Fast Sampling Diffusion on large quantities of images using an additional layer

1. Abstract

In recent times, generative models have become a hot topic in the field of Artificial Intelligence. Namely, there is a recent surge in interest regarding the topic of Diffusion models, said to be able to generate images of higher quality compared to Generative Adversarial Networks (GANs). However, a known limitation about the diffusion models is the inferior sampling speed compared to other generative models. Many research papers aim to improve this sampling speed by reducing the sampling steps by modifying mathematical formulas in diffusion models. However, we have found no research papers that modify the architecture of the diffusion models to increase sampling speed. By adding an additional layer into the model, we aim to discover the effects that it has on the sampling speed and quality of the images in this paper.

2. Introduction

With the rapid development in the field of Artificial Intelligence, AI models have become more versatile in terms of functionality, a primary example being generative models. Generative models can generate like images, text, audio or even videos that are ideally similar to the original data when training. One of the more recent generative models released is the diffusion model. Diffusion models utilize the idea of parameterized Markov chains trained using variational inference to produce samples matching the data after finite time (Ho et al., 2020). Essentially, this means that the diffusion model performs the process of adding noise to their training data (known as the forward diffusion process) and recovering the data to its original form (known as the reverse diffusion process). The whole point of the model is to be able to recover the image by learning to remove the gaussian noise added during the training period. Through this, diffusion models will be able to generate new images that are unseen in the training dataset.

A significant issue that occurs when dealing with diffusion models is the sampling speed. As shown in our mathematical formulas (1.4), each iteration is dependent on the previous step in both the diffusion and denoising process. Since diffusion models train in an iterative nature, it takes multiple iterations to generate just one image, causing the sampling process to be extremely slow. There are samplers that are built to specifically deal with such issues, such as the Denoising Diffusion Implicit Models (DDIM). The main idea of DDIM is to replace the explicit representation of the diffusion process with implicit probabilistic models that have the same training procedures as Denoising Diffusion Probabilistic Mode (DDPM). DDIMs construct a class of non-Markovian diffusion processes that have the same training objective as DDPMs (which is Markovian) but can generate images much faster and with equal or better quality (Nichol & Dhariwal, 2021). There are also other samplers such as DPM-solver (Lu et al., 2022), which can generate quality images with few steps (approximately 20 steps) just by using mathematical equations, increasing the sampling speed by a marginal amount. There are also other samplers like Euler, PLMS, LMS Karras, and Heun which help in speeding up the sampling speed (Andrew, 2023).

The reason we have chosen to focus on the aspect of speed in diffusion models is because recent research mainly focusses on reducing the sampling steps by changing the mathematic formulation of the reverse diffusion, but none of them reduce the sampling steps by changing the model architecture. Thus, our objective for this project is to add an additional layer to increase the sampling speed when sampling a large sample of images at once while maintaining the sample quality.

3. Background

3.1 Diffusion model

The diffusion model is a generative model that outputs the noise to diffuse an image. Then, the noise will pass through some formula to reconstruct the image. The training of the diffusion model is divided into 2 steps, forward diffusion, $q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})$ (process of adding noise) and backward diffusion, $p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ (process of removing noise). The detailed proof for diffusion model will be located at Appendix A

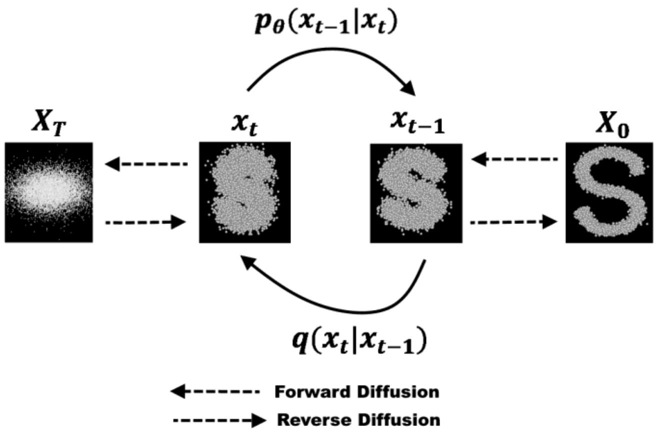


Image Sources

2.1.1 Forward Diffusion

Let $\boldsymbol{x}_0 \sim q(\boldsymbol{x})$ represent the data point sampled from a real distribution $q(\boldsymbol{x})$. The forward diffusion process is defined by a process by which small amounts of Gaussian noise are added to the sample over T time steps. This process produces a sequence of noisy samples $\{\boldsymbol{x}_t\}_{t=1}^{T}$. Then the posterior $q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})$ is given by

