Report of Application of Machine Learning

Name: Jingjing Gao
ID Number: jg2098

24 March 2024

1 Training a Diffusion Model

1.1 a

Briefly describe this model and the training algorithm.

Denoising diffusion probabilistic models (DDPMs) are a class of generative models that work by iteratively adding noise to an input signal (like an image, text, or audio) and then learning to denoise from the noisy signal to generate new samples. Jonathan Ho proposed a two-step process: a noising(diffusion) process and a denoising process.

1.1.1 Diffusion process

The diffusion process in DDPM (Ho et al., 2020) is defined by a Markov chain that gradually adds Gaussian noise to the data according to a noise schedule. The diffusion process is defined by:

$$q(z_{1:T}|x) = \prod_{t=1}^{T} q(z_t|z_{t-1}), \tag{1}$$

where T is the length of the diffusion process, and $z_T, \ldots, z_t, z_{t-1}, \ldots, z_1$ is a sequence of latent variables with the same size as the clean sample x.

The diffusion process is parameterized with a set of parameters called the noise schedule $(\beta_1, \ldots, \beta_T)$, which defines the variance of the noise added at each step:

$$q(z_t|z_{t-1}) := \mathcal{N}(z_t; \sqrt{1-\beta_t}z_{t-1}, \beta_t I), \tag{2}$$

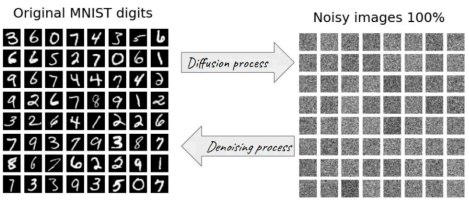
Since we are using a Gaussian noise random variable at each step, the diffusion process can be simulated for any number of steps with the closed formula:

$$z_t = \sqrt{\alpha_t}x + \sqrt{1-\alpha_t}\varepsilon, \tag{3}$$

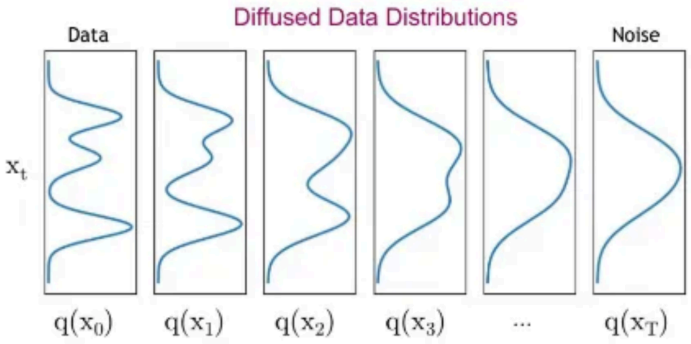
where $\alpha_t = \prod_{t=1}^{T}(1-\beta_t)$ and $\varepsilon \sim \mathcal{N}(0, I)$.

Implementation of the reverse process

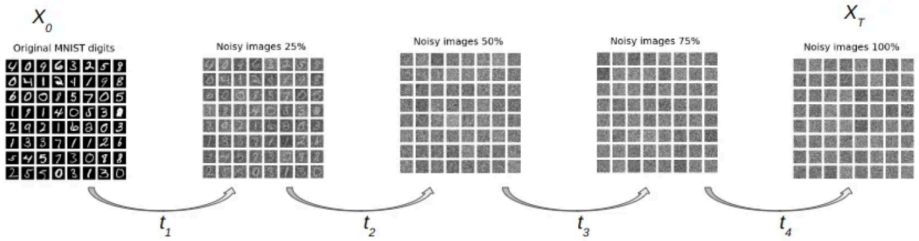
We have an original input signal of a digit from the MNIST dataset as shown in the square grid below in Figure 1. The digits are in high dimensions meaning that they have a more complex distribution as illustrated in Figure 2. Ho proposed that we reduce the complexity by adding noise to these digits until the structure of the data distribution is a simple Gaussian noise that is easy to sample from. So this process of systematically and slowly destroying the structure in the original input data distribution is called the diffusion process.



Figuur 1: Diffusion Process and Denoising Process of DDPM.



Figuur 2: Diffused Data Distributions.



Figuur 3: Diffusion Process.

To get the noisy digits from the original digits as illustrated in Figure 3, we need two pieces:

1. Defining the diffusion steps. Simply that means how many times we are going to add noise to the original image. This includes the minimum and maximum noise scheduler (betas) values (we realize this by using the `ddpm_schedules` function in Python code), that is the lowest and the largest amount of noise we can add to the input data. In the original work by Ho et al., betas are put in a linear space from 0.0001 to 0.02 with 1000 diffusion steps, and that is what we also used for the baseline experiment.

2. Generate Gaussian noise of the same shape as the input data and add the noise iteratively as shown in Figure 3 with t diffusion timesteps.

1.1.2 Reverse process

Now that we have a simple distribution from the diffusion process, we can then learn a reverse process of the diffusion process that restores structure in data and yields a highly flexible and tractable generative model of the data.

In other words, we learn a series of probabilistic mappings back from latent variable z_T to z_{T-1}, from z_{T-1} to z_{T-2}, and so on, until we reach the data x. The true reverse distributions $q(z_{t-1}|z_t)$

of the diffusion process are complex multi-modal distributions that depend on the data distribution $P(x)$. We approximate these as normal distributions:

$$P(z_T) = \text{Norm}_{z_T}[0, I] \tag{4}$$

$$P(z_{t-1}|z_t, \phi_t) = \text{Norm}_{z_{t-1}}\left(f_t[z_t, \phi_t], \sigma_t^2 I\right) \tag{5}$$

$$P(x|z_1, \phi_1) = \text{Norm}_x\left(f_1[z_1, \phi_1], \sigma_1^2 I\right), \tag{6}$$

where $f_t[z_t, \phi_t]$ is a neural network that computes the mean of the normal distribution in the estimated mapping from z_t to the preceding latent variable z_{t-1}. The terms $\{\sigma_t^2\}$ are predetermined. If the hyperparameters β_t in the diffusion process are close to zero (and the number of time steps T is large), then this normal approximation will be reasonable.

