Deep Unsupervised Learning using Nonequilibrium Thermodynamics

Team **SVM (61)**
By
Sudipta Halder (2021202011)
Vishal Pandey (2021201070)
Manish Raushan (2021801005)

Paper Name and Link

Deep Unsupervised Learning using Nonequilibrium Thermodynamics

https://arxiv.org/pdf/1503.03585.pdf

Code GitHub Link

— https://github.com/RajaSudipta/SMAI-Project

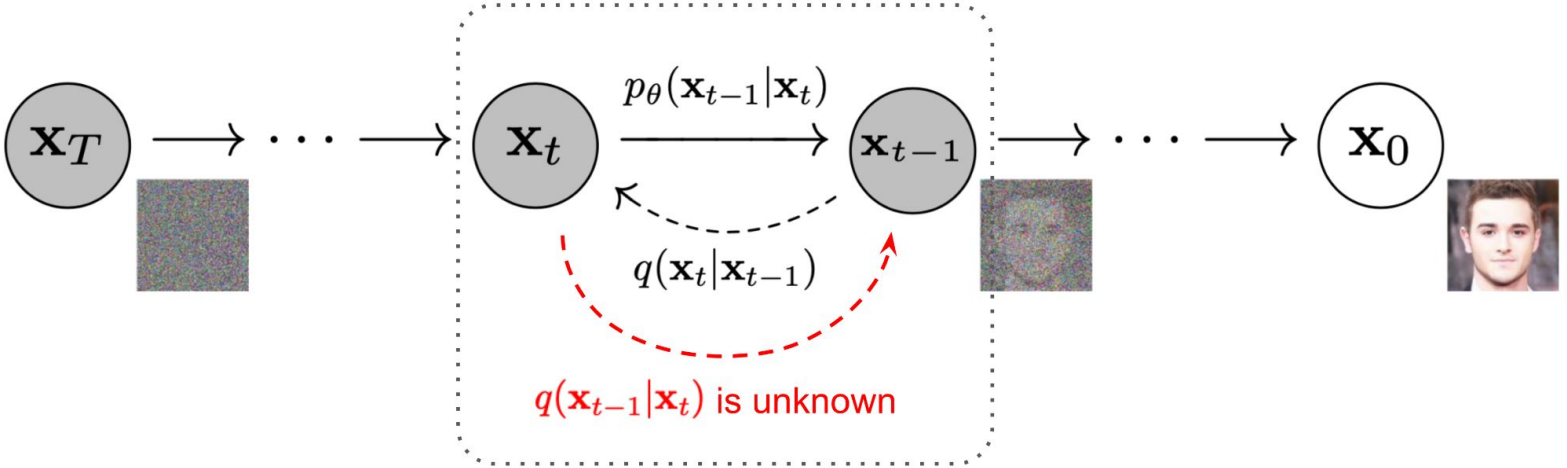
How to Interpret Diffusion Models in AI?

- Diffusion models are deep generative models that work by adding noise (Gaussian noise) to the available training data (also known as the forward diffusion process) and then reversing the process (known as denoising or the reverse diffusion process) to recover the data.

- The model gradually learns to remove the noise.

- This learned denoising process generates new, high-quality images from random seeds (random noised images), as shown in the illustration below.

Use variational lower bound

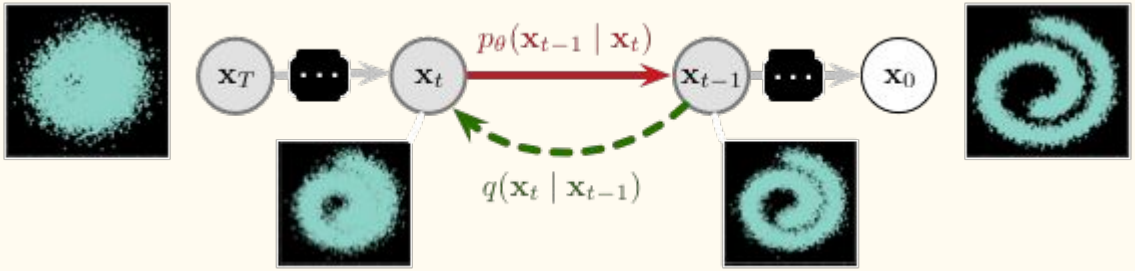


Motivation

- DALL-E 2
- Stability AI

How does it work??

- These models are based on two reciprocal processes that represent two Markov chains of random variables.
- One process $q(x_t \mid x_{t-1})$ that gradually adds noise to the input data (called the diffusion or forward process), destroying the signal up to full noise.
- In the opposite direction, the reverse process $p(x_{t-1} \mid x_t)$ tries to learn how to invert this diffusion process (transform random noise into a high-quality waveform).



- As we can see, the forward (and fixed) process q(xt | xt-1) gradually introduces noise at each step.
- Oppositely, the reverse (parametric) process p(xt-1 | xt) must learn how to denoise local perturbations.
- Hence, learning involves estimating a large number of small perturbations, which is more tractable than trying to directly estimate the full distribution with a single potential function.
- Both processes can be defined as parametrized Markov chains

Formalization

- Diffusion models are based on a series of latent variables x_1, x_2, ... , x_T that have the same dimensionality as a given input data, which is labeled as $x_0 \sim q(x_0)$.
- Then, we need to define the behavior of two process
  - Forward diffusion : $q(x_t \mid x_{t-1})$
  - Reverse Diffusion : $p(x_{t-1} \mid x_t)$

Forward process

In the forward process, the data distribution $q(\mathbf{x}_0)$ is gradually converted into an analytically tractable distribution $\pi(\mathbf{y})$, by repeated application of a Markov diffusion kernel $T_\pi(\mathbf{y} \mid \mathbf{y}'; \beta)$, with a given diffusion rate β.

$$q(\mathbf{x}_t \mid \mathbf{x}_{t-1}) = T_\pi(\mathbf{x}_t \mid \mathbf{x}_{t-1}; \beta_t)$$

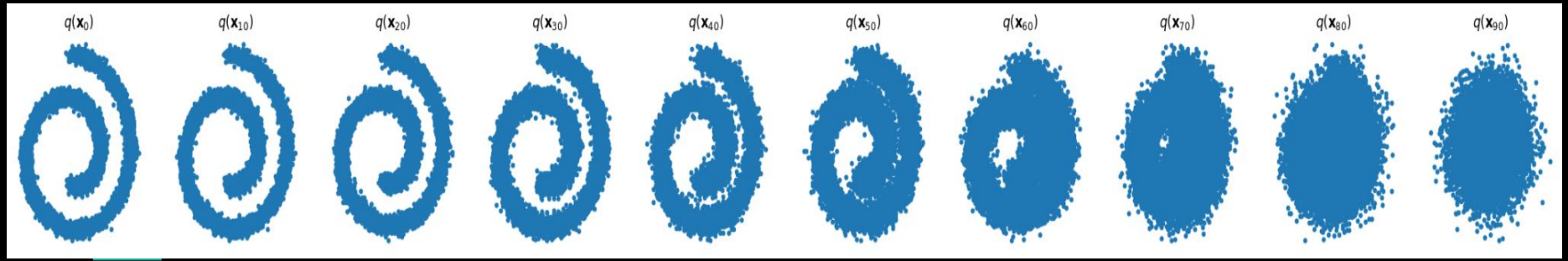
This diffusion kernel can be set to gradually inject Gaussian noise, given a variance schedule $\beta_1, \cdots, \beta_T$ such that

$$q(\mathbf{x}_t \mid \mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I})$$

The complete distribution $q(\mathbf{x}_{0:T})$ is called the *diffusion* process and is defined as

$$q(\mathbf{x}_{0:T}) = q(\mathbf{x}_0) \prod_{t=1}^{T} q(\mathbf{x}_t \mid \mathbf{x}_{t-1})$$

Forward process of adding noise



$$\alpha_t = 1 - \beta_t$$

$$\bar{\alpha}_t = \prod_{s=1}^{t} a_s$$

$$\begin{aligned}
q(x_t|x_{t-1}) &= \mathcal{N}(x_t, \sqrt{1-\beta_t}x_{t-1}, \beta_t I) \\
&= \sqrt{1-\beta_t}x_{t-1} + \sqrt{\beta_t}\epsilon \\
&= \sqrt{\alpha_t}\, x_{t-1} + \sqrt{1-\alpha_t}\, \epsilon \\
&= \sqrt{\alpha_t\alpha_{t-1}}\, x_{t-2} + \sqrt{1-\alpha_t\alpha_{t-1}}\, \epsilon \\
&= \sqrt{\alpha_t\alpha_{t-1}\alpha_{t-2}}\, x_{t-3} + \sqrt{1-\alpha_t\alpha_{t-1}\alpha_{t-2}}\, \epsilon \\
&= \sqrt{\alpha_t\alpha_{t-1}\ldots\alpha_1\alpha_0}\, x_0 + \sqrt{1-\alpha_t\alpha_{t-1}\ldots\alpha_1\alpha_0}\, \epsilon \\
&\boxed{= \sqrt{\bar{\alpha}_t}\, x_0 + \sqrt{1-\bar{\alpha}_t}\, \epsilon}
\end{aligned}$$

- No need to go like x_0 -> x_1 -> x_2 -> ... x_T. Directly we can go from x_0 -> x_T using this equation

Notably, the forward process posteriors are tractable when conditioned on $\mathbf{x}_0$

$$q(\mathbf{x}_t \mid \mathbf{x}_{t-1}) = \mathcal{N}(\mathbf{x}_{t-1}; \mathbf{1} - \beta_t \mathbf{x}_{t-1}, \beta_t \mathbf{I})$$

And we can obtain the corresponding mean $\tilde{\mu}_t(\mathbf{x}_t, \mathbf{x}_0)$ and variance $\tilde{\beta}$ as

$$\tilde{\mu}_t(\mathbf{x}_t, \mathbf{x}_0) = \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1 - \bar{\alpha}_t}\mathbf{x}_0 + \frac{\sqrt{\bar{\alpha}_t}(1 - \bar{\alpha}_{t-1})}{1 - \bar{\alpha}_t}\mathbf{x}_t$$

$$\tilde{\beta}_t = \frac{1 - \bar{\alpha}_{t-1}}{1 - \bar{\alpha}_t}\beta_t$$

- Given x0 and xT we can calculate mean and variance. Also, model will predict mean and variance. From these two types of mean and variance we would calculate kl-divergence and entropy for calculating the loss.

Reverse process

The generative distribution that we aim to learn will be trained to perform the *reverse* trajectory, starting from Gaussian noise to gradually remove local perturbations. Therefore the reverse process starts with our given tractable distribution $p(\mathbf{x}_T) = \pi(\mathbf{x}_T)$ and is described as

$$p_\theta(\mathbf{x}_{0:T}) = p(\mathbf{x}_T) \prod_{t=1}^{T} p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t)$$

Each of the transitions in this process can simply be defined as conditional Gaussians (**note**: which is reminiscent of the definition of VAEs). Therefore, during learning, only the mean and covariancce for a Gaussian diffusion kernel needs to be trained

$$p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \mu_\theta(\mathbf{x}_t, t), \mathbf{\Sigma}_\theta(\mathbf{x}_t, t))$$

The two functions defining the mean $\mu_\theta(\mathbf{x}_t, t)$ and covariance $\mathbf{\Sigma}_\theta(\mathbf{x}_t, t)$ can be parametrized by deep neural networks. Note also that these functions are parametrized by t, which means that a single model can be used for all time steps.