$$q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}) = \mathcal{N}\left(\boldsymbol{x}_t; \boldsymbol{\mu} = \sqrt{1-\beta_t}\boldsymbol{x}_{t-1}, \boldsymbol{\Sigma} = \beta_t \boldsymbol{I}\right) \tag{1.1}$$

where the step sizes over T are controlled by a variance schedule $\{\beta_t \in (0,1)\}_{t=1}^{T}$. We can then use equation (1.1) and the Markov property to formulate the posterior distribution $q(\boldsymbol{x}_1, \boldsymbol{x}_2 \ldots \boldsymbol{x}_T|\boldsymbol{x}_0)$ or simplified notation, $q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)$. It is given by

$$\begin{aligned} q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0) &= q(\boldsymbol{x}_1|\boldsymbol{x}_0)q(\boldsymbol{x}_2|\boldsymbol{x}_1, \boldsymbol{x}_0) \ldots q(\boldsymbol{x}_T|\boldsymbol{x}_{T-1}, \boldsymbol{x}_{T-2}, \ldots, \boldsymbol{x}_0) \\ &= q(\boldsymbol{x}_1|\boldsymbol{x}_0)q(\boldsymbol{x}_2|\boldsymbol{x}_1) \ldots q(\boldsymbol{x}|\boldsymbol{x}_{T-1}) \\ &= \prod_{t=1}^{T} q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}) \end{aligned} \tag{1.2}$$

The distinguishable features of $\boldsymbol{x}_0$ tend to diminish gradually as t becomes larger and the variance of $\boldsymbol{x}_T$ is said to be uniformly distributed across all features as $T \to \infty$. In such cases, we call $\boldsymbol{x}_T$ as an isotropic Gaussian distribution. By a convenient property of Gaussian distribution, we can obtain the posterior distribution in any arbitrary time step t, i.e., $q(\boldsymbol{x}_t|\boldsymbol{x}_0)$, by using the following property:

$$\boxed{q(\boldsymbol{x}_t|\boldsymbol{x}_0) = \mathcal{N}\left(\boldsymbol{x}_t; \sqrt{\bar{\alpha}_t}\boldsymbol{x}_0, (1-\bar{\alpha}_t)\boldsymbol{I}\right)} \tag{1.3}$$

where $\alpha_t = 1 - \beta_t$ and $\bar{\alpha}_t = \prod_{r=0}^{t-1} \alpha_{t-r}$

2.1.2 Reverse Diffusion

In reverse diffusion process, the goal is to reconstruct the real (true) sample from a Gaussian noise input, $x_T \sim \mathcal{N}(0, I)$ by reversing the process and sampling from $q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$. Unfortunately, estimating $q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ is a difficult task because it requires the entire dataset when applying Bayes' Theorem. Therefore, we have to learn from an estimation model, namely, p_θ such that

$$p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t) \sim \mathcal{N}\left(\boldsymbol{x}_{t-1}; \boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t), \boldsymbol{\Sigma}_\theta(\boldsymbol{x}_t, t)\right) \tag{1.4}$$

and

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

(1.5)

However, the posterior $q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)$ is trackable when it is conditioned on $\boldsymbol{x}_0$, In other words, $q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \boldsymbol{x}_0)$ can be obtained by applying Bayes' theorem. i.e.

$$\boxed{q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \boldsymbol{x}_0) = \mathcal{N}\left(\boldsymbol{x}_{t-1}; \tilde{\boldsymbol{\mu}}(\boldsymbol{x}_t, \boldsymbol{x}_0), \tilde{\beta}_t\boldsymbol{I}\right)}$$

(1.6)

where the mean vector $\tilde{\boldsymbol{\mu}}(\boldsymbol{x}_t, \boldsymbol{x}_0)$ and covariance matrix $\tilde{\beta}_t\boldsymbol{I}$ are given by

$$\boxed{\tilde{\boldsymbol{\mu}}(\boldsymbol{x}_t, \boldsymbol{x}_0) = \frac{1}{\sqrt{\alpha_t}}\left(\boldsymbol{x}_t + \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_t\right), \qquad \tilde{\beta}_t\boldsymbol{I} = \frac{\beta_t(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\boldsymbol{I}}$$

where $\boldsymbol{\epsilon}_t = \frac{x_0\sqrt{\bar{\alpha}_t} - x_t}{\sqrt{1-\bar{\alpha}_t}}$ = error at time step t

