k
4k
6k
8k
10k
12k
14k
Step

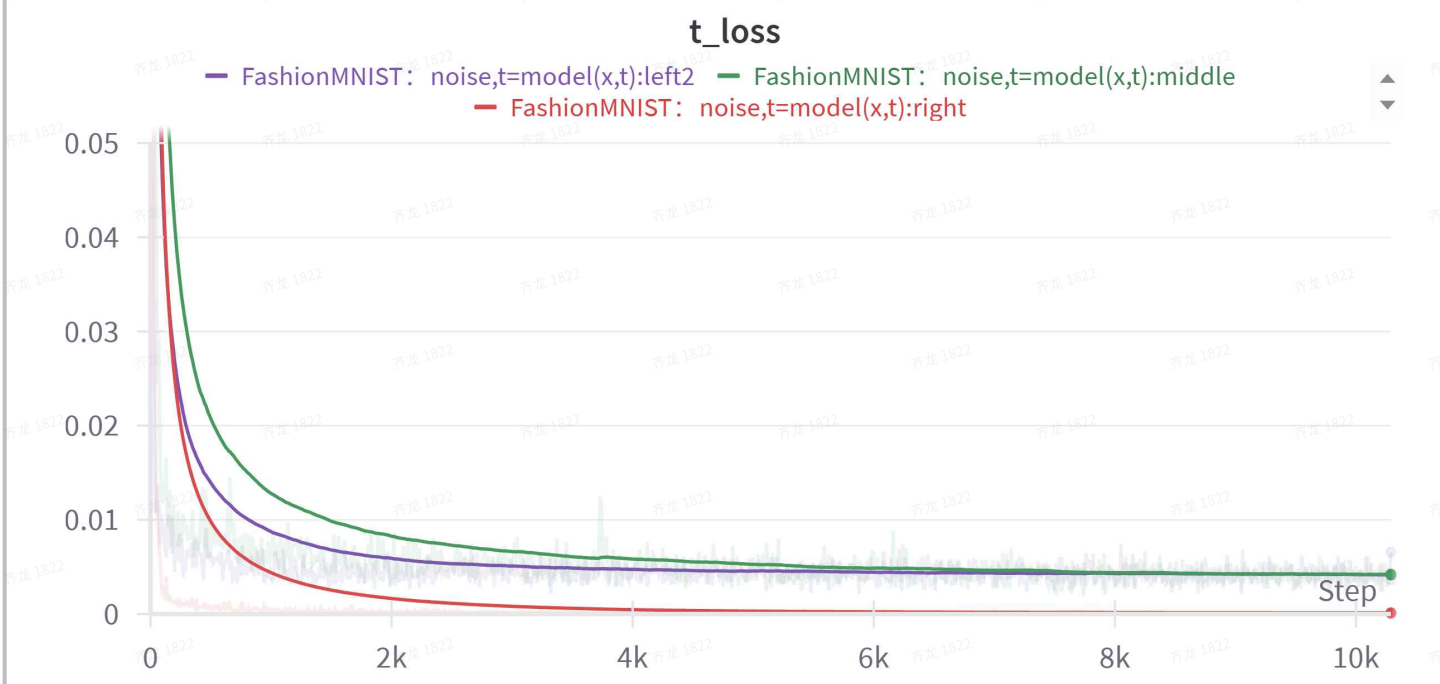
all


loss
FashionMNIST：noise,t=model(x,t):middle
FashionMNIST：noise,t=model(x,t):right
FashionMNIST：noise,t=model(x):zero
FashionMNIST：noise=model(x):zero
FashionMNIST：noise=model(x):left
FashionMNIST：noise,t=model(x):left
FashionMNIST：noise=model(x,t)
0.05
0.04
0.03
0.02
0.01
0
2k
4k
6k
8k
10k
12k
14k
Step

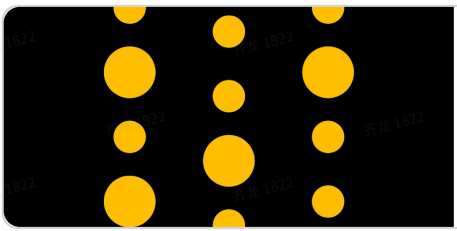

frechet_inception_distance
FashionMNIST：noise,t=model(x,t):middle
FashionMNIST：noise,t=model(x,t):right
FashionMNIST：noise,t=model(x):zero
FashionMNIST：noise=model(x):zero
FashionMNIST：noise=model(x):left
FashionMNIST：noise,t=model(x):left
FashionMNIST：noise=model(x,t)
250
200
150
100
50
0
2k
4k
6k
8k
10k
12k
14k
Step


f_score
FashionMNIST: noise,t=model(x,t):middle
FashionMNIST: noise,t=model(x,t):right
FashionMNIST: noise,t=model(x):zero
FashionMNIST: noise=model(x):zero
FashionMNIST: noise=model(x):left
FashionMNIST: noise,t=model(x):left
FashionMNIST: noise=model(x,t)
0.6
0.5
0.4
0.3
0.2
0.1
0
Step
2k
4k
6k
8k
10k
12k
14k

Predict t loss



More detail:



https://api.wandb.ai/links/fenneishi/etk7f6ke

TimeStep-Free diffusion model

Publish your model insights with interactive plots for performance metrics, predictions, and hyperparameters. Made by fenneishi using Weights & Biases

6. Q&&A

Overfitting?