We generate new examples from $P(x)$ using ancestral sampling. We start by drawing z_T from $P(z_T)$. Then we sample z_{T-1} from $P(z_{T-1}|z_T, \phi_T)$, sample z_{T-2} from $P(z_{T-2}|z_{T-1}, \phi_{T-1})$, and so on until we finally generate x from $P(x|z_1, \phi_1)$.

Now we replace the model $\hat{z}_{t-1} = f_t[z_t, \phi_t]$ with a new model $\varepsilon = g_t[z_t, \phi_t]$, which predicts the noise ε that was mixed with x to create z_t:

$$\hat{z}_{t-1} = \frac{1}{\sqrt{1-\beta_t}} z_t - \frac{\beta_t}{\sqrt{1-\alpha_t}\sqrt{1-\beta_t}} g_t[z_t, \phi_t]. \tag{7}$$

We get noisy latent z_{t-1}:

$$z_{t-1} = \hat{z}_{t-1} + \sigma_t \varepsilon \tag{8}$$

where $\sigma_t = \sqrt{\beta}_t$ and $\varepsilon \sim \mathcal{N}(0, I)$.

Implementation of the reverse process

1. **Model architecture**:

   A 2D Convolutional Neural Network (CNN) designed to estimate the diffusion process, which can subsequently be leveraged for generating realistic images. Each CNN block includes a convolutional layer, layer normalization, and an activation function. This modular design facilitates hierarchical feature extraction, enabling the representation of increasingly abstract features as the data traverses through the network. Notably, the architecture is highly customizable, allowing for parameter adjustments such as the number of hidden channels per block and the size of convolutional kernels.

   The model incorporates a dedicated time encoding module to handle temporal information. This module employs sinusoidal embeddings to integrate temporal context into the network's architecture. By incorporating temporal dynamics into the input data, the model gains the capability to analyze changes and patterns over time, enhancing its predictive and generative capabilities.

   During the forward pass of the model, input images undergo processing through the CNN blocks, with each block refining the representation of features extracted from the input data. Simultaneously, the model embeds time information into the output of the initial CNN block, incorporating temporal context into the spatial representations of the data. As a result, the model stands as a framework capable of generating realistic images while preserving temporal coherence and capturing intricate spatiotemporal relationships within the data.

2. **Gaussian distribution parameterization**:

   We just need the model to predict the distribution mean and standard deviation given the noisy image and time step. Ho et al. only predicted the mean of the Gaussian, and that is what we also did and had the variance fixed. During the forward process, we use β values to control the variance of the forward diffusion, and now during the reverse denoising processes,

we'll use σ shown in the denoising formula in Equation 8. Often linear schedules, β, are set equal to σ^2.

1.1.3 Training Algorithm

This leads to straightforward algorithms for both training the model (Algorithm 18.1 in Prince) and sampling (Algorithm 18.2) as shown in Figure 4.

The training procedure of ε is defined in Alg. 18.1. Given the input dataset $\mathcal{D}$, the algorithm samples ε, the training data x, and t. The noisy latent state z_t is calculated by Equation 3 and fed to the DDPM neural network g_t. A gradient descent step is taken in order to estimate the ε noise with the DDPM network g_t. The objective for the diffusion model is a variational bound on the model data log likelihood. The code corresponds to the "forward functionïn the DDPM class.

The complete inference algorithm is presented in Alg. 18.2. Starting from Gaussian noise and then reversing the diffusion process step-by-step, by iteratively employing the update rule of Eq. 7. The code corresponds to the "sample functionïn the DDPM class.

Algorithm 18.1: Diffusion model training

Input: Training data $\mathbf{x}$
Output: Model parameters $\boldsymbol{\phi}_t$
repeat
  for $i \in \mathcal{B}$ **do** // For every training example index in batch
    $t \sim \text{Uniform}[1, \ldots T]$ // Sample random timestep
    $\boldsymbol{\epsilon} \sim \text{Norm}[\mathbf{0}, \mathbf{I}]$ // Sample noise
    $\ell_i = \left\| \mathbf{g}_t\left[\sqrt{\alpha_t}\mathbf{x}_i + \sqrt{1-\alpha_t}\boldsymbol{\epsilon}, \boldsymbol{\phi}_t\right] - \boldsymbol{\epsilon} \right\|^2$ // Compute individual loss
  Accumulate losses for batch and take gradient step
until converged

Algorithm 18.2: Sampling

Input: Model, $\mathbf{g}_t[\bullet, \boldsymbol{\phi}_t]$
Output: Sample, $\mathbf{x}$
$\mathbf{z}_T \sim \text{Norm}_{\mathbf{z}}[\mathbf{0}, \mathbf{I}]$ // Sample last latent variable
for $t = T \ldots 2$ **do**
  $\hat{\mathbf{z}}_{t-1} = \frac{1}{\sqrt{1-\beta_t}}\mathbf{z}_t - \frac{\beta_t}{\sqrt{1-\alpha_t}\sqrt{1-\beta_t}}\mathbf{g}_t[\mathbf{z}_t, \boldsymbol{\phi}_t]$ // Predict previous latent variable
  $\boldsymbol{\epsilon} \sim \text{Norm}_{\boldsymbol{\epsilon}}[\mathbf{0}, \mathbf{I}]$ // Draw new noise vector
  $\mathbf{z}_{t-1} = \hat{\mathbf{z}}_{t-1} + \sigma_t\boldsymbol{\epsilon}$ // Add noise to previous latent variable
$\mathbf{x} = \frac{1}{\sqrt{1-\beta_1}}\mathbf{z}_1 - \frac{\beta_1}{\sqrt{1-\alpha_1}\sqrt{1-\beta_1}}\mathbf{g}_1[\mathbf{z}_1, \boldsymbol{\phi}_1]$ // Generate sample from $\mathbf{z}_1$ without noise

Figuur 4: diffusion model training algorithm and sampling algorithm.