- The reverse process consists in inferring the values of the mean and log variance for a given timestep.
- Then, once we have learned the corresponding model, we can perform the denoising of any given timestep, by providing both the sample x_T at a given time step, and that time step t that we can use to condition the models for $\mu_\theta(x_t, t)$ and $\mathbf{\Sigma}_\theta(x_t, t)$.
- Finally, obtaining samples from the model is given by running through the whole Markov chain in reverse, starting from the normal distribution to obtain samples from the target distribution.

Model probability

The complete probability of the generative model is defined as

$$p_\theta(\mathbf{x}_0) = \int p_\theta(\mathbf{x}_{0:T}) d\mathbf{x}_{1:T}$$

At first sight, this integral appears intractable. However, using a similar approach than variational inference, this integral can be rewritten as

$$\begin{aligned} p_\theta(\mathbf{x}_0) &= \int p_\theta(\mathbf{x}_{0:T}) \frac{q(\mathbf{x}_{1:T} \mid \mathbf{x}_0)}{q(\mathbf{x}_{1:T} \mid \mathbf{x}_0)} d\mathbf{x}_{1:T} \\ &= \int q(\mathbf{x}_{1:T} \mid \mathbf{x}_0) \frac{p(\mathbf{x}_{0:T})}{q(\mathbf{x}_{1:T} \mid \mathbf{x}_0)} d\mathbf{x}_{1:T} \end{aligned}$$

2.4. Training

Training amounts to maximizing the model log likelihood,

$$L = \int d\mathbf{x}^{(0)} q\left(\mathbf{x}^{(0)}\right) \log p\left(\mathbf{x}^{(0)}\right) \tag{10}$$

$$= \int d\mathbf{x}^{(0)} q\left(\mathbf{x}^{(0)}\right) \cdot \log \left[\begin{array}{c} \int d\mathbf{x}^{(1\cdots T)} q\left(\mathbf{x}^{(1\cdots T)}|\mathbf{x}^{(0)}\right) \cdot \\ p\left(\mathbf{x}^{(T)}\right) \prod_{t=1}^{T} \frac{p\left(\mathbf{x}^{(t-1)}|\mathbf{x}^{(t)}\right)}{q\left(\mathbf{x}^{(t)}|\mathbf{x}^{(t-1)}\right)} \end{array} \right], \tag{11}$$

which has a lower bound provided by Jensen's inequality,

$$L \geq \int d\mathbf{x}^{(0\cdots T)} q\left(\mathbf{x}^{(0\cdots T)}\right) \cdot \log \left[p\left(\mathbf{x}^{(T)}\right) \prod_{t=1}^{T} \frac{p\left(\mathbf{x}^{(t-1)}|\mathbf{x}^{(t)}\right)}{q\left(\mathbf{x}^{(t)}|\mathbf{x}^{(t-1)}\right)} \right]. \tag{12}$$

As described in Appendix B, for our diffusion trajectories this reduces to,

$$L \geq K \tag{13}$$

$$K = -\sum_{t=2}^{T} \int d\mathbf{x}^{(0)} d\mathbf{x}^{(t)} q\left(\mathbf{x}^{(0)}, \mathbf{x}^{(t)}\right) \cdot D_{KL}\left(q\left(\mathbf{x}^{(t-1)}|\mathbf{x}^{(t)}, \mathbf{x}^{(0)}\right) \middle|\middle| p\left(\mathbf{x}^{(t-1)}|\mathbf{x}^{(t)}\right)\right) + H_q\left(\mathbf{X}^{(T)}|\mathbf{X}^{(0)}\right) - H_q\left(\mathbf{X}^{(1)}|\mathbf{X}^{(0)}\right) - H_p\left(\mathbf{X}^{(T)}\right). \tag{14}$$

where the entropies and KL divergences can be analytically computed. The derivation of this bound parallels the derivation of the log likelihood bound in variational Bayesian methods.

As in Section 2.3 if the forward and reverse trajectories are identical, corresponding to a quasi-static process, then the inequality in Equation 13 becomes an equality.

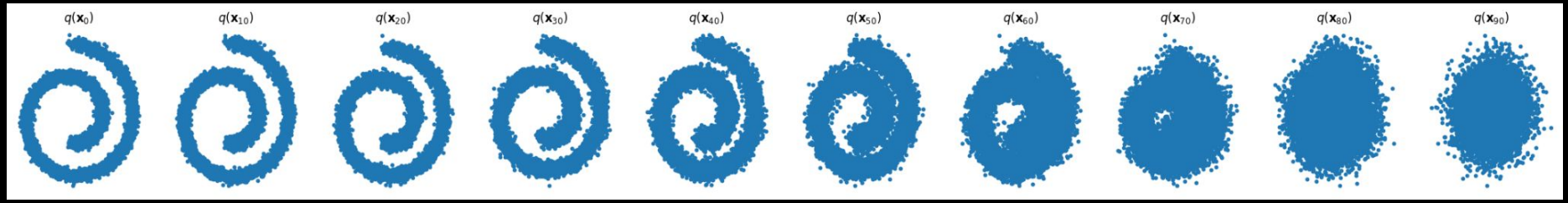
Training consists of finding the reverse Markov transitions which maximize this lower bound on the log likelihood,

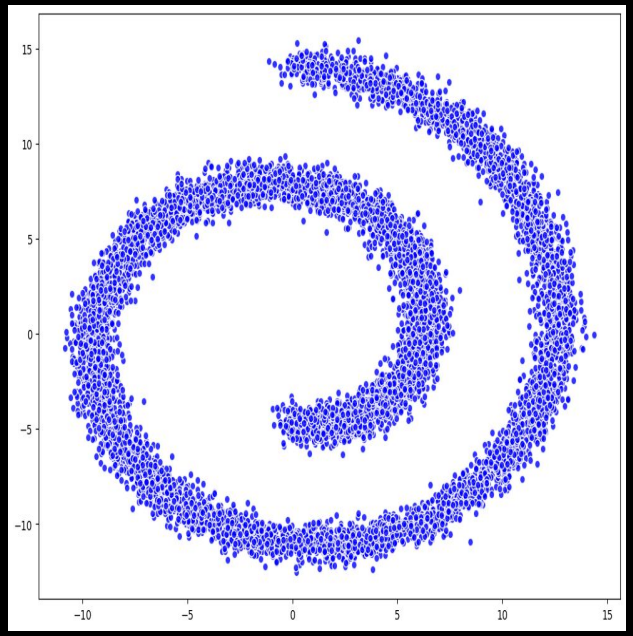
$$\hat{p}\left(\mathbf{x}^{(t-1)}|\mathbf{x}^{(t)}\right) = \underset{p\left(\mathbf{x}^{(t-1)}|\mathbf{x}^{(t)}\right)}{\operatorname{argmax}} K. \tag{15}$$

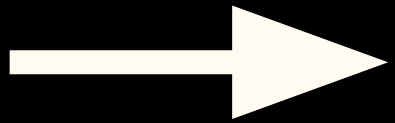
Datasets

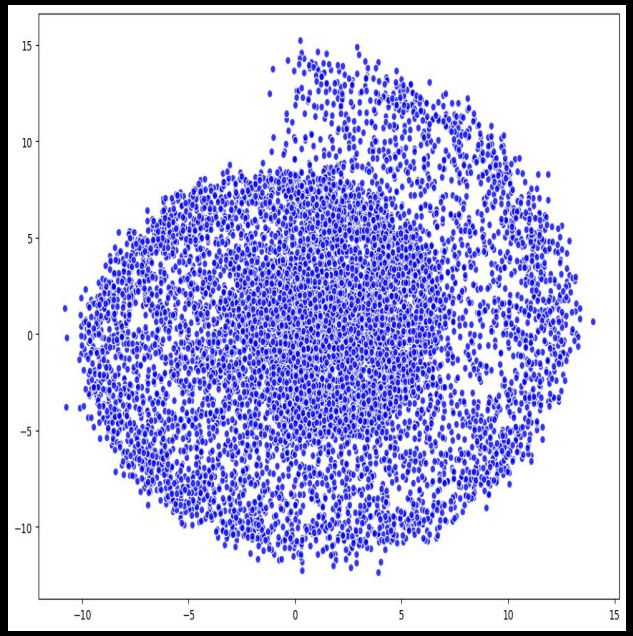
- Swiss Roll
- Binary Heartbeat Distribution

Results on Swiss-Roll Dataset

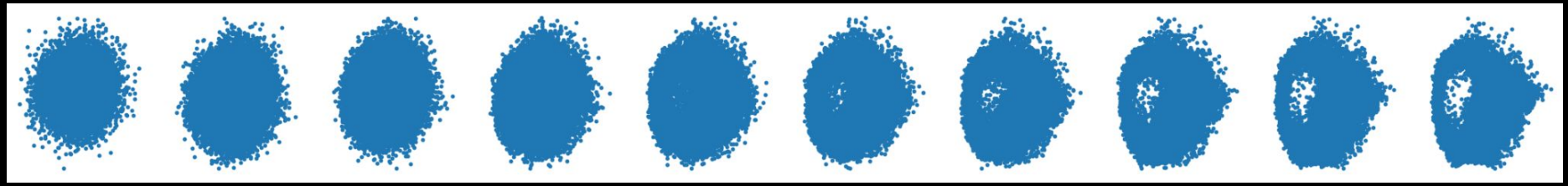

$q(\mathbf{x}_0)$
$q(\mathbf{x}_{10})$
$q(\mathbf{x}_{20})$
$q(\mathbf{x}_{30})$
$q(\mathbf{x}_{40})$
$q(\mathbf{x}_{50})$
$q(\mathbf{x}_{60})$
$q(\mathbf{x}_{70})$
$q(\mathbf{x}_{80})$
$q(\mathbf{x}_{90})$