> The reason why it works in the cases of partial t input could be due to overfitting caused by the simplicity of the dataset.

Yes, this possibility exists. However, the random escape collapse phenomenon observed in the $Zero$ case can partly refute the overfitting hypothesis. In the $Zero$ case, the model's output shows a clear dichotomy, with a certain probability of absolute failure (generating pure noise images) and a certain probability of success. This contrasts with the $Left$, $Middle$ and $Right$ cases, where at the early stages of training (training steps < 1000), all generated images are of low quality—neither complete failures (pure noise) nor successes. If the effectiveness of $Left$, $Middle$ and $Right$ were due to overfitting, then the $Zero$ case should exhibit similar behavior to the early stages of $Left$, $Middle$ and $Right$ training, rather than displaying the random escape collapse phenomenon (assuming the $Zero$ case is indeed more challenging, leading to training difficulties).

TFDM not better than TCDM?

> Based on the previous theoretical analysis, TFDM should be able to achieve better local optima and thus better performance. However, the experimental results show no significant performance difference between the two. What causes this?

The most likely reason is that the Fashion-MNIST dataset is too simple to highlight the architectural performance differences between the two models.

Additionally, apart from adding one line to control t_signal, TFDM's architecture is entirely based on the current TCDM. The current TCDM architecture is designed for timestep-conditioned scenarios and does not consider timestep-free situations. To fully unleash the performance potential of TFDM, more targeted improvements may be necessary, which is a task for future work.

100% no t ?

> The Zero case did not work, so what is the value of this research?

This is an ongoing study, not a finalized result. The purpose of sharing it is to inspire thought and discussion. And the author believes that, based on current progress, the random collapse phenomenon(see Appendix A for details) of Zero's escape can be solved, and it is not particularly difficult.

On the other hand, even if the random escape collapse phenomenon in the $Zero$ case cannot be solved, the potential benefits of TFDM discussed in the Motivation chapter are still partially realized. For example, in the $Right$ mode, when t is small (e.g., less than 30% of T), it is completely timestep-free. We can switch from timestep-conditioned to timestep-free when $t < 30\%T$ to gain potential benefits such as *Speed Up*, More Iterations, Higher Quality, Towards Better Local Optima, Decoupling and Schedule Free, and Model as First-class Citizen.

7. todo

1. 100% don't input t

Resolving the random escape collapse phenomenon(see Appendix A for details) in the Zero case.

2. large real-world high-resolution dataset

One possible reason why diffusion is more widely used than auto-regressive methods in the field of visual generation is that visual data has higher dimensionality compared to text. In other words, diffusion is better suited for high-dimensional data. Therefore, it is crucial to determine whether improvements to diffusion models still work effectively at high resolutions.

Additionally, Fashion-MNIST only has 60,000 training samples. While the random escape collapse phenomenon can partially support the argument that there is no overfitting, it still

cannot provide 100% proof. To definitively prove that overfitting is not an issue, the generative tasks must be extended to large and real-world datasets

3. SOTA

To prove that SFDM can indeed push the neural network to reach better local optima, it must achieve state-of-the-art performance.

4. Decoupling&&Schedule-Free and Model as First-Class Citizen

The current focus of the research is on verifying whether the model can function correctly without the input of t and accurately predict t . The next steps will build on this work to further achieve the Decoupling&&Schedule-Free and Model as First-Class Citizen concepts discussed in the Motivation section, rationally restructuring the existing diffusion architecture. The following is a continuously updated task list:

- Extend from discrete timesteps to continuous timesteps
- Extend from predicting t to predicting the inverse signal-to-noise ratio (nsr)
- Extend from diffusion schedule to NSR schedule
- ...

APPENDIX

A R*andom Escape Collapse Phenomenon*

What is *Escape Collapse Phenomenon* ?

Samples 1

t_signal: $Left$

Samples 2

t_signal: $Zero$

Samples 3

t_signal: $Zero$

Samples 4

t_signal: $Left$

training_step:300× training_step:900 training_step:9000 training_step:9000

Figure 9

- Samples 1 and 2 achieved similar loss and FID, but for different reasons. Figure 1's generated images have similar quality, while Figure 2 shows significant random collapse phenomena, leading to a polarization of image quality.
- Both Samples 2 and 3 are in $Zero$ mode. Comparing them reveals that although the random collapse phenomenon is alleviated with more training, it does not disappear.
- Comparing Samples 3 and 4 shows that, after excluding randomly collapsed images, $Zero$ and $Left$ can achieve comparable generation quality with sufficient training. In other words, as long as the random collapse phenomenon is addressed, a timestep-free model can be achieved 100%.

We refer to this randomly occurring sampling collapse phenomenon, which does not disappear with increased training and only appears in the Zero case, as ***Random Escape Collapse***.

What causes Random Escape Collapse?

Since Random Escape Collapse only occurs in Zero mode and does not appear in $Left$, $Middle$, or $Right$ cases, it can be reasonably inferred that Random Escape Collapse is not related to t values close to 0 or T, but is instead related to t values close to $T/2$. The closer it is to $T/2$, the higher the correlation.