1.2 b

1.2.1 Training process

The training process follows three steps:

1. Data preparation: the MNIST dataset is loaded with tensor conversion and normalization and split into batches using DataLoader.

2. Model and training setup: an instance of a CNN model is created to serve as the ground truth estimator $g_t[z_t, \phi_t]$ within the DDPM class. A DDPM is then instantiated with beta values for the noise schedule and number of diffusion steps (`n_T`). An Adam optimizer is then configured to update the model parameters.

3. Training loop: the model is trained by iterating over batches from the DataLoader for each epoch. For each batch, gradients of the optimizer are reset at the start. A loss(equation 9),

between the actual noise in the data and the noise predicted by the CNN, is then computed using the DDPM previously instantiated. Next, the loss is backpropagated to update model parameters. Finally, the model generates and saves samples at regular intervals by reversing the diffusion process.

$$\text{loss} = ||g_t[\sqrt{\alpha_t}x + \sqrt{1-\alpha_t}\varepsilon, \phi_t] - \varepsilon||^2 \tag{9}$$

1.2.2 Experiment Overview

In this experiment, we aim to investigate the impact of different hyperparameter settings on the training of Denoising Diffusion Probabilistic Models (DDPMs). Specifically, we focus on variations in hidden layer sizes, noise schedules, and diffusion steps. To evaluate the effectiveness of each configuration, we rely on standard metrics such as the loss curve and the quality of the generated samples, assessed through the Fréchet Inception Distance (FID) score.

The FID score serves as a widely used metric for assessing the quality of generated samples, particularly in image generation tasks. It quantifies the similarity between the distributions of feature representations extracted from real and generated images by a pre-trained deep neural network like InceptionV3.

To compute the FID score, feature vectors are extracted from both real and generated images using the pre-trained neural network. These vectors encode high-level semantic information about the images. Subsequently, the FID score is calculated based on the Fréchet distance between the multivariate Gaussian distributions defined by the mean and covariance of these feature representations.

A lower FID score suggests a closer resemblance between the distributions of feature representations of generated images and real images, indicating higher quality and fidelity of the generated samples. Thus, the FID score provides a quantitative measure to evaluate and compare different generative models' performance.

In my study, I calculate the Fréchet Inception Distance (FID) between 128 original images and the generated samples from the trained model.

1.2.3 Experiment Details

1. **Default Hyperparameters**

   - Hidden layers: (16, 32, 32, 16)
   - Noise schedule betas: (1e-4, 0.02)
   - Diffusion steps: $n_T = 1000$

   Results: 100 epochs were run. Loss and FID score curves shown in Figures 5(a) and 6(a).

2. **Smaller Noise Schedule**

   - Hidden layers: (16, 32, 32, 16)
   - Noise schedule betas: (1e-4, 0.002)
   - Diffusion steps: $n_T = 1000$

   Results: The noise schedule is decreased to (1e-4, 0.002). 100 epochs were run. Loss and FID score curves shown in Figures 5(b) and 6(b).

3. **Larger Diffusion Steps**

   - Hidden layers: (16, 32, 32, 16)
   - Noise schedule betas: (1e-4, 0.02)
   - Diffusion steps: $n_T = 2000$

Results: The diffusion steps are increased to 2000. 100 epochs were run. Loss and FID score curves shown in Figures 5(c) and 6(c).

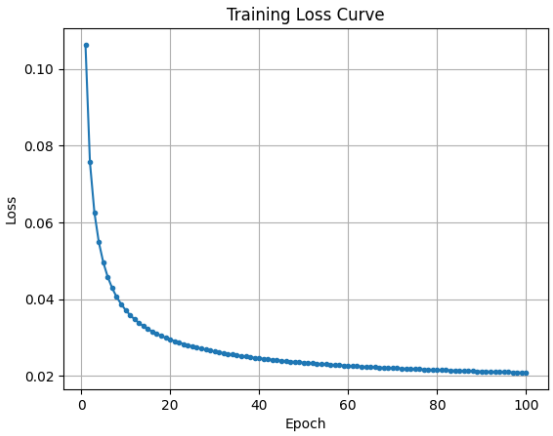
4. **Larger Hidden Layers**

   - Hidden layers: (64, 128, 128, 64)
   - Noise schedule betas: (1e-4, 0.002)
   - Diffusion steps: $n_T = 1000$

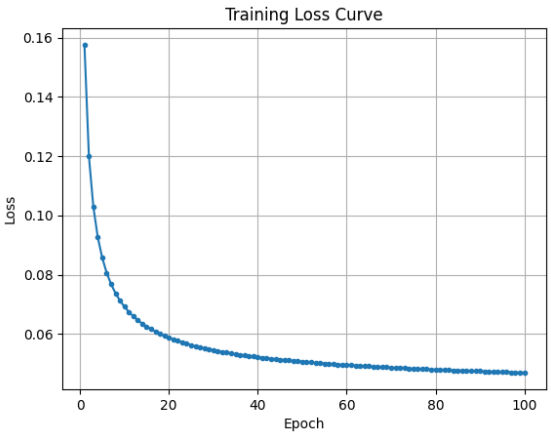
   Results: The hidden layers of the CNN are increased to (64, 128, 128, 64). Due to the larger CNN, the neural network becomes more complex, resulting in a more time-consuming training process. Therefore, only 38 epochs were run. Loss and FID score curves shown in Figures 5(d) and 6(d).