15
10
5
0
−5
−10
−10
−5
0
5
10
15

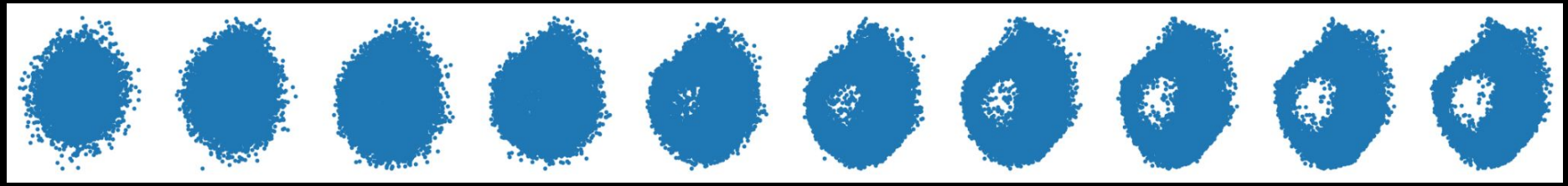



15
10
5
0
−5
−10
−10
−5
0
5
10
15

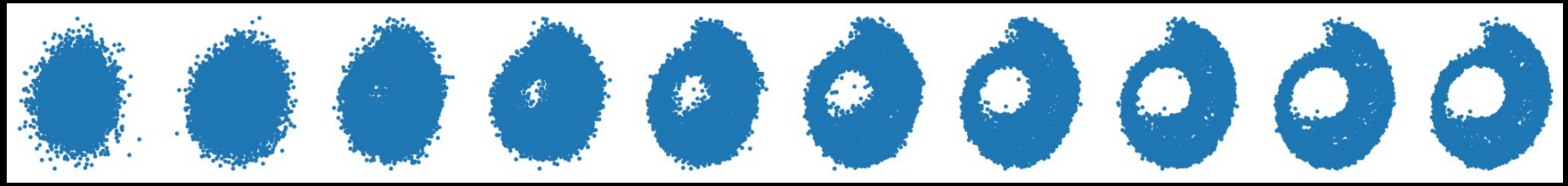
Epoch 5000



Epoch 15000



Epoch 20000

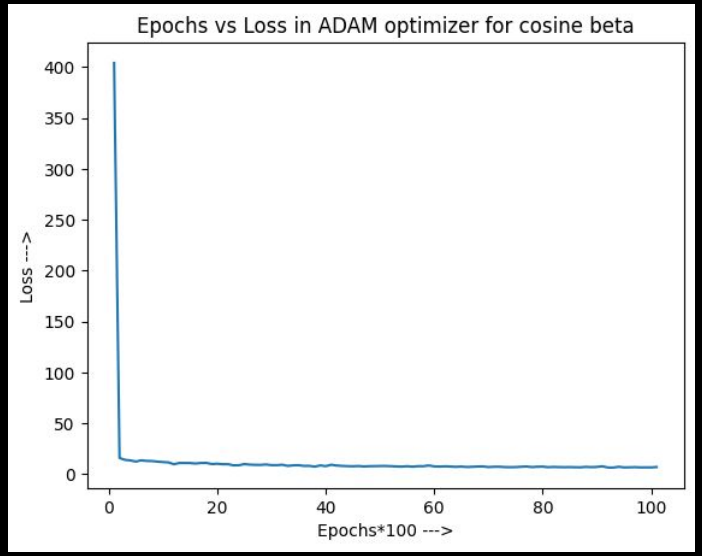


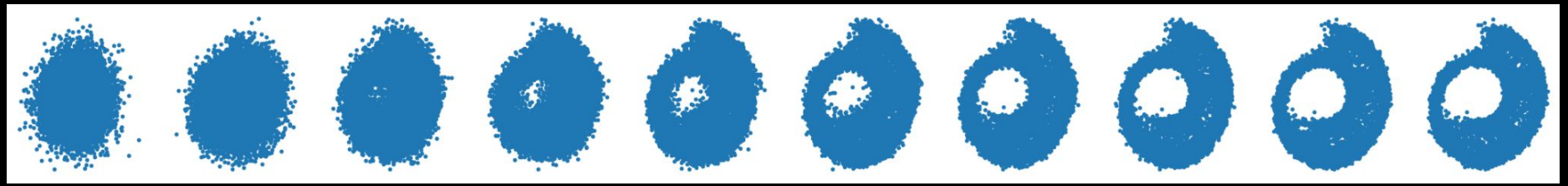
Observations on Swiss-Roll Dataset

- Tried different optimizers like **ADAM**, **SGD**
- Tried **SGD** with **momentum**
- Tried **SGD** with different **learning rates**
- Tried **Exponential learning rate scheduler** with **SGD**

ADAM with lr = 1e-3

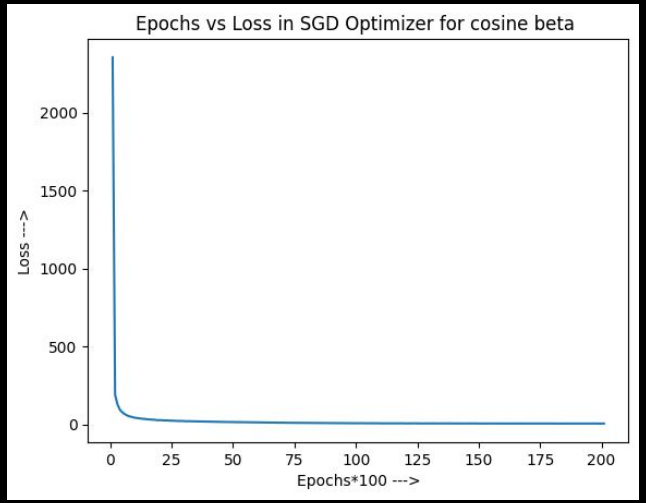
```
Loss at epoch 0: 189.8205108642578
Loss at epoch 1000: 10.199263572692871
Loss at epoch 2000: 7.748318672180176
Loss at epoch 3000: 8.863369941711426
Loss at epoch 4000: 4.928744316101074
Loss at epoch 5000: 7.851652145385742
Loss at epoch 6000: 10.593099594116211
Loss at epoch 7000: 11.302024841308594
Loss at epoch 8000: 5.04187536239624
Loss at epoch 9000: 13.53701400756836
Loss at epoch 10000: 5.288584232330322
```

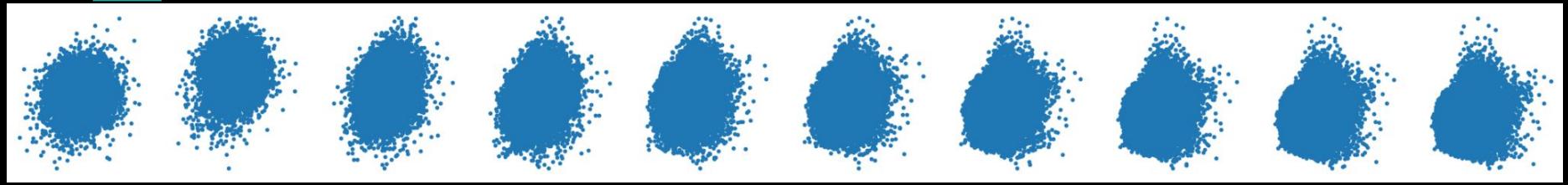




SGD with lr = 1e-3

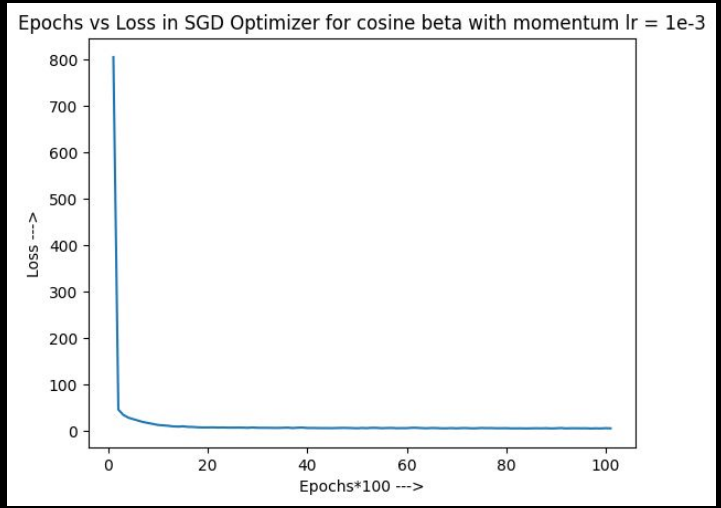
```
Loss at epoch 0: 1736.308837890625
Loss at epoch 1000: 40.05021286010742
Loss at epoch 2000: 30.18717384338379
Loss at epoch 3000: 24.098255157470703
Loss at epoch 4000: 12.923372268676758
Loss at epoch 5000: 14.345239639282227
Loss at epoch 6000: 11.737753868103027
Loss at epoch 7000: 10.44119644165039
Loss at epoch 8000: 7.27048397064209
Loss at epoch 9000: 6.894687652587891
Loss at epoch 10000: 7.518470764160156
```

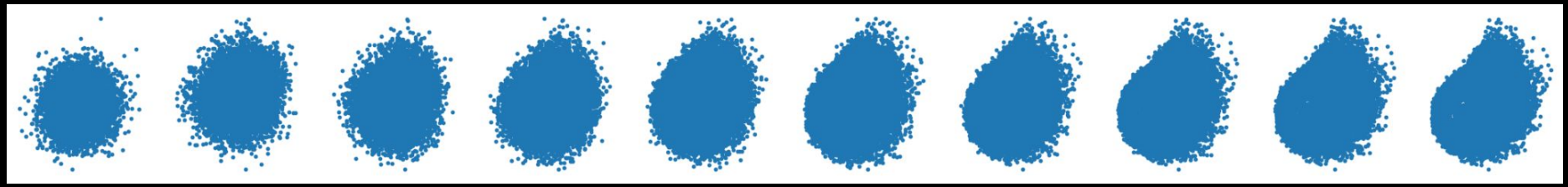




SGD with lr = 1e-3 & momentum

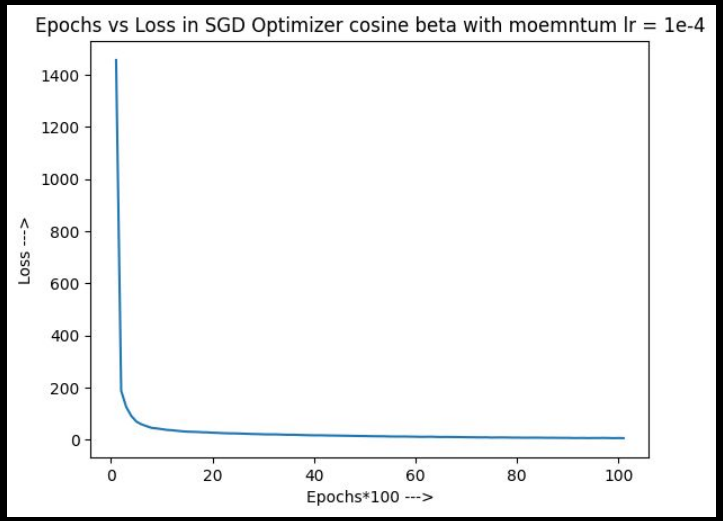
```
Loss at epoch 0: 474.4627685546875
Loss at epoch 1000: 16.98560333251953
Loss at epoch 2000: 5.544562339782715
Loss at epoch 3000: 7.445226192474365
Loss at epoch 4000: 4.380074977874756
Loss at epoch 5000: 4.513489246368408
Loss at epoch 6000: 4.156125545501709
Loss at epoch 7000: 2.308318614959717
Loss at epoch 8000: 6.270468711853027
Loss at epoch 9000: 10.138538360595703
Loss at epoch 10000: 3.5098228454589844
```