The following is a hypothesis regarding the possible cause of the Random Escape Collapse phenomenon—***the random term*** $+\sigma_t \cdot \mathcal{N}(0,1)$ ***causes*** $\mathbf{x}_t$ ***to escape from the denoising probability flow.***

Below is the denoising iteration formula in the Zero case. Except for replacing $\epsilon_\theta(\mathbf{x}_t, t)$ with $\epsilon_\theta(\mathbf{x}_t)$, everything else is consistent with the original DDPM.

$$\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(\mathbf{x}_t)\right) + \sigma_t \cdot \mathcal{N}(0,1)$$

Simplified to:

$$\mathbf{x}_{t-1} = \mathbf{x}_{t-1}^{predict} + \sigma_t \cdot \mathcal{N}(0,1)$$

From the above equation, it can be seen that the term $+\sigma_t \cdot \mathcal{N}(0,1)$ creates a pulling force that draws $\mathbf{x}_{t-1}^{predict}$ towards $\mathcal{N}(0,1)$, which is the starting point of the denoising process—— $\mathbf{x}_t$.

Next, we will analyze this pulling force in three different cases.

- When t is relatively large, that is, close to T, $\mathbf{x}_{t-1}^{predict}$ itself is already close to $\mathcal{N}(0,1)$, so there is no significant effect.
- When t is relatively small, that is, close to 1, σ_t is close to $zero$, so the pulling force towards $\mathcal{N}(0,1)$ is negligible.

- However, when t is close to $T/2$, $\mathbf{x}_{t-1}^{predict}$ is no longer close to $\mathcal{N}(0,1)$, and σ_t is no longer close to $zero$. The pulling force towards $\mathcal{N}(0,1)$ cannot be ignored. ***The term*** $+\sigma_t \cdot \mathcal{N}(0,1)$ ***causes*** $\mathbf{x}_{t-1}$ ***to have a higher degree of overlap with*** $\mathcal{N}(0,1)$ ***compared to*** $\mathbf{x}_{t-1}^{predict}$. Then, when using the formula $\mathbf{x}_{t-2} = \frac{1}{\sqrt{\alpha_{t-1}}}\left(\mathbf{x}_{t-1} - \frac{1-\alpha_{t-1}}{\sqrt{1-\bar{\alpha}_{t-1}}}\epsilon_\theta(\mathbf{x}_{t-1})\right) + \sigma_{t-1} \cdot \mathcal{N}(0,1)$ to iteratively solve for $\mathbf{x}_{t-2}$, because $\mathbf{x}_{t-1}$ has a higher degree of overlap with $\mathcal{N}(0,1)$ compared to $\mathbf{x}_{t-1}^{predict}$, ***the neural network*** ϵ_θ ***might incorrectly assume the input is*** $\mathbf{x}_{t-s}^{predict}$ ***(where*** $1 < s$ ***) instead of*** $\mathbf{x}_{t-1}^{predict}$, leading to erroneous results and causing the Random Escape Collapse phenomenon.

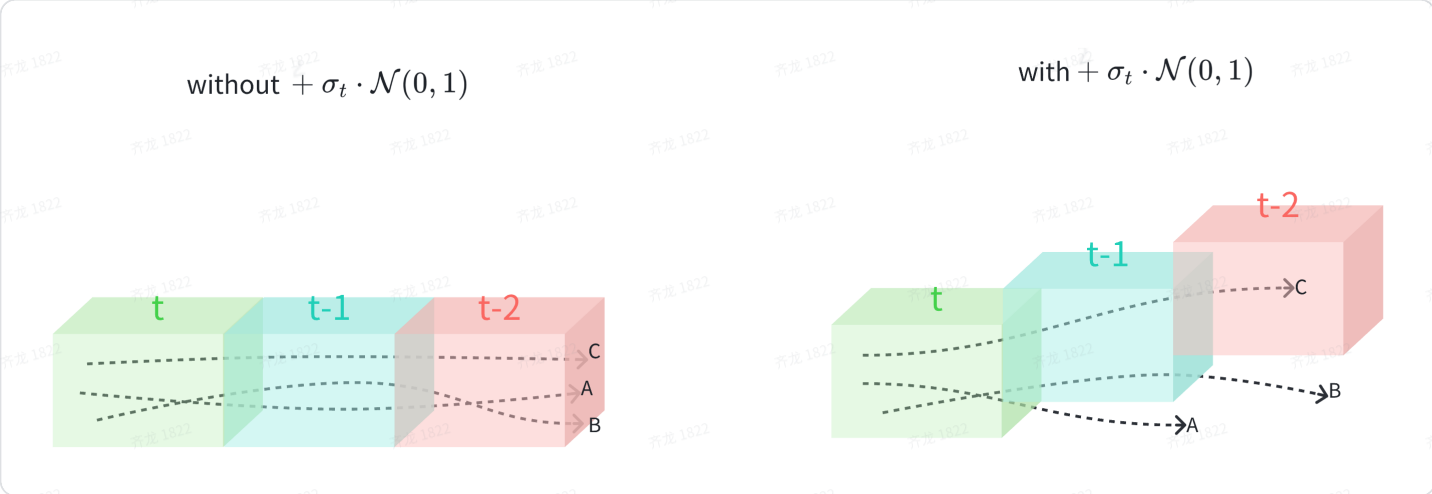


Figure 11: randomly escape from the denoising probability flow pipe

But For TCDM:

$$\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(\mathbf{x}_t, t)\right) + \sigma_t \cdot \mathcal{N}(0,1)$$

$$\mathbf{x}_{t-2} = \frac{1}{\sqrt{\alpha_{t-1}}}\left(\mathbf{x}_{t-1} - \frac{1-\alpha_{t-1}}{\sqrt{1-\bar{\alpha}_{t-1}}}\epsilon_\theta(\mathbf{x}_{t-1}, t-1)\right) + \sigma_{t-1} \cdot \mathcal{N}(0,1)$$