1.2.4 Results Presentation

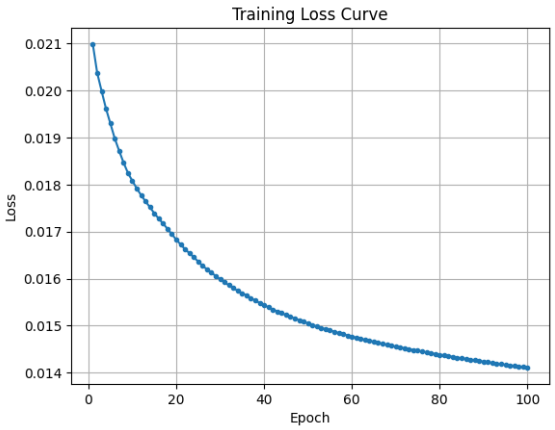
Loss Curve



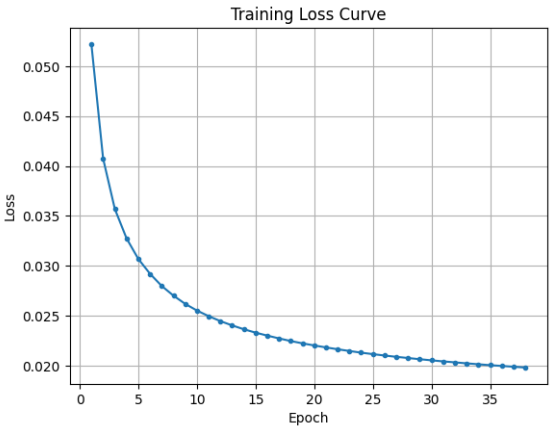
(a) Loss Curve of Default Hyperparameters



(b) Loss Curve of Smaller Noise Schedule



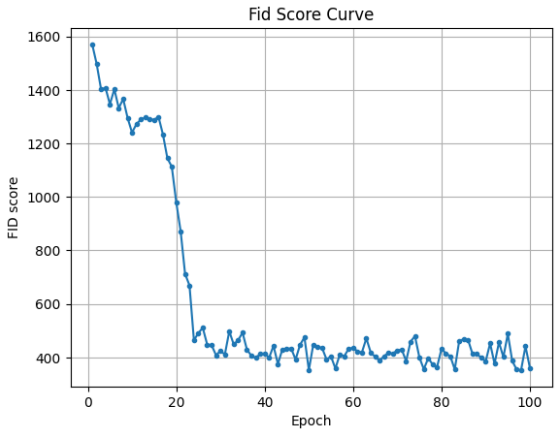
(c) Loss Curve of Larger Diffusion Steps



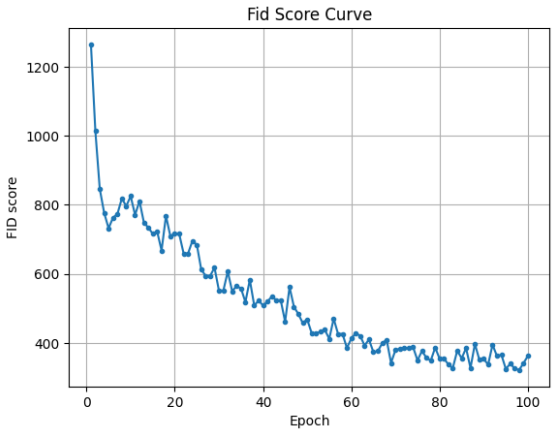
(d) Loss Curve of Larger Hidden Layers

Figuur 5: Loss Curves

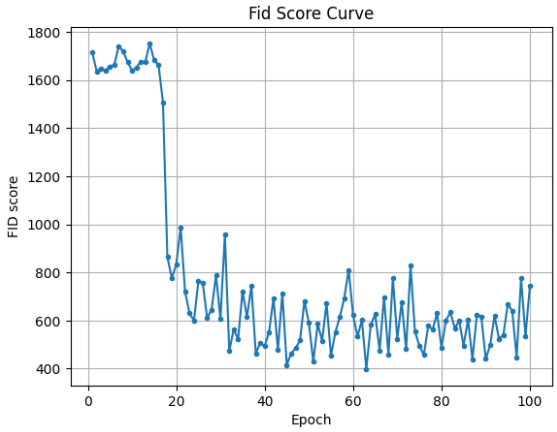
FID Score Curves



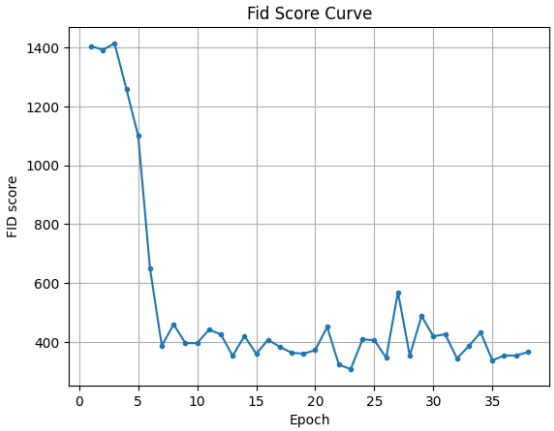
(a) FID Score Curve of Default Hyperparameters