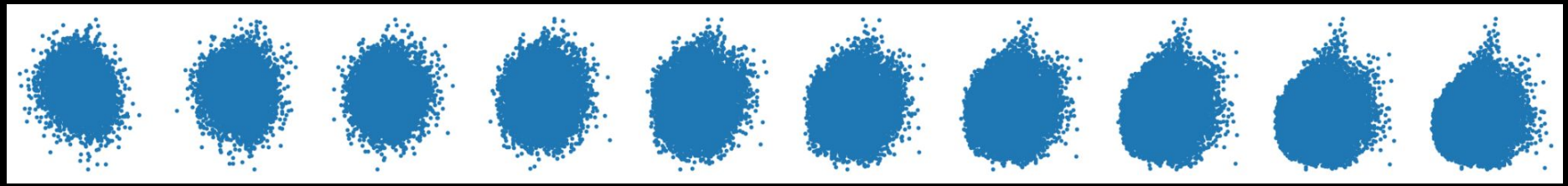




SGD with lr = 1e-4 & momentum

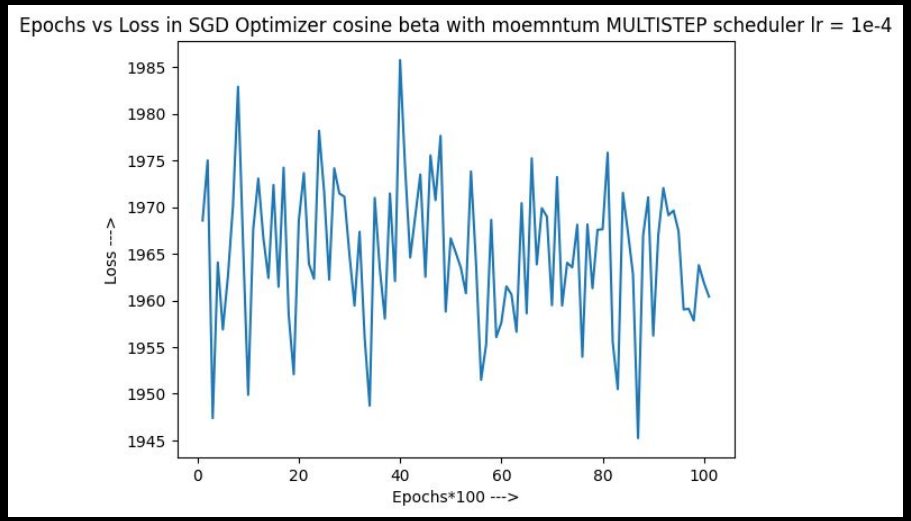
```
Loss at epoch 0: 1308.933349609375
Loss at epoch 1000: 36.70450973510742
Loss at epoch 2000: 25.268613815307617
Loss at epoch 3000: 18.483257293701172
Loss at epoch 4000: 18.706167221069336
Loss at epoch 5000: 13.685323715209961
Loss at epoch 6000: 12.569723129272461
Loss at epoch 7000: 8.665175437927246
Loss at epoch 8000: 7.795670509338379
Loss at epoch 9000: 10.912981033325195
Loss at epoch 10000: 5.655966281890869
```

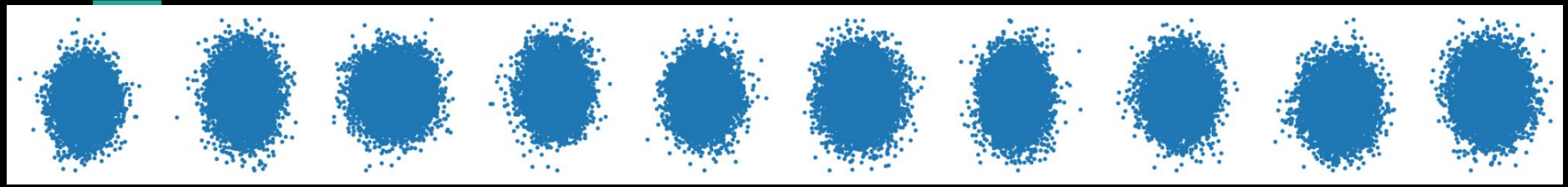




SGD with momentum & Exponential LR

```
Loss at epoch 0: 2024.6009521484375
Loss at epoch 1000: 1954.3485107421875
Loss at epoch 2000: 1857.2830810546875
Loss at epoch 3000: 1956.352783203125
Loss at epoch 4000: 1922.092529296875
Loss at epoch 5000: 1862.5538330078125
Loss at epoch 6000: 2004.3612060546875
Loss at epoch 7000: 1978.520263671875
Loss at epoch 8000: 2012.7254638671875
Loss at epoch 9000: 1897.0361328125
Loss at epoch 10000: 2208.8837890625
```





Observations

- ADAM optimizer giving best result, almost regenerating the image back
- SGD with momentum also working fine, loss quite low, but fails to reconstruct the image as ADAM
- When tried with Exponential LR, loss on higher side, graph Epoch vs Loss graph not smooth, lots of fluctuations.

Experiments/ Improvements on Swiss Roll

- **Cosine** scheduler to create betas instead of Linear scheduler.
- Implemented MSE loss instead of kl-divergence and entropy to calculate loss.
- Timestep vary(100, 150, ..)

Linear

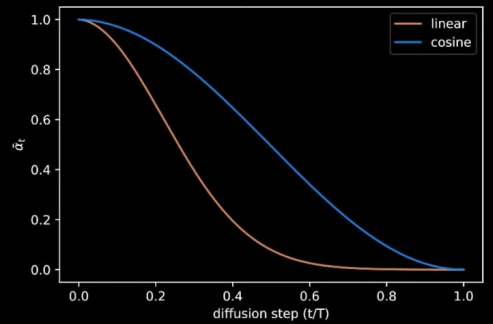
```
if schedule == 'linear':
    betas = torch.linspace(start, end, n timesteps)
```

Cosine

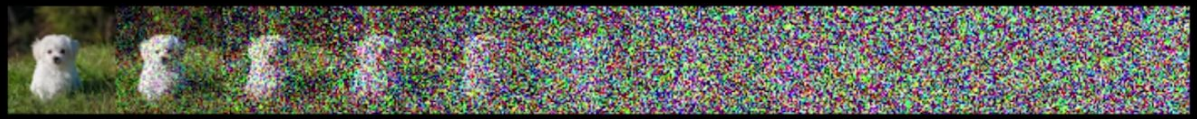
$$f(t) = \cos\left(\frac{t/T + s}{1+s} \cdot \frac{\pi}{2}\right)^2$$

$$\bar{\alpha}_t = \frac{f(t)}{f(0)}$$

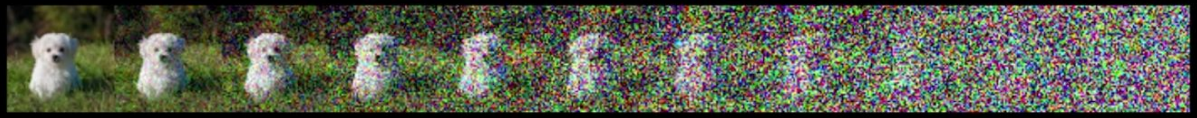
$$\beta_t = 1 - \frac{\bar{\alpha}_t}{\bar{\alpha}_{t-1}}$$

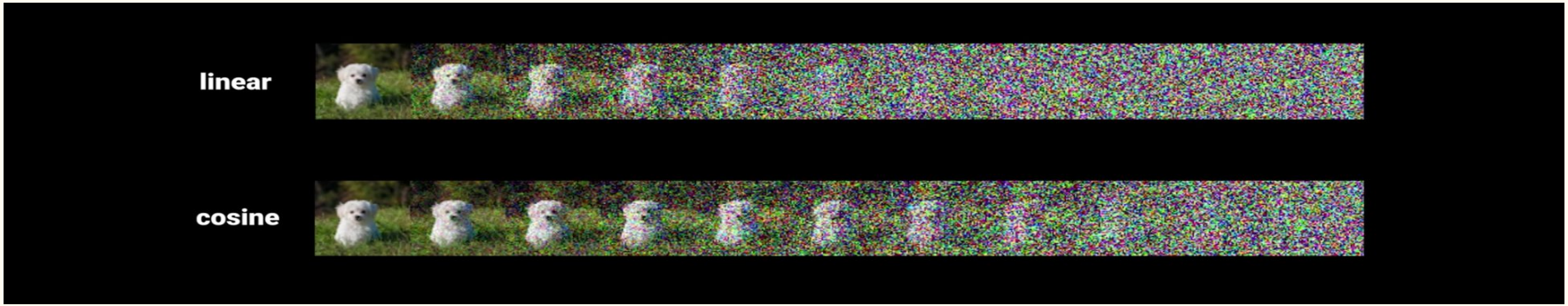


linear

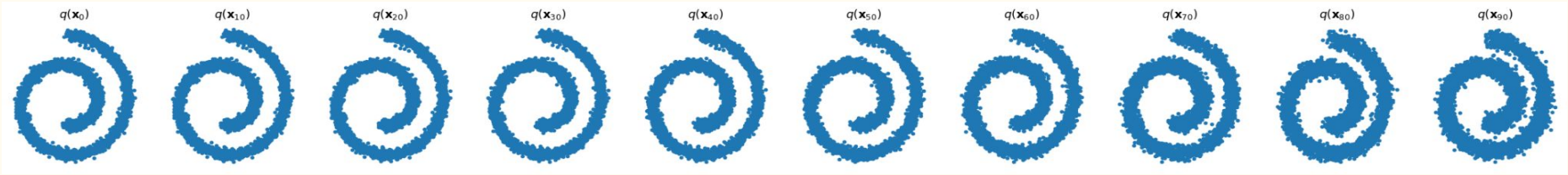


cosine

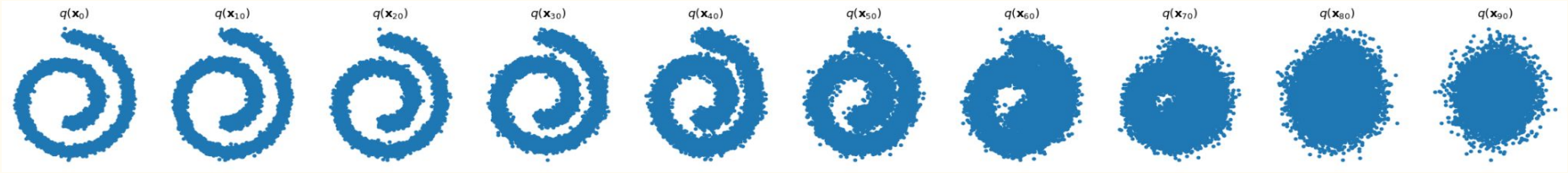




Linear



Cosine



Loss are minimized havoc when applied cosine beta

Linear

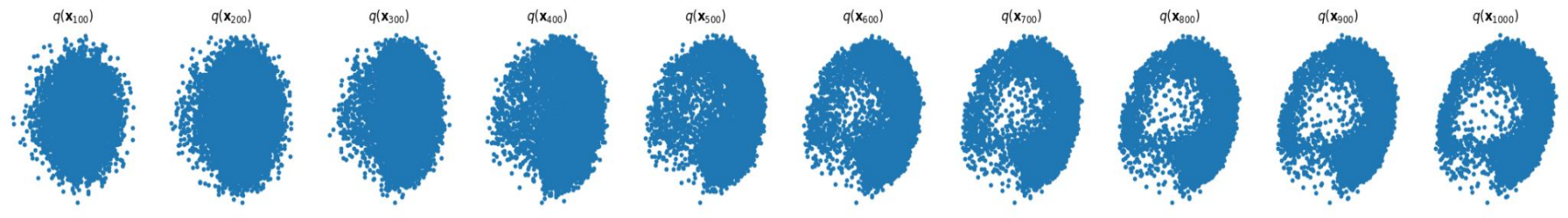
```
tensor(441.3799, grad_fn=<MeanBackward0>)
tensor(47.0368, grad_fn=<MeanBackward0>)
tensor(39.6782, grad_fn=<MeanBackward0>)
tensor(50.6556, grad_fn=<MeanBackward0>)
tensor(42.3937, grad_fn=<MeanBackward0>)
tensor(43.3973, grad_fn=<MeanBackward0>)
tensor(40.4870, grad_fn=<MeanBackward0>)
tensor(39.2022, grad_fn=<MeanBackward0>)
tensor(39.9992, grad_fn=<MeanBackward0>)
tensor(33.5111, grad_fn=<MeanBackward0>)
tensor(37.8390, grad_fn=<MeanBackward0>)
```