Since t is provided as an input to the model, and the input method of t is as follows(conditional instance normalization):

```
x_t = self.norm(x_t)
scale, shift = self.time_mlp(time)
x_t = (scale + 1) * x_t  + shift
```

Figure 10

The actual input to the $Unet$ becomes $(scale + 1) * x_t + shift$ from x_t. Different values of t have different scales and shifts, leading to different distributions. In other words, ***by input*** t ***, the distributions of*** x_t ***at different t values are forcibly dispersed.*** This reduces or mitigates

the impact of $+\sigma_t \cdot \mathcal{N}(0,1)$ on increasing overlap, significantly lowering the probability of random escape.

The following series of images visualize the distribution of x_t at different scales when x_0 is two-dimensional. It is evident that as the scale increases, the distributions of x_t are clearly separated.

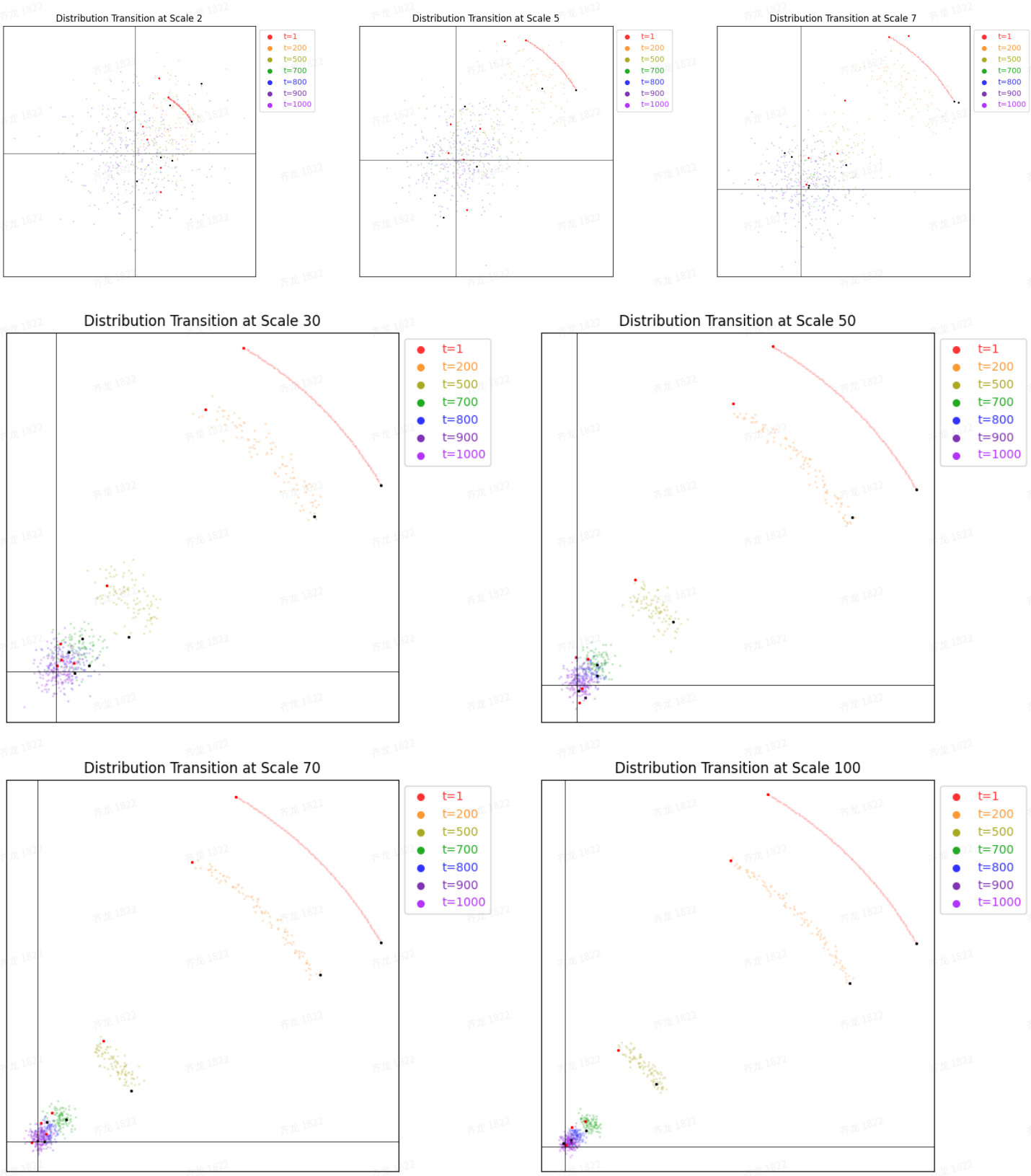


Figure 12

> When you try to delete a common & base module of a system, you will discover certain truths.