2.1.2 Loss function of Diffusion model

The formulation of the loss function involves finding the evidence lower bound (ELBO) of the estimating model $p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t) \sim \mathcal{N}\left(\boldsymbol{x}_{t-1}; \boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t), \boldsymbol{\Sigma}_\theta(\boldsymbol{x}_t, t)\right)$ and optimize it. However, the final loss function is fairly simple, which can be shown as:

$$\boxed{L_{simple} = \sum_{t=2}^{T} \left\|\boldsymbol{\epsilon}_\theta\left(\sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \left(\sqrt{1-\bar{\alpha}_t}\right)\cdot\boldsymbol{\epsilon}_t, t\right) - \boldsymbol{\epsilon}_t\right\|^2 + C}$$

(1.7)

where $\boldsymbol{\epsilon}_\theta$ is the output of the model and $\boldsymbol{\epsilon}_t$ is the actual error at timestep t

The formulation of this loss function is intuitively correct as the model is just minimizing the MSE of the reconstructed output with original error.

3.2 Connection of Diffusion model with Noise Conditioned score networks

Noise Conditioned Score Networks (NCSN) is a method used to formulate the loss function of the score function, $\nabla x \log p(x)$ (Ayan Das, 2021). The score function can provide information about local density and directionality of the data distribution. The idea of noise conditioned score networks (NCSN) are introduced to be combined with the diffusion model. The score function of $q(x_t|x_0) \sim \mathcal{N}\left(\sqrt{\overline{\alpha}_t}x_0, (1-\overline{\alpha}_t)I\right)$ is

$$\boxed{\mathbf{s}_\theta(\mathbf{x}_t, t) \approx \nabla_{\mathbf{x}_t} \log q\,(\mathbf{x}_t) = -\frac{\boldsymbol{\epsilon}_\theta(\mathbf{x}_t, t)}{\sqrt{1-\overline{\alpha}_t}}} \tag{1.8}$$

When we compare with the loss function of diffusion model (1.7) and the formula that is shown above, it shows that the diffusion model is just training the score function! The detailed proof of the score function is shown at Appendix B

3.3 Progressive Distillation for Fast Sampling of Diffusion Models

Progressive Distillation is an algorithm that aims to improve the sampling speed of diffusion models. Through this algorithm, it can iteratively half the number of required sampling steps. This is done by distilling a slow teacher diffusion model into a faster student model (Salimans & Ho, 2022). In our code implementation, the 'v' objective will be used but with reweights of loss function.

4. Methodology

4.1 General Flow

<u>**Part 1 (For Folder_1)**</u>

a) The source code of the implementation of the diffusion model on CIFAR-10 is obtained from Google Colab and is placed in Folder_1. This code implements the diffusion model using the U-net.

b) The source code is first modified by following:

i. Modifying the training loop so that it will stop training if validation loss continues to decrease for MAX_COUNT times

ii. Add in Settings section to make the process of tuning the settings easier

iii. Add the evaluation section using inception score

iv. Change the demo() function to accept argument about number of reconstruct step and number of sample

c) The diffusion model in the source code is trained. Its inception score on reconstructed images for different number of reconstructed step and number of images are computed as well.

<u>**Part 2 (For Folder_2)**</u>

d) The source code in Folder_1 is cloned again to Folder_2 and the label information passed to the model is removed. Then, the diffusion model is trained again and inception scores on the reconstructed images for different number of reconstructed step and number of images are computed.

e) Folder_1 and Folder_2 will act as a baseline for further implementation

Part 3 (For Folder_3)

f) After that, the pre-trained model and source code in Folder_2 is cloned to Folder_3.

g) The source code in Folder_3 is modified by applying the following:

i. An additional layer is added for unconditioned, pre-trained U-net.

ii. Changes on the demo() and

iii. Add in sample_new_v1() and sample_new_v2 () function to increase the sample speed.

iv. Changes the eval_loss() function by pass the output of U-Net toward additional layer

h) The U-net in Folder_3 is frozen, and the additional layer is trained. Then, inception scores on reconstructed images for different number of reconstructed step and number of images are computed using sample_new_v1() function and sample_new_v2() function.