Cosine

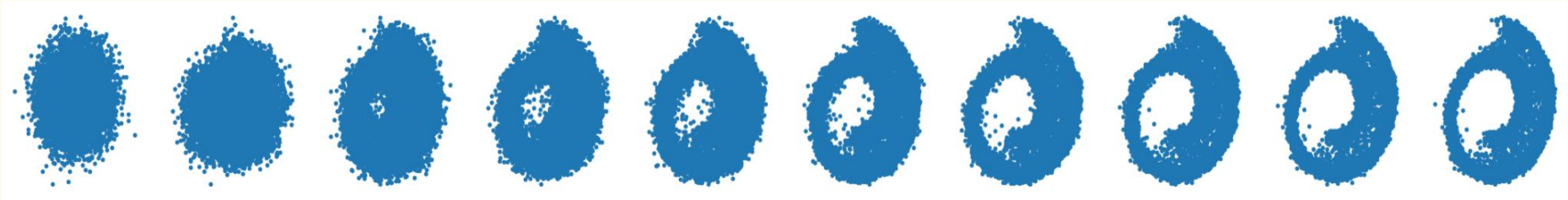
```
Loss at epoch 0: 189.8205108642578
Loss at epoch 1000: 10.199263572692871
Loss at epoch 2000: 7.748318672180176
Loss at epoch 3000: 8.863369941711426
Loss at epoch 4000: 4.928744316101074
Loss at epoch 5000: 7.851652145385742
Loss at epoch 6000: 10.593099594116211
Loss at epoch 7000: 11.302024841308594
Loss at epoch 8000: 5.04187536239624
Loss at epoch 9000: 13.53701400756836
Loss at epoch 10000: 5.288584232330322
```

Image reconstructed almost perfectly when applied cosine

Linear in ADAM



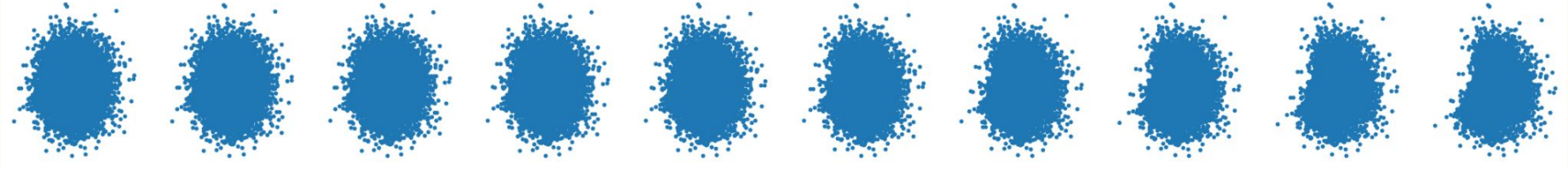
Cosine in ADAM



MSE loss instead of kl-divergence for calculating loss

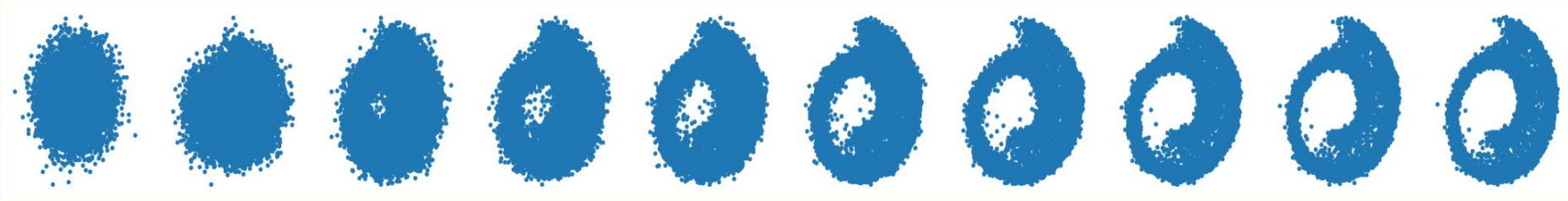
```
# Perform diffusion for step t, x0 -> xt
x_t = q_sample(x_0, t)

model_output = model(x_t, t)

mean, log_var = torch.split(model_output, 2, dim=-1)

actual_var = torch.exp(0.5 * log_var)
rand_noise = torch.randn_like(x_0)
sample = mean + actual_var * rand_noise

total_loss = F.mse_loss(sample, x_t)
```

Results are not so good, but would reconstruct better when epochs will be increased.

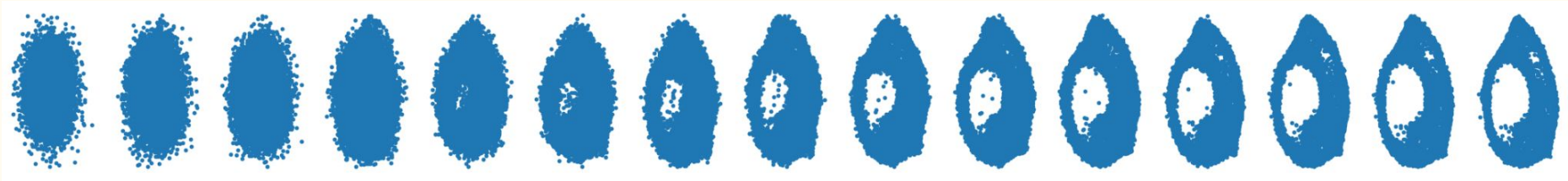


Timestep varry

Timestep = 100



Timestep = 150



Challenges

- For each model training, it’s taking hours

- If no. of datapoints are more(>10000) in the dataset, it’s taking too much time to compute and also consuming too much RAM, which ends in crashing session in COLAB.

- If n_timesteps >= 200, it’s taking too much time to compute and also consuming too much RAM, which ends in crashing session in COLAB.

Future Work

- Tweak the model structure and compare the results how they are performing against each other. For eg. changing the layer size, adding more layers etc.

Binary Heartbeat Distribution Dataset

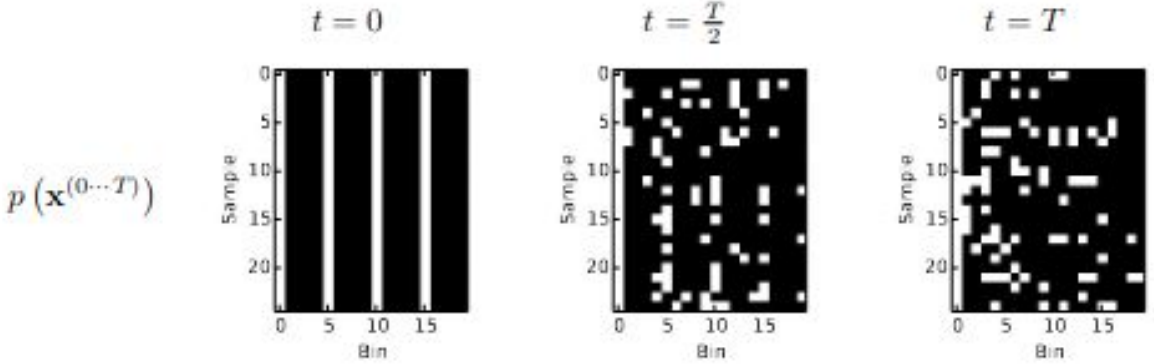


Figure 2. Binary sequence learning via binomial diffusion. A binomial diffusion model was trained on binary 'heartbeat' data, where a pulse occurs every 5th bin. Generated samples (left) are identical to the training data. The sampling procedure consists of initialization at independent binomial noise (right), which is then transformed into the data distribution by a binomial diffusion process, with trained bit flip probabilities. Each row contains an independent sample. For ease of visualization, all samples have been shifted so that a pulse occurs in the first column. In the raw sequence data, the first pulse is uniformly distributed over the first five bins.

$$\mathbf{x}_i^{(t+1)} = \mathbf{x}_i^{(t-1)} + \epsilon_t + \epsilon_{t+1} - \epsilon_t\epsilon_{t+1} - 2(\epsilon_t + \epsilon_{t+1} - \epsilon_t\epsilon_{t+1})\mathbf{x}_i^{(t-1)}$$

denoting $\tilde{\epsilon}_{t+1} := \epsilon_t + \epsilon_{t+1} - \epsilon_t\epsilon_{t+1}$, we see that

$$\mathbf{x}_i^{(t+1)} = \mathbf{x}_i^{(t-1)} + \tilde{\epsilon}_{t+1} - 2\tilde{\epsilon}_{t+1}\mathbf{x}_i^{(t-1)}$$

$$\tilde{\epsilon}_{t+1} = \tilde{\epsilon}_t + \epsilon_{t+1} - \tilde{\epsilon}_t\epsilon_{t+1}$$

$$q\left(\mathbf{x}^{(t)}|\mathbf{x}^{(0)}\right) = \mathcal{B}\left(\mathbf{x}^{(0)}(1-\tilde{\beta}_t) + 0.5\tilde{\beta}_t\right)$$

$$p(\theta|y) = \frac{p(y|\theta)p(\theta)}{p(y)}$$

$$y = \mathbf{x}^{(t)}$$

$$\theta = \mathbf{x}^{(t-1)}$$

$$p(\theta) = q\left(\mathbf{x}^{(t-1)}|\mathbf{x}^{(0)}\right) = \mathcal{B}\left(\mathbf{x}^{(0)}(1-\tilde{\beta}_{t-1}) + 0.5\tilde{\beta}_{t-1}\right)$$

$$p(y|\theta) = q\left(\mathbf{x}^{(t)}|\mathbf{x}^{(t-1)}, \mathbf{x}^{(0)}\right) = \mathcal{B}\left(\mathbf{x}^{(t-1)}(1-\beta_t) + 0.5\beta_t\right)$$

$$p(\theta|y) = q\left(\mathbf{x}^{(t-1)}|\mathbf{x}^{(t)}, \mathbf{x}^{(0)}\right)$$

$$p(y) = q\left(\mathbf{x}^{(t)}|\mathbf{x}^{(0)}\right)$$

$$p(\theta|y) = \frac{q\left(\mathbf{x}^{(t)}|\mathbf{x}^{(t-1)}, \mathbf{x}^{(0)}\right) q\left(\mathbf{x}^{(t-1)}|\mathbf{x}^{(0)}\right)}{q\left(\mathbf{x}^{(t)}|\mathbf{x}^{(0)}\right)}$$

KL divergence of Multivariate Bernoulli Distributions

In order to calculate K, we will need to calculate

$$D_{KL}\left(q\left(\mathbf{x}^{(t-1)}|\mathbf{x}^{(t)},\mathbf{x}^{(0)}\right)||p\left(\mathbf{x}^{(t-1)}|\mathbf{x}^{(t)}\right)\right)$$

$$H(p(x)) = -\sum_{x\in\mathcal{X}} p(x)\log p(x)$$

$$= -\sum_{k=0}^{n}\binom{n}{k}(1-0.5\tilde{\beta}_T)^k(0.5\tilde{\beta}_T)^{n-k}\log\Big((1-0.5\tilde{\beta}_T)^k(0.5\tilde{\beta}_T)^{n-k}\Big)$$