B. Decoupling

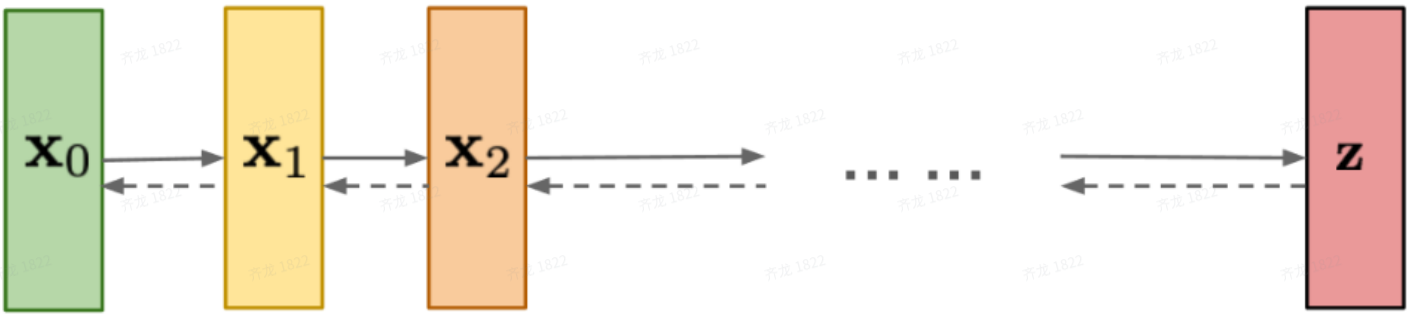


Figure 13

The diffusion-based method can generally be divided into two processes: the Forward Diffusion Process (hereinafter referred to as the ***AddNoise Process***) and the Reverse Diffusion Process (hereinafter referred to as the ***DeNoise Process***). The AddNoise Process is used to obtain pairs of noisy images for training neural network, while the **DeNoise Process** is used to employ the well trained neural network for inference (i.e., image generation).

In the AddNoise Process, the ***Diffusion-Schedule*** is closely coupled with a ***Noise-Generator*** (i.e., torch.randn(x_0)), forming a stateful ***AddNoiser***. By recursively running this AddNoiser, noisy images at all timesteps are obtained (during actual training, we can sample x_t at any arbitrary timestep t in a closed form), as follows:

$$AddNoiser = DiffusionSchedule \oplus NoiseGenerator$$

$$AddNoise\ Process = AddNoiser(AddNoiser(AddNoiser(\cdots AddNoiser(x_0)\cdots)))$$

In the DeNoise Process, the ***Diffusion-Schedule*** is closely coupled with a ***Noise-Generator*** (i.e., the well trained neural network), forming a stateful ***DeNoiser***. By recursively running this DeNoiser, noisy images at all timesteps are obtained.

$$DeNoiser = DiffusionSchedule \oplus NeuralNetwork$$

$$DenoiseProcess = DeNoiser(DeNoiser(DeNoiser(\cdots DeNoiser(x_T)\cdots)))$$

In the DeNoiser, the Diffusion Schedule is tightly coupled with the neural network to accomplish the denoising task. How do they cooperate, and what are their respective roles? Is it possible to conceptually decouple them as much as possible from an appropriate perspective?

The same logic applies to the AddNoiser.

> To decouple a production system, a good approach is to first decouple the outputs of the entire system. For example, if we want to understand the different departments within Apple,

> we can categorize the iPhone into hardware and software components, allowing us to reasonably infer that Apple is divided into hardware and software departments.

We can start by decoupling the output of the DeNoiser （or AddNoiser）, that is, by decoupling $\mathbf{x}_t$. If $\mathbf{x}_t$ is an image, we can split it by channels, such as the RGB channels, or by height and width, similar to the patch method, dividing the image into multiple blocks. These are common decoupling methods in the field of deep learning for vision. Now, let's consider a different perspective, as follows:

$$\mathbf{x}_t = \sigma_{\mathbf{x}_t}(\bar{\mathbf{x}}_t + \tilde{\mathbf{x}}_t)$$

Here, $\sigma_{\mathbf{x}_t}$ is the standard deviation of $\mathbf{x}_t$, and $\bar{\mathbf{x}}_t$ is the mean of $\mathbf{x}_t$. The calculations for $\sigma_{\mathbf{x}_t}$, $\bar{\mathbf{x}}_t$,and $\tilde{\mathbf{x}}_t$ can be referenced from the following PyTorch code:

```
std = img.std()
mean = img.mean()
x_tilde = (img - mean) / std
```

Through the above formulas, we decompose $\mathbf{x}_t$ into "*standard deviation*", "*mean*" and "*fluctuations (or patterns) around the mean within a unit standard deviation*" corresponding to $\sigma_{\mathbf{x}_t}$, $\bar{\mathbf{x}}_t$ and $\tilde{\mathbf{x}}_t$ respectively.