(b) FID Score of Smaller Noise Schedule



(c) FID Score of Larger Diffusion Steps



(d) FID Score of Larger Hidden Layers

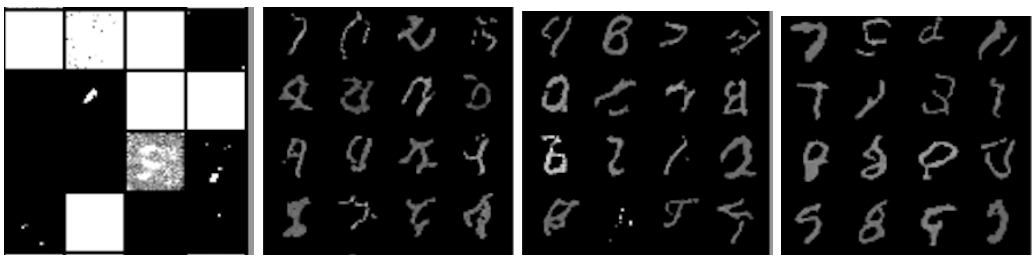
Figuur 6: FID Score

Generated Samples

Figure(7-10) show samples generated at different epochs for each configuration and provide visual insight into the progression of generated images.

Generated Samples of Default Hyperparameters:

Generated samples at epoch 1, 25, 50, and 100 are depicted in Figure 7. It took about 25 epochs to consistently generate symbols, and around 50 epochs to start seeing recognizable numbers.



(a) Epoch 1 (b) Epoch 25 (c) Epoch 50 (d) 100

Figuur 7: Generated Samples of Default Hyperparameters

Generated Samples of Smaller Noise Schedule:

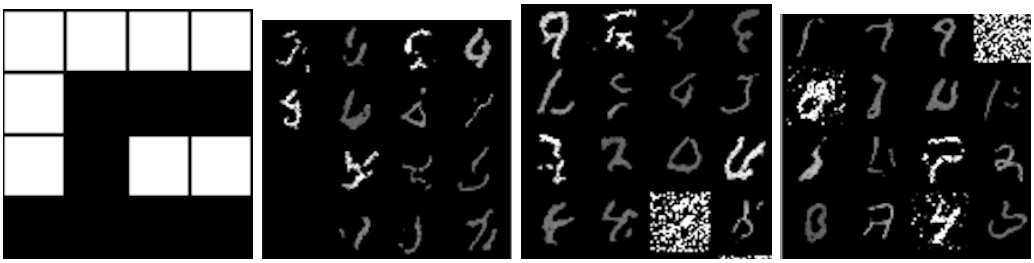
Generated samples at epoch 1, 16, and 100 are depicted in Figure 8. It needed about 1 epoch to start consistently generating symbols, and approximately 16 epochs to start seeing recognizable numbers.

(a) Epoch 1 (b) Epoch 16 (c) Epoch 100

Figuur 8: Generated Samples of Smaller Noise Schedule

Generated Samples of Larger Diffusion Steps:

Generated samples at epoch 1, 25, 41, and 100 are depicted in Figure 9. It needed about 25 epochs to consistently generate symbols, and approximately 41 epochs to start seeing recognizable numbers.

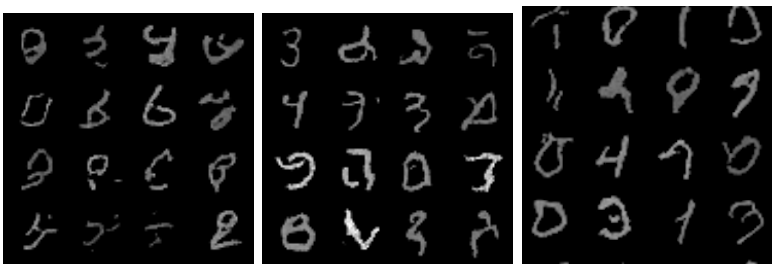


(a) Epoch 1 (b) Epoch 25 (c) Epoch 41 (d) Epoch 100

Figuur 9: Generated Samples of Larger Diffusion Steps

Generated Samples of Larger Hidden Layers:

Generated samples at epoch 2, 5, and 38 are depicted in Figure 10. It needed about 2 epochs to start consistently generating symbols, and approximately 5 epochs to start seeing recognizable numbers.



(a) Epoch 2 (b) Epoch 5 (c) Epoch 38

Figuur 10: Generated Samples of Larger Hidden Layers

1.2.5 Results Analysis

a. Difference in Loss Curves for Four Different Sets of Hyperparameters

1. **Default Hyperparameters:**

- The loss consistently decreases with increasing epochs. Notably, there's a rapid reduction in loss from epoch 0 to 20, followed by a more gradual decline until convergence. This convergence indicates successful training.

2. **Smaller Noise Schedule ($\beta = [0.0001, 0.02]$):**

   - The loss curve closely resembles that of the default hyperparameters. However, it exhibits slightly higher loss values.

3. **Larger Diffusion Steps ($n_T = 2000$):**

   - The loss curve converges to a smaller value compared to the default hyperparameters. This could be attributed to the complexity of the true reverse distributions $q(z_{t-1}|z_t)$ in the diffusion process, approximated as normal distributions. With smaller β_t values and a larger number of time steps T, the normal approximation becomes more accurate, resulting in a smaller loss.