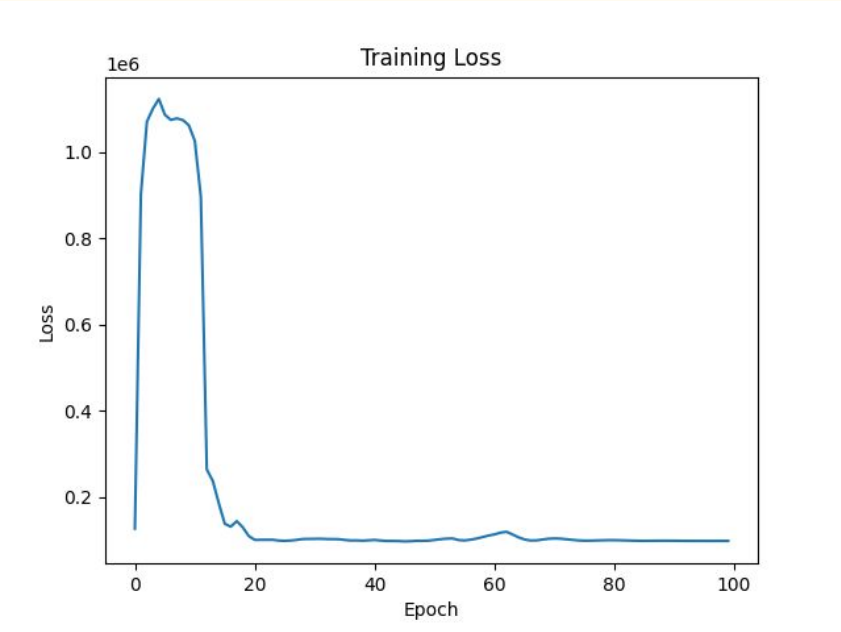
Results on Binary Heartbeat Distribution

Experiments

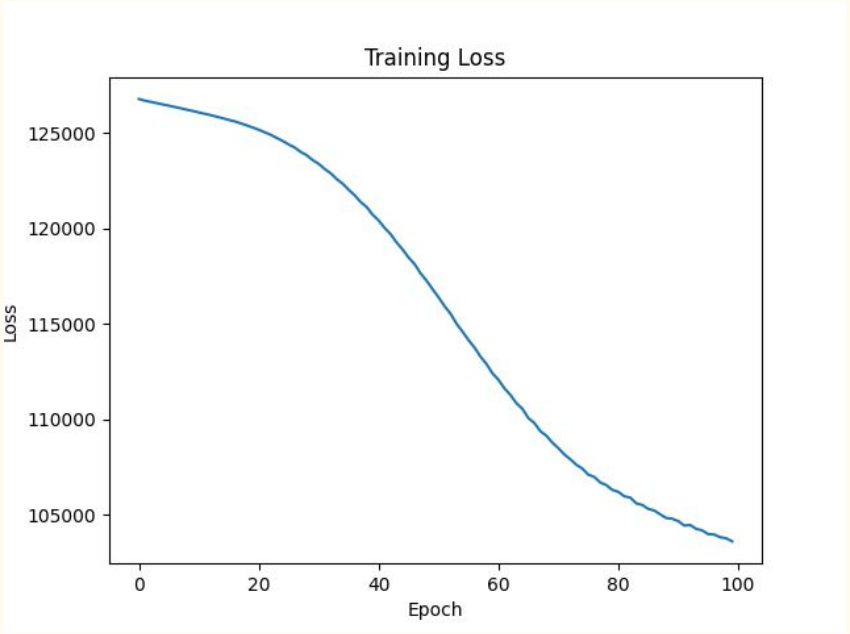
- SGD vs ADAM at different learning rates
- ADAM with diff learning rates
- ADAM with diff epochs
- Noising
- Final reversal
- Model tweak