Here, we present a hypothesis without proof:

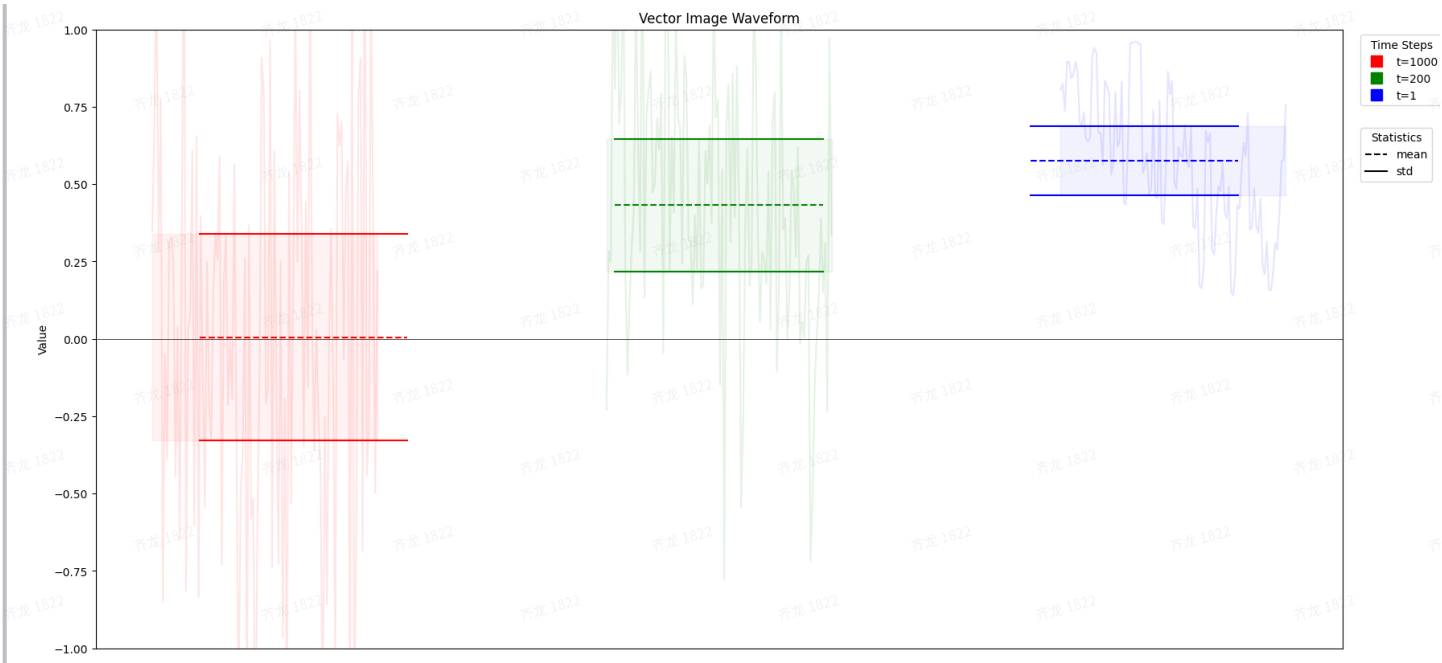
All the information of $\mathbf{x}_t$ is stored in $\tilde{\mathbf{x}}_t$, and is independent of $\sigma_{\mathbf{x}_t}$ and $\bar{\mathbf{x}}_t$.

Based on this hypothesis, we can further decompose $\mathbf{x}_t$ as:

$$\mathbf{x}_t = \sigma_{\mathbf{x}_t}(\bar{\mathbf{x}}_t + infomation_t)$$

We can further visualize this using the following code:

$$\mathbf{x}_t = \sigma_{\mathbf{x}_t}(\bar{\mathbf{x}}_t + infomation_t)$$



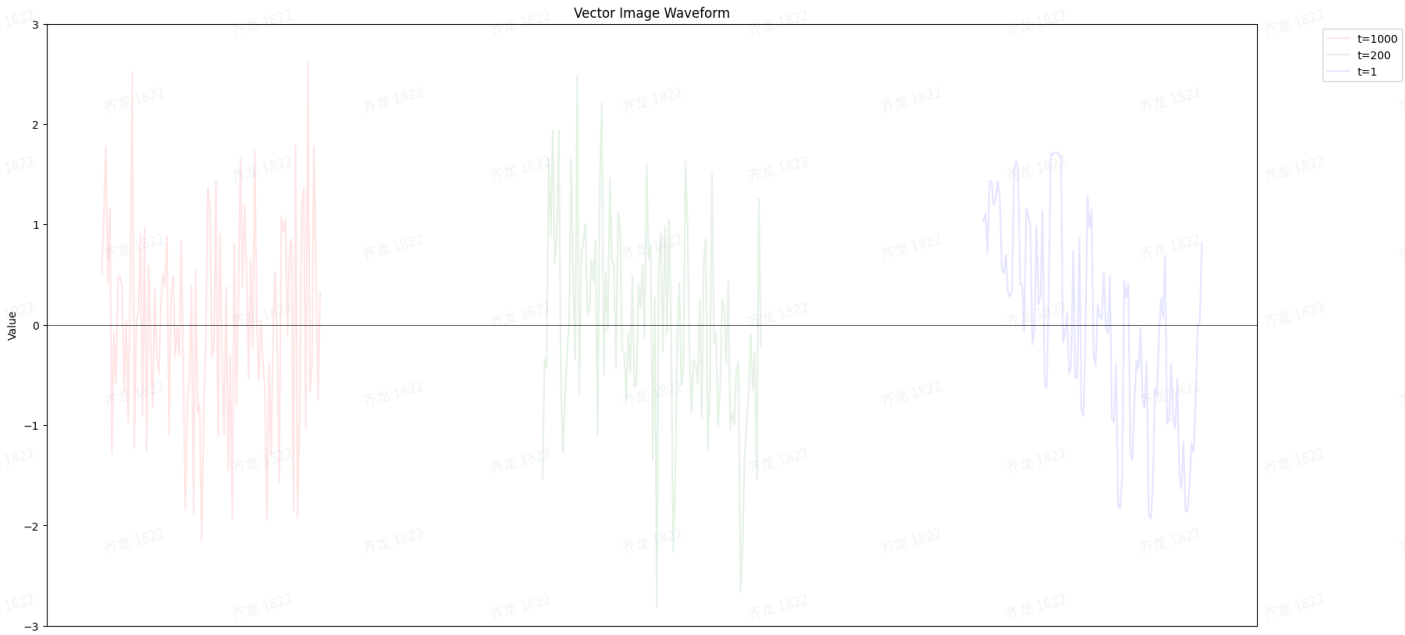

```
T = 1000
x_0 = PIL.Image.Image.open('test.png')
for t in zip([T, int(T // 5), 1])
    img = q_sample(x_0,t).flatten()
    mean,std = img.mean(),img.std()
    plt.plot(vec_img)
    plt.axhline(y=mean,linestyle="--")
    plt.axhline(y=std,linestyle="solid")
```