4. **Larger Hidden Layers (n_hidden_layers = (64,128,128,64)):**

   - The loss curve converges faster than the default hyperparameters and reaches a lower final value. This phenomenon is likely due to the increased complexity afforded by larger hidden layers in the CNN. Consequently, the model can estimate g_t more accurately, leading to faster convergence and smaller loss values.

Difference in FID Score Curves for Four Different Sets of Hyperparameters

1. **Default Hyperparameters:**

   - Generally, the FID score decreases with epoch progression. Notably, from epoch 0 to epoch 25, a rapid decline is observed, followed by stabilization around FID = 400.

2. **Smaller Noise Schedule ($\beta = [0.0001, 0.02]$):**

   - Initially, there is a sharp decrease in the FID score from epoch 0 to epoch 5. Subsequently, a slower decrease with minor fluctuations is noted, converging to nearly FID = 400. The FID score is slightly lower than that of the default hyperparameters.

3. **Larger Diffusion Steps ($n_T = 2000$):**

   - Initially, the FID score fluctuates around 1700 from epoch 0 to epoch 15. A significant decrease is then observed from epoch 15 to epoch 20, followed by substantial fluctuations around 600 until epoch 100.

4. **Larger Hidden Layers ($n\_hidden\_layers = (64, 128, 128, 64)$):**

   - From epoch 0 to epoch 5, a rapid decline in the FID score is noted. Subsequently, from epoch 6 to epoch 38, it stabilizes around FID = 400, showcasing a smaller value compared to the FID score achieved with default hyperparameters.

Difference in Generated Samples for Four Different Sets of Hyperparameters

1. **Default Hyperparameters:**

   - Generated samples at epoch 1, 25, 50, and 100 are depicted in Figure 7. It took about 25 epochs to consistently generate symbols, and around 50 epochs to start seeing recognizable numbers.

2. **Smaller Noise Schedule ($\beta = [0.0001, 0.02]$):**

- Generated samples at epoch 1, 16, and 100 are depicted in Figure 8. It needed about 1 epoch to start consistently generating symbols, and approximately 16 epochs to start seeing recognizable numbers.

3. **Larger Diffusion Steps ($n_T = 2000$):**

   - Generated samples at epoch 1, 25, 41, and 100 are depicted in Figure 9. It needed about 25 epochs to consistently generate symbols, and approximately 41 epochs to start seeing recognizable numbers.

4. **Larger Hidden Layers ($n\_hidden\_layers = (64, 128, 128, 64)$):**

   - Generated samples at epoch 2, 5, and 38 are depicted in Figure 10. It needed about 2 epochs to start consistently generating symbols, and approximately 5 epochs to start seeing recognizable numbers.

1.3 c

I resent an analysis of each trained model in 1.2. I present both high quality and low quality samples from the trained model in figure 7 to figure 10.

2 Custom Degradation

2.1 a: Degradation Strategy using Gamma Distribution Noise

My degradation strategy involves adding gamma distribution noise instead of Gaussian noise. This strategy is inspired by a paper titled "Denoising Diffusion Gamma Models"by E. Nachmani. It expands the framework of diffusion generative processes by incorporating a new noise distribution, namely the Gamma Distribution. This introduces a new type of model called Denoising Diffusion Gamma Models.

First, let us define the Gamma diffusion process, and then I will present a way to sample from this process.

In the Gaussian case, the diffusion equation can be written as:

$$x_t = \sqrt{1-\beta_t}x_{t-1} + \sqrt{\beta_t}\epsilon_t \tag{10}$$

where ϵ_t is the Gaussian noise at step t.

One can denote $\Gamma(k, \theta)$ as the Gamma distribution, where k and θ are the shape and scale parameters, respectively. We modify Eq. 10 by adding noise that follows a Gamma distribution during the diffusion process:

$$x_t = \sqrt{1-\beta_t}x_{t-1} + (g_t - E(g_t)) \tag{11}$$

where $g_t \sim \Gamma(k_t, \theta_t)$, $\theta_t = \sqrt{\alpha_t}\theta_0$, and $k_t = \frac{\beta_t}{\alpha_t\theta_0^2}$. Note that θ_0 and β_t are hyperparameters, and the noise term has zero mean.

Since the sum of Gamma random variables (with the same scale parameter) is distributed as a Gamma distribution, one can derive a closed form for x_t, i.e., an equation to calculate x_t from x_0:

$$x_t = \sqrt{\alpha_t}x_0 + (\bar{g}_t - \bar{k}_t\theta_t) \tag{12}$$

where $\bar{g}_t \sim \Gamma(\bar{k}_t, \theta_t)$ and $\bar{k}_t = \prod_{i=1}^{t} k_i$.

Similarly to Eq. 7, the inference is given by:

$$X_{t-1} = \frac{x_t - \frac{1-\alpha_t}{\sqrt{1-\alpha_t}}\epsilon_t(x_t, \phi_t)}{\sqrt{\alpha_t}} + \sigma_t\frac{\bar{g}_t - E(\bar{g}_t)}{\sqrt{V(\bar{g}_t)}} \tag{13}$$

In Algorithm 3, I describe the training procedure. The input includes: (i) initial scale θ_0, (ii) the dataset d, (iii) the maximum number of steps in the diffusion process T, and (iv) the noise schedule $\beta_1, ..., \beta_T$. The training algorithm samples: (i) an example x_0, (ii) number of step t, and (iii) noise ϵ. Then it calculates x_t from x_0 using Eq. 12. The neural network $f_t(x_t, \phi_t)$ takes x_t as input and is conditional on the time step t. Next, it takes a gradient descent step to approximate the normalized noise

$$\frac{\bar{g}_t - \bar{k}_t\theta_t}{\sqrt{1-\alpha_t}} \tag{14}$$

with the neural network $\epsilon_t(x_t, \phi)$ (same as $g_t(x_t, \phi)$ in DDPM). The main changes between Algorithm 3 and the single Gaussian case (i.e., Alg. 18.1) are the following: (i) calculating the Gamma parameters, (ii) x_t update equation, and (iii) the gradient update equation.

The inference procedure is given in Algorithm 4. It starts from a zero-mean noise x_T sampled from $\Gamma(\theta_T, \bar{k}_T)$. Next, for T steps, the algorithm estimates x_{t-1} from x_t using Eq. 13. Note that as in (Song et al., 2020a), $\sigma_t = \sqrt{\beta_t}$. Algorithm 4 replaces the Gaussian version (i.e., Alg. 18.2) with the following: (i) the starting sampling point x_T, (ii) the sampling noise, and (iii) the x_t update equation.

Algorithm 3 Gamma Training Algorithm

Input: initial scale θ_0, dataset d, diffusion process length T, noise schedule $\beta_1, ..., \beta_T$
repeat
  $x_0 \sim d(x_0)$
  $t \sim \mathcal{U}(\{1, ..., T\})$
  $\bar{g}_t \sim \Gamma(\bar{k}_t, \theta_t)$
  $x_t = \sqrt{\bar{\alpha}_t}x_0 + (\bar{g}_t - \bar{k}_t\theta_t)$
  Take a gradient descent step on: $\left|\frac{\bar{g}_t - \bar{k}_t\theta_t}{\sqrt{1-|\bar{\alpha}_t|}} - \varepsilon_\theta(x_t, t)\right|$
until converged

Algorithm 4 Gamma Inference Algorithm

$\gamma \sim \Gamma(\theta_T, \bar{k}_T)$
$x_T = \gamma - \theta_T * \bar{k}_T$
for t = T, ..., 1 **do**
  $x_{t-1} = \frac{x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon(x_t,t)}{\sqrt{\alpha_t}}$
  if t > 1 **then**
    $z \sim \Gamma(\theta_{t-1}, \bar{k}_{t-1})$
    $z = \frac{z - \theta_{t-1}\bar{k}_{t-1}}{\sqrt{(1-\bar{\alpha}_t)}}$
    $x_{t-1} = x_{t-1} + \sigma_t z$
  end if
end for

Figuur 11: Gamma Training Algorithm and Gamma Inference Algorithm.

Denoising diffusion probabilistic models use the variational lower bound to minimize the negative log likelihood. Minimizing the variational lower bound for DDGM is equivalent to minimizing the L1 norm between the sampled noise and the estimated noise:

$$\text{loss} = |\frac{\bar{g}_t - \bar{k}_t\theta_t}{\sqrt{1-\alpha_t}} - \epsilon_t(x_t, \phi)| \tag{15}$$

Thus, the loss that is used in the Algorithm.3 is given by equation(15).

2.2 b: Experiments and Results

2.2.1 Training process

The training process follows three steps:

1. Data preparation: the MNIST dataset is loaded with tensor conversion and normalization and split into batches using DataLoader.

2. Model and training setup: an instance of a CNN model is created to serve as the ground truth estimator $\epsilon_\theta[x_t, t]$ within the DDGM class. A DDGM is then instantiated with beta values for the noise schedule and number of diffusion steps (`n_T`). An Adam optimizer is then configured to update the model parameters.

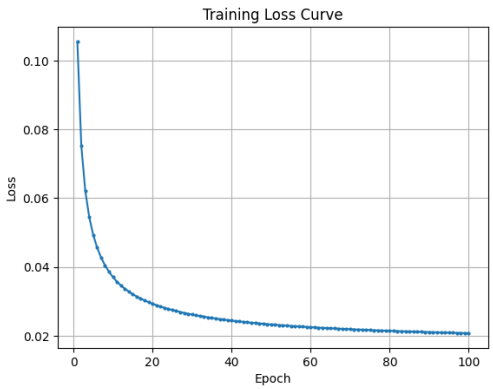
3. Training loop: the model is trained by iterating over batches from the DataLoader for each epoch. For each batch, gradients of the optimizer are reset at the start. A loss(equation 15), between the actual noise in the data and the noise predicted by the CNN, is then computed using the DDGM previously instantiated. Next, the loss is backpropagated to update model parameters. Finally, the model generates and saves samples at regular intervals by reversing the diffusion process.

2.2.2 Experiment settings

Similar to (Song et al., 2020a), the training noise schedule $\beta_1, ..., \beta_T$ is linear with values ranging from 0.0001 to 0.02. The hidden layers are set as (16, 32, 32, 16). For other hyperparameters (e.g., learning rate, batch size, etc.), I use the same parameters as those in DDPM with default hyperparameters. I use one T4 GPU provided in Google Colab to train our models. The θ_0 parameter for the Gamma distribution is set to 0.001.

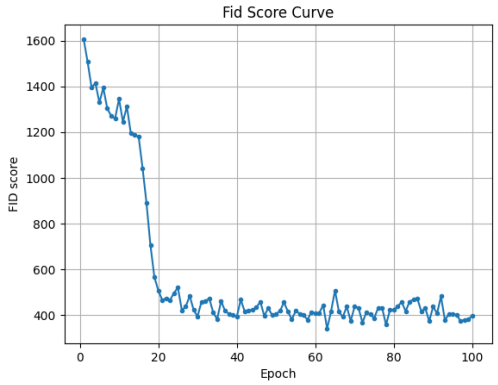
2.2.3 Results and Analysis

I ran 100 epochs and plotted the average loss of each epoch. The loss curve is shown in Figure 12. The loss consistently decreases with increasing epochs. Notably, there's a rapid reduction in loss from epoch 0 to 20, followed by a more gradual decline until convergence. This convergence indicates successful training.



Figuur 12: Loss Curve of DDGM.

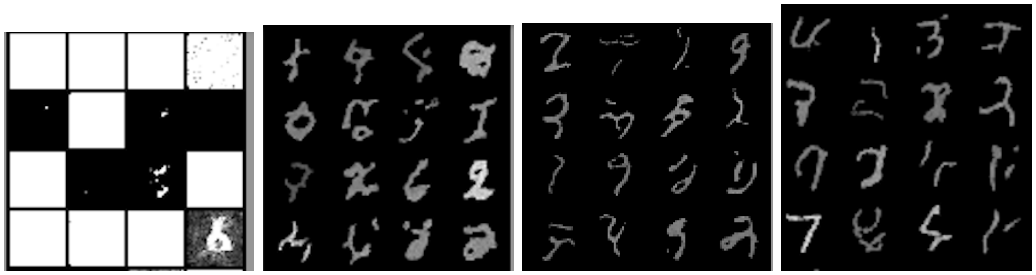
Additionally, I generated 128 samples each epoch and used these generated samples along with the original images from the MNIST dataset to calculate the Fréchet Inception Distance (FID) score, evaluating the quality of the generated samples. The FID curve is presented in Figure 13. Generally, the FID score decreases with epoch progression. Notably, from epoch 0 to epoch 20, a rapid decline is observed, followed by stabilization around FID = 400.



Figuur 13: FID Score Curve of DDGM.

In Figure 14, I present some generated samples (at epoch 1, epoch 19, epoch 40, and epoch 100) obtained using gamma noise.

When trained with the default hyperparameters, I found that it needed approximately 19 epochs to start consistently generating symbols, although those symbols did not resemble numbers yet. It took about 40 epochs until I started seeing recognizable numbers being generated.



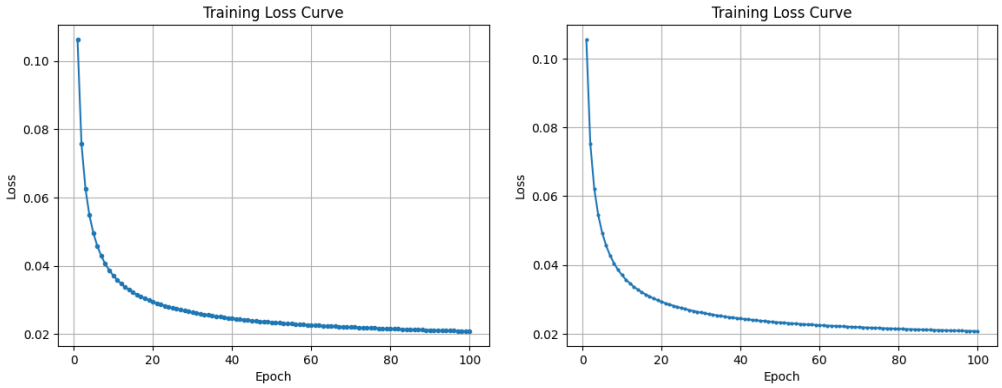
(a) Epoch 1 (b) Epoch 19 (c) Epoch 40 (d) Epoch 100

Figuur 14: Generated Samples of DDGM

2.3 c: Comparison

2.3.1 Comparison of the Loss Curve

From Figure 15, we can see that the loss curve is very similar between DDPM and DDGM.

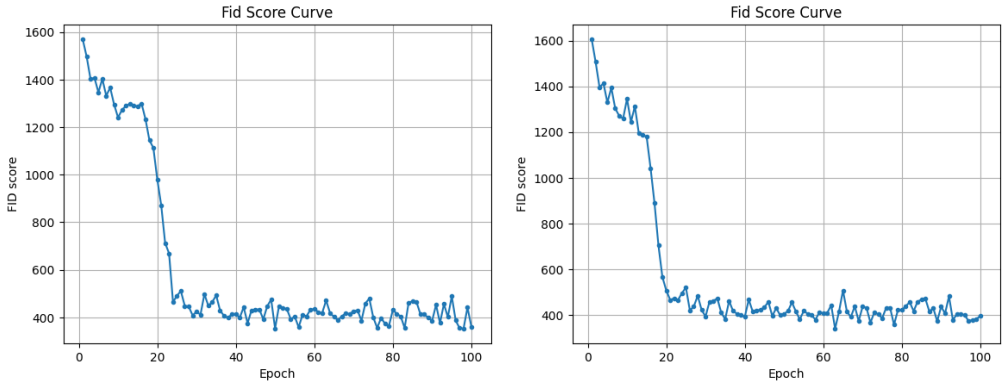


(a) Loss curve of DDPM (b) Loss curve of DDGM

Figuur 15: Comparison of the Loss Curve

2.3.2 Comparison of the FID Score Curve

From Figure 16, we observe that, in general, the FID score curve is very similar between DDGM and DDPM. However, the FID score of DDGM decreases faster than DDPM, with DDGM reaching a nearly FID score of 400 after approximately 19 epochs, while DDPM requires 25 epochs.



(a) FID Score Curve of DDPM (b) FID Score Curve of DDGM

Figuur 16: Comparison of the FID Score Curve

2.3.3 Comparison of the Generated Samples

After comparing the generated samples between DDPM and DDGM, I observed that DDGM can produce high-fidelity samples faster. In DDPM, it takes around 25 epochs to start seeing symbols generated, whereas in DDGM, it only takes around 18 epochs to start seeing symbols and numbers being generated.