ADAM vs SGD at different learning rates

ADAM $\mathrm{lr} = 0.5$

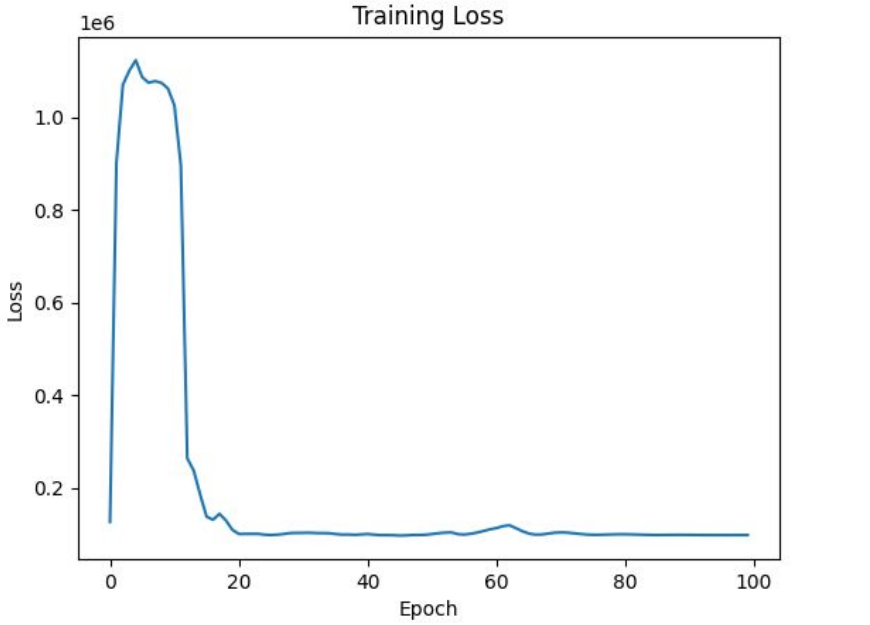


SGD $\mathrm{lr} = 0.5$

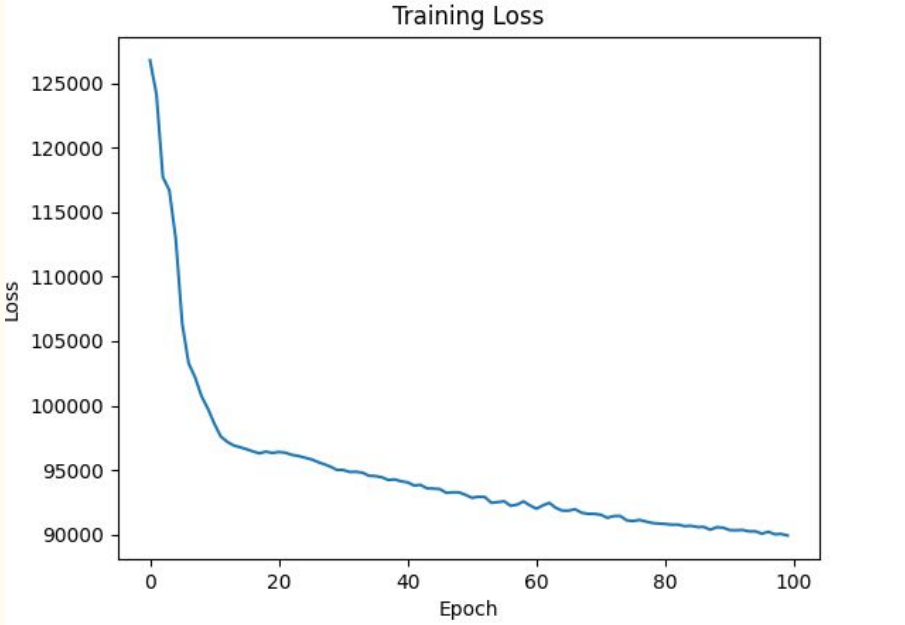


ADAM with different learning rates

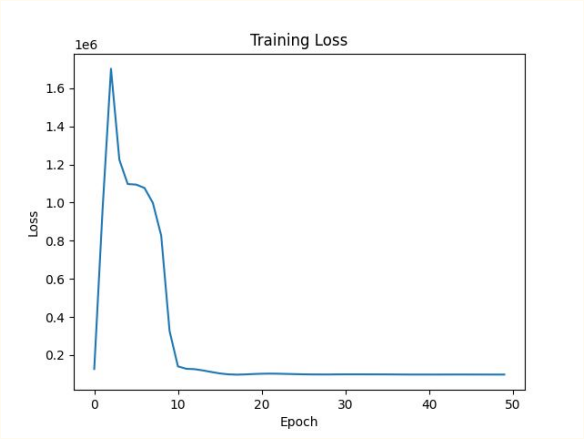
ADAM $\mathrm{lr} = 0.5$

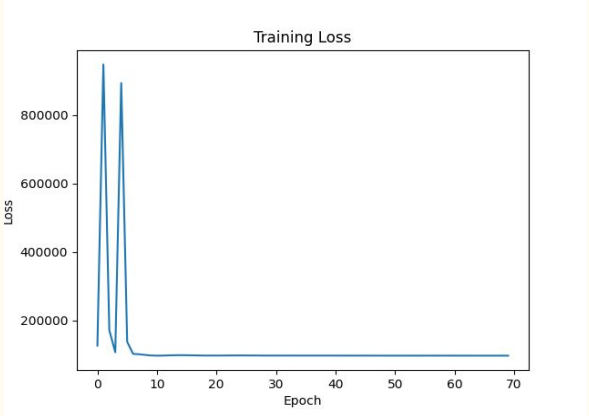


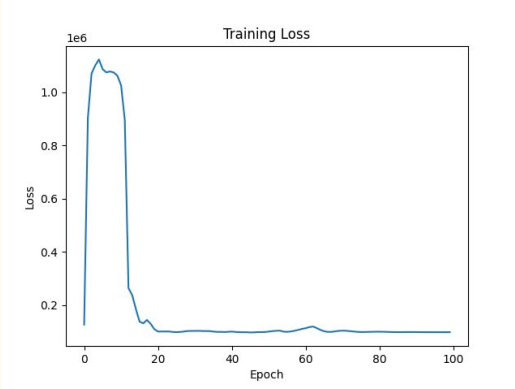
ADAM $\mathrm{lr} = 0.05$



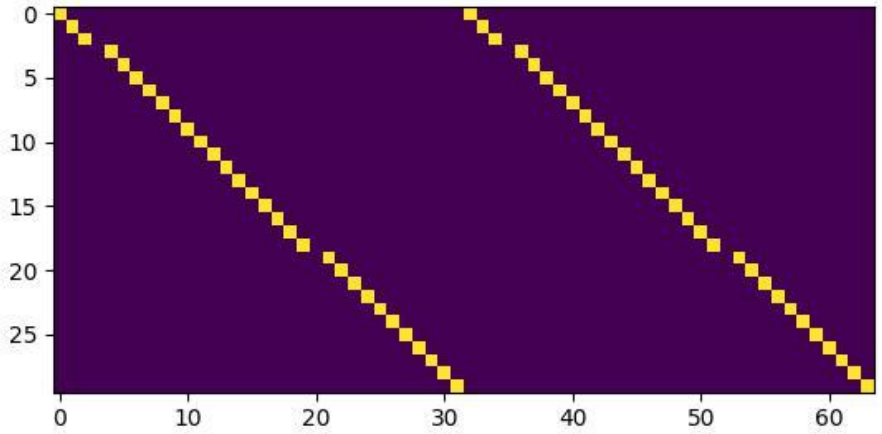
ADAM with different epochs





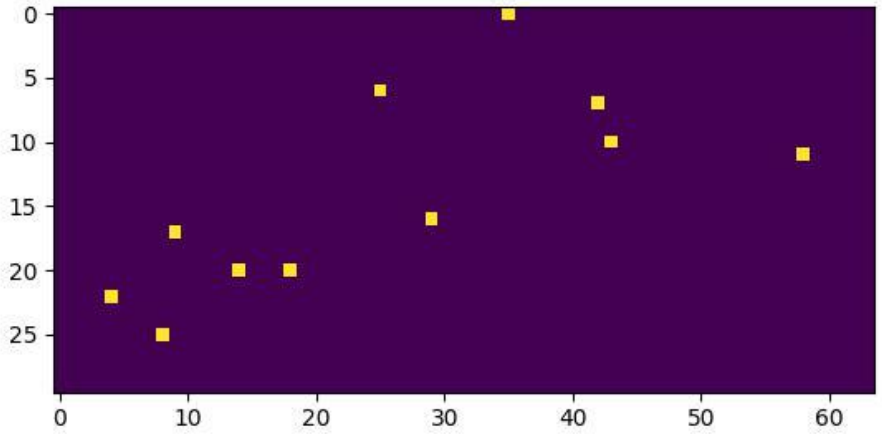


Noising for epochs = 30



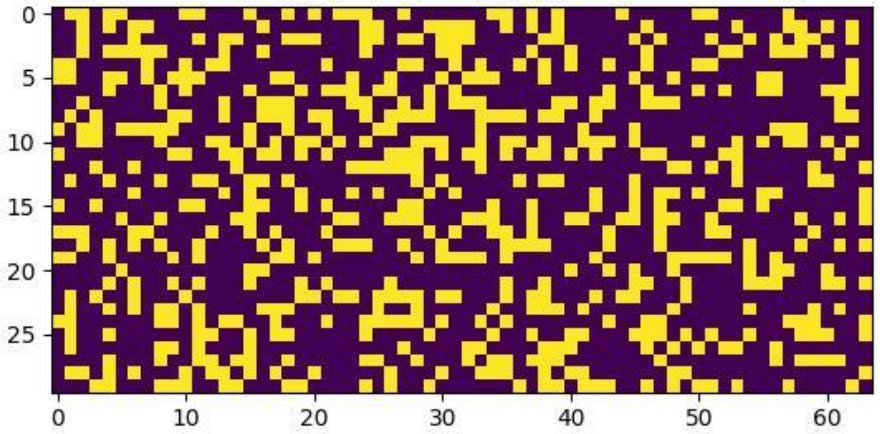
Actual data

Noising for epochs = 30



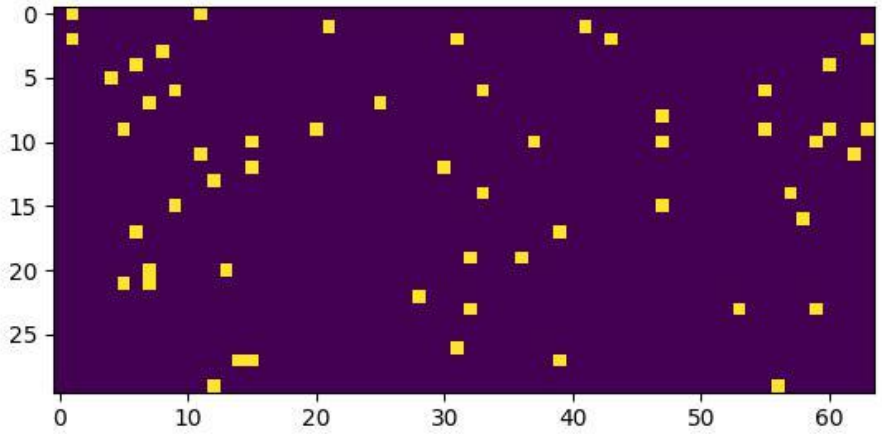
Adding Noise after epoch 10

Noising for epochs = 30



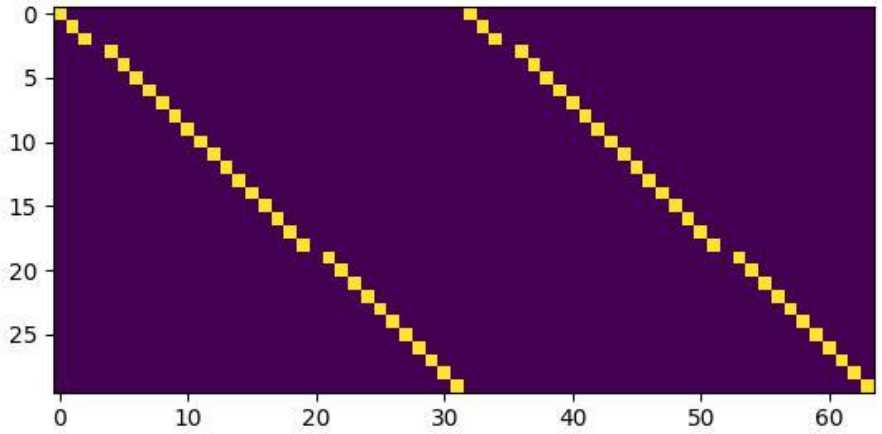
Adding Noise after epoch 20

Noising for epochs = 30



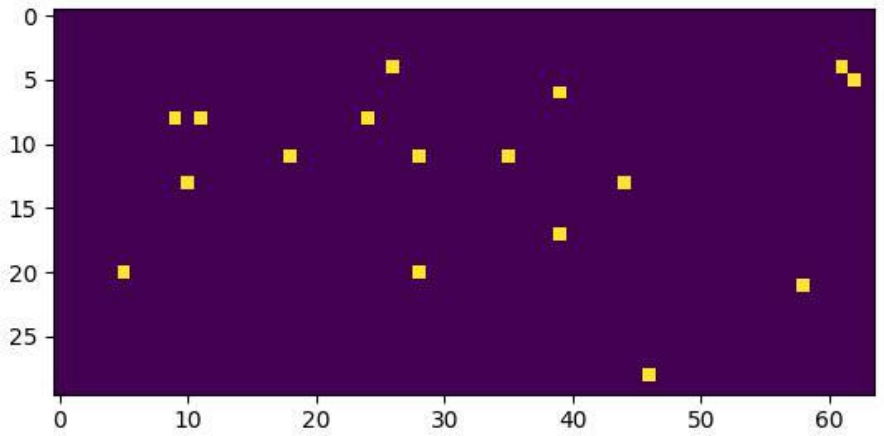
Data after regeneration from pure and random noise

Noising for epochs = 70



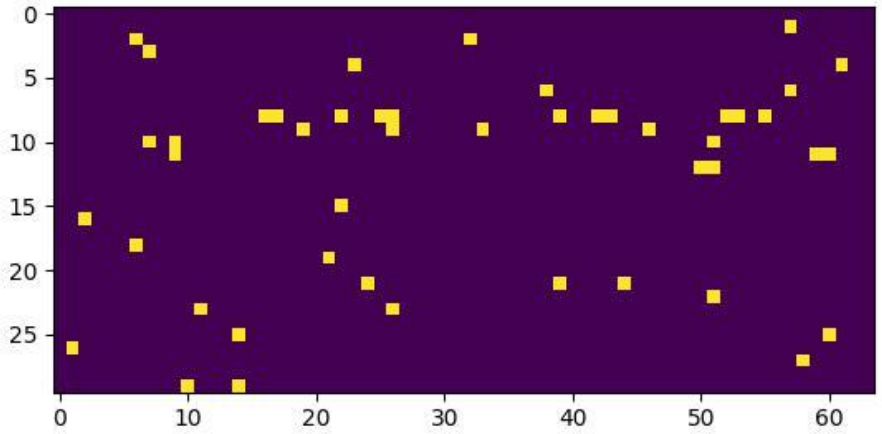
Actual data

Noising for epochs = 70



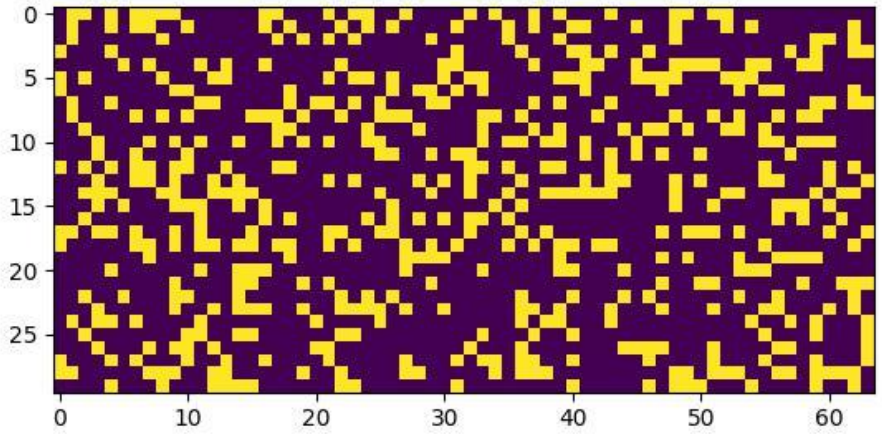
Adding Noise after epoch 10

Noising for epochs = 70



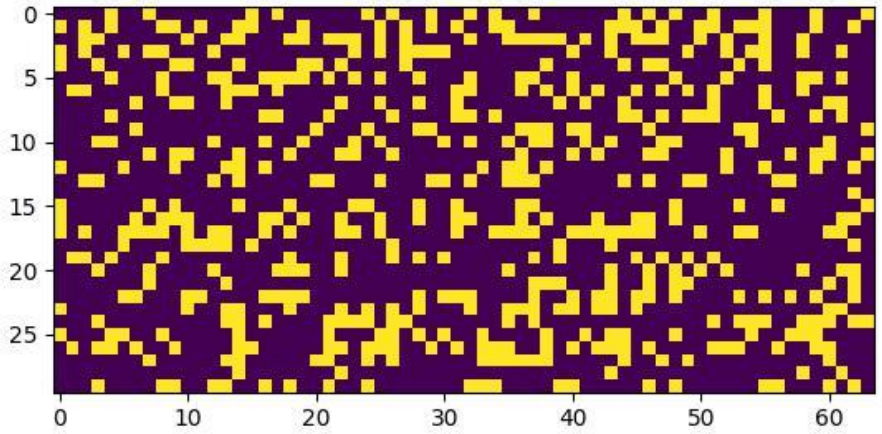
Adding Noise after epoch 20

Noising for epochs = 70



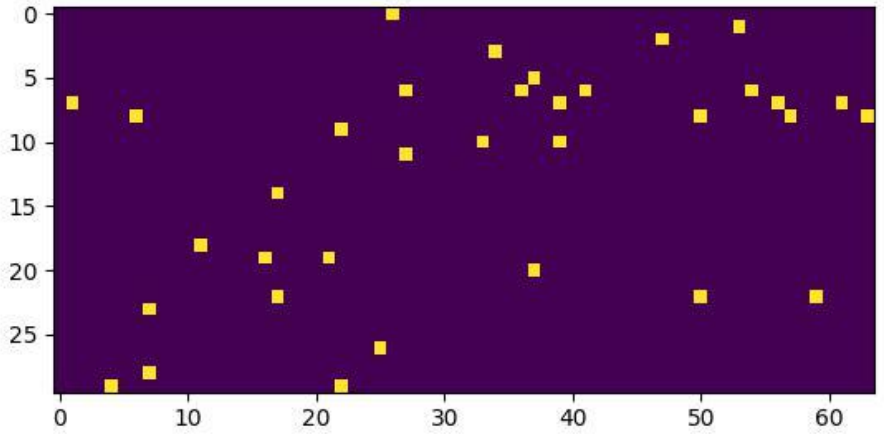
Adding Noise after epoch 40

Noising for epochs = 70



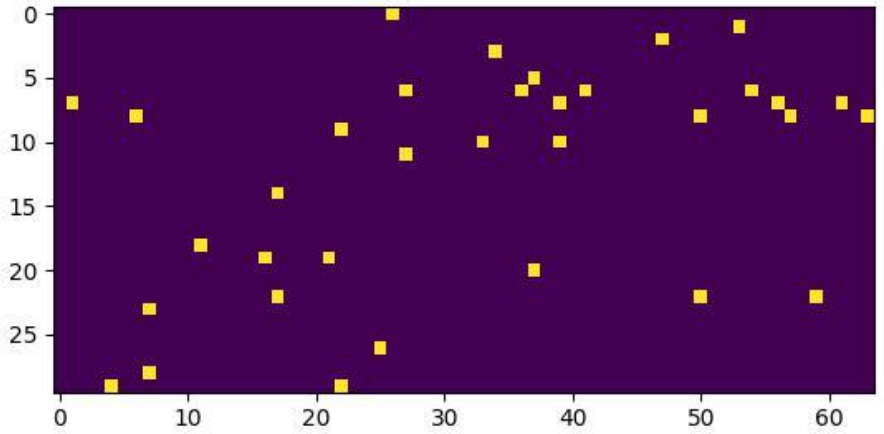
Adding Noise after epoch 60

Noising for epochs = 70



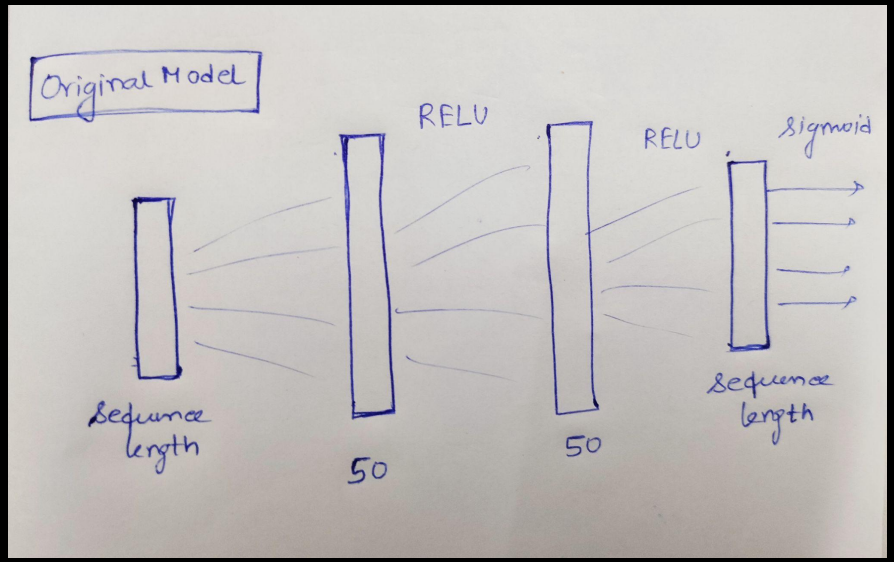
Data after regeneration from pure and random noise

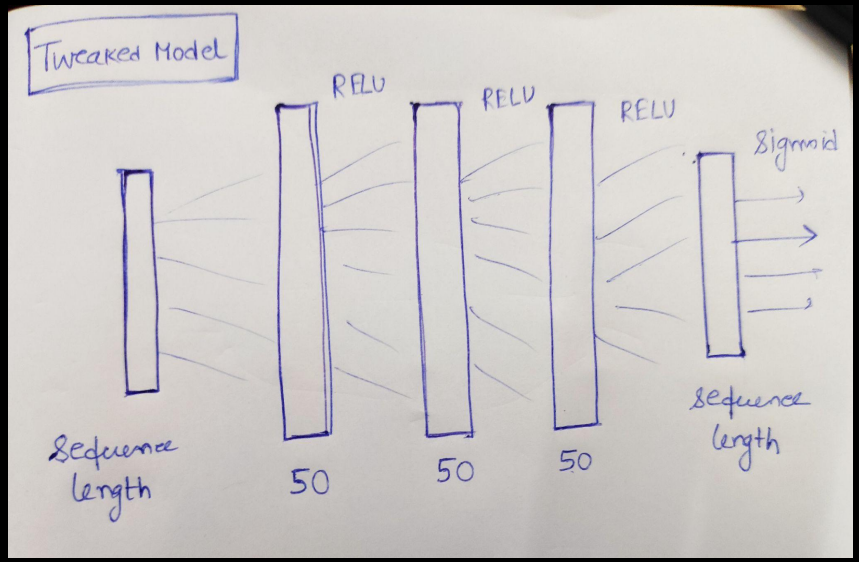
Noising for epochs = 70



Data after regeneration from pure and random noise

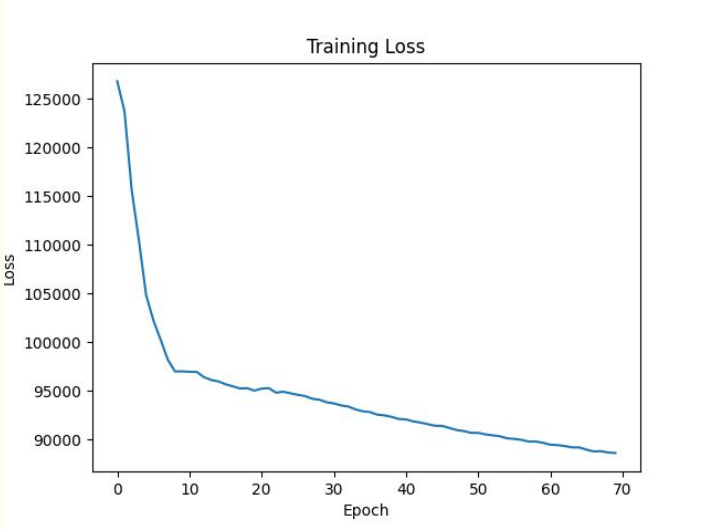
Comparison of Models



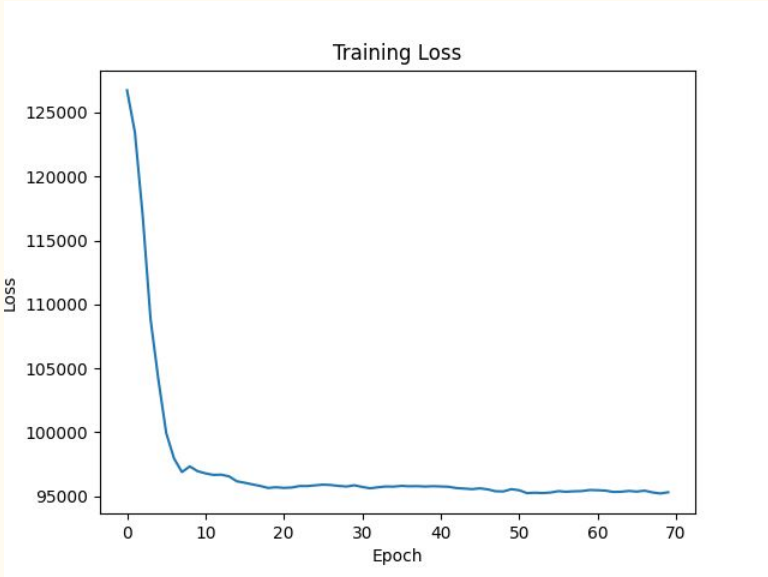


Comparison of models

Proposed Model with
ADAM lr = 0.05

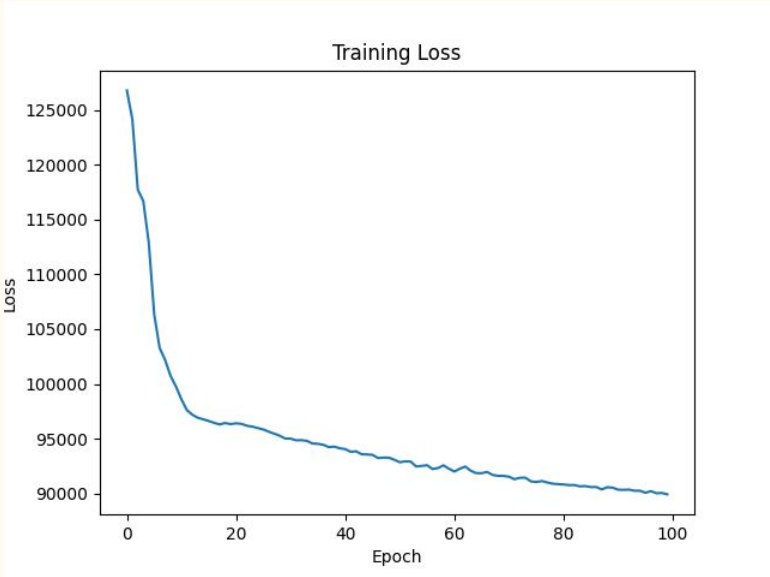


Tweaked Model
ADAM lr = 0.05

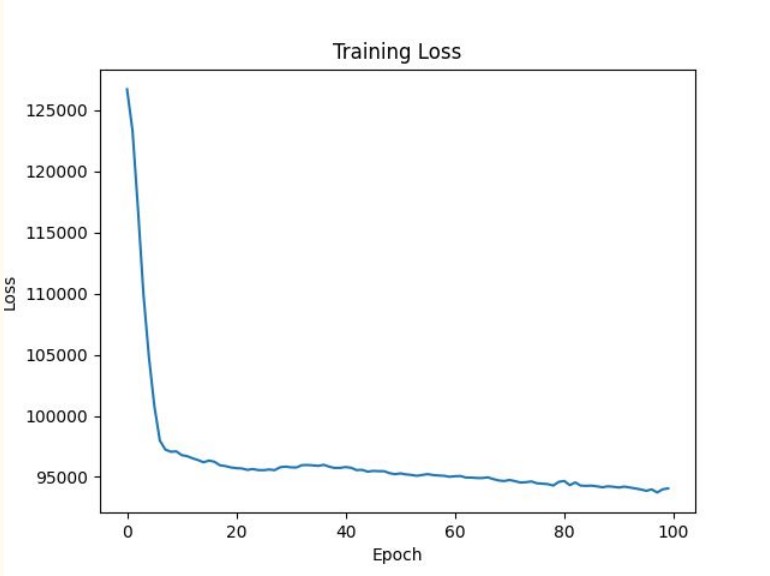


Comparison of models

Proposed Model with
ADAM lr = 0.05



Tweaked Model
ADAM lr = 0.05



Thank You