$$\mathbf{x}_t = \sigma_{\mathbf{x}_t}(\bar{\mathbf{x}}_t + infomation_t)$$


```
T = 1000
x_0 = PIL.Image.Image.open('test.png')
```

```
for t in zip([T, int(T // 5), 1])
    img = q_sample(x_0,t).flatten()
    mean,std = img.mean(),img.std()
    # plt.plot(vec_img)
    plt.axhline(y=mean,linestyle="--")
    plt.axhline(y=std,linestyle="solid")
```

$$\mathbf{x}_t = \sigma_{\mathbf{x}_t}(\bar{\mathbf{x}}_t + information_t)$$




```
T = 1000
x_0 = PIL.Image.Image.open('test.png')
for t in zip([T, int(T // 5), 1])
    img = q_sample(x_0,t).flatten()
    mean,std = img.mean(),img.std()
    plt.plot((vec_img - mean) / std)
    # plt.axhline(y=mean,linestyle="--")
    # plt.axhline(y=std,linestyle="solid")
```

Figure 14

Based on the decomposition of $\mathbf{x}_t$, we will now decompose the AddNoiser. In DDPM, the original definition of AddNoiser is as follows:

$$\mathbf{x}_t = \sqrt{\alpha_t}\mathbf{x}_{t-1} + \sqrt{1-\alpha_t}\boldsymbol{\epsilon}_{t-1}$$

After some simplification, we get:

$$\mathbf{x}_t = \sqrt{\alpha_t}(\mathbf{x}_{t-1} + \frac{\sqrt{1-\alpha_t}}{\sqrt{\alpha_t}}\boldsymbol{\epsilon}_{t-1})$$

Substituting the decomposition of $\mathbf{x}_t$

$$\mathbf{x}_t = \sqrt{\alpha_t}\left[\sigma_{\mathbf{x}_{t-1}}(\bar{\mathbf{x}}_{t-1}+\tilde{\mathbf{x}}_{t-1}) + \frac{\sqrt{1-\alpha_t}}{\sqrt{\alpha_t}}\boldsymbol{\epsilon}_{t-1}\right]$$

$$\mathbf{x}_t = \sqrt{\alpha_t}\sigma_{\mathbf{x}_{t-1}}\left[(\bar{\mathbf{x}}_{t-1}+\tilde{\mathbf{x}}_{t-1}) + \frac{\sqrt{1-\alpha_t}}{\sqrt{\alpha_t}\sigma_{\mathbf{x}_{t-1}}}\boldsymbol{\epsilon}_{t-1}\right]$$

Let $\sqrt{\alpha_t}\sigma_{\mathbf{x}_{t-1}}$ be denoted as S, and let $\frac{\sqrt{1-\alpha_t}}{\sqrt{\alpha_t}\sigma_{\mathbf{x}_{t-1}}}$ be denoted as K. The resulting equation is as follows. Note that S and K are not only related to α_t (i.e., the Diffusion Schedule), but also to $\sigma_{\mathbf{x}_{t-1}}$, which depends on both the Diffusion Schedule and the dataset. Therefore, S and K depend on both the Diffusion Schedule and the dataset.

$$\mathbf{x}_t = S\left[\bar{\mathbf{x}}_{t-1}+\tilde{\mathbf{x}}_{t-1}+K\boldsymbol{\epsilon}_{t-1}\right]$$

For greater semantic clarity, it is further simplified to:

$$\mathbf{x}_t = S\cdot\left[\bar{\mathbf{x}}_{t-1}+(\mathbf{information}_{t-1}+K\cdot\boldsymbol{noise})\right]$$

In the above equation, the purple part represents the AddNoiser (more accurately, AddNoiser * $\sigma_{\mathbf{x}_t}$).

$$\mathbf{x}_t = S\cdot\left[\bar{\mathbf{x}}_{t-1}+(\mathbf{information}_{t-1}+K\cdot\boldsymbol{noise})\right]$$

Further,with the dataset fixed, the blue part represents the Diffusion Schedule, and the green part represents the noise generator.

Similarly, the DeNoiser can be decomposed as follows:

$$\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left[\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t,t)\right]$$

$$\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left[\sigma_{\mathbf{x}_t}(\bar{\mathbf{x}}_t+\tilde{\mathbf{x}}_t) - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t,t)\right]$$

$$\mathbf{x}_{t-1} = \frac{\sigma_{\mathbf{x}_t}}{\sqrt{\alpha_t}}\left[\bar{\mathbf{x}}_t+\tilde{\mathbf{x}}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}\sigma_{\mathbf{x}_t}}\boldsymbol{\epsilon}_\theta(\mathbf{x}_t,t)\right]$$

$$\mathbf{x}_{t-1} = S^{-1}\left[\bar{\mathbf{x}}_t+\tilde{\mathbf{x}}_t+K^{-1}(-\boldsymbol{\epsilon}_\theta(\mathbf{x}_t,t))\right]$$

$$\mathbf{x}_{t-1} = S^{-1}\cdot\left[\bar{\mathbf{x}}_t+(\mathbf{information}_t+K^{-1}\cdot\boldsymbol{\Delta information})\right]$$

The purple part represents the DeNoiser,

$$\mathbf{x}_{t-1} = S^{-1}\cdot\left[\bar{\mathbf{x}}_t+(\mathbf{information}_t+K^{-1}\cdot\boldsymbol{\Delta information})\right]$$

The blue part represents the Diffusion Schedule, and the green part represents the noise predictor, or information generator, which is the Neural Network.

To summarize, the above analysis mathematically decomposes the AddNoise Process into recursive calls to the AddNoiser and further decouples the AddNoiser into the Diffusion Schedule and the noise generator, placing them in a single equation to clarify their interrelationships. The same logic applies to the DeNoise Process and decomposition results of the AddNoiser and DeNoiser are symmetrical and mutually corroborative.

Simply put, the result of the decomposition is that the Diffusion Schedule is responsible for scaling, while the information generator (Neural Network) and noise generator handle the addition and removal of information.

In TFDM, the situation is completely different for the DeNoiser, because TFDM can predict t during the inference phase without requiring t as input. Therefore, the DeNoiser can intelligently infer its state without needing to obtain its state from the Diffusion Schedule. Simply put, ***the DeNoiser does not rely on the Diffusion Schedule and is instead 100% determined by the information generator (Neural Network).***

$$\mathbf{x}_{t-1} = S^{-1} \cdot \left[\bar{\mathbf{x}}_t + (\mathbf{information}_t + K^{-1} \cdot \mathbf{\Delta information})\right]$$

Of course, there is a remaining question: What is the information-theoretic meaning of S^{-1} and k^{-1} ? In the absence of a Schedule, how can we determine S^{-1} and k^{-1} from the predicted t?

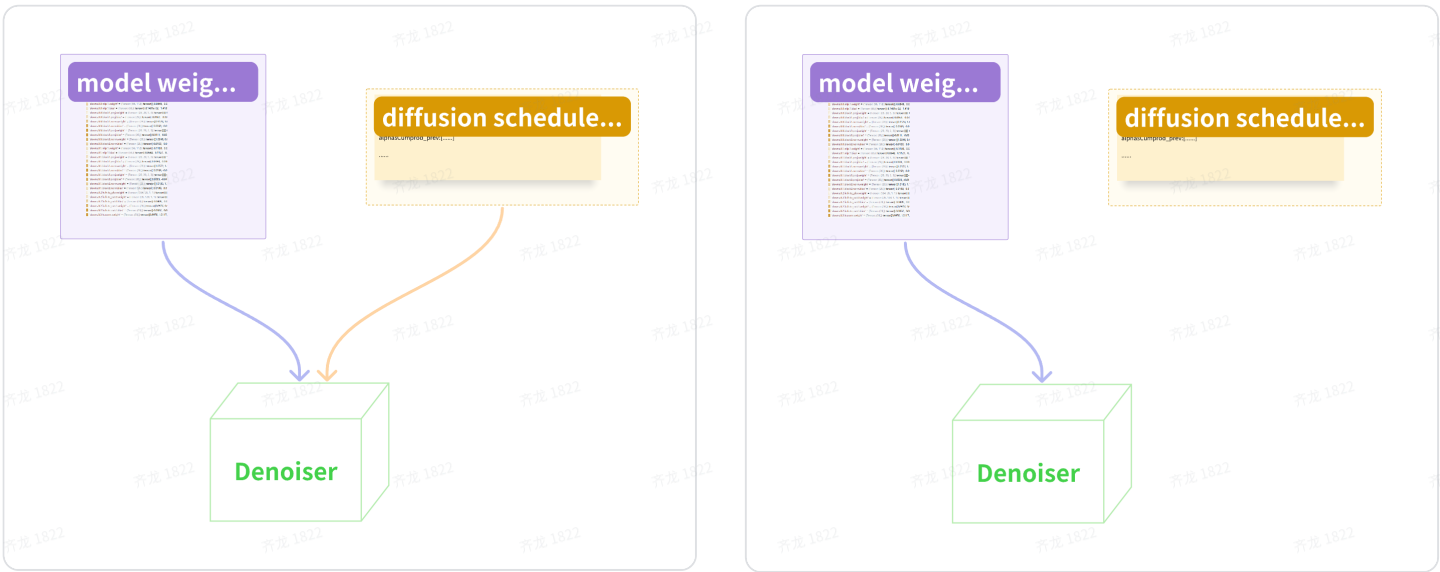


Figure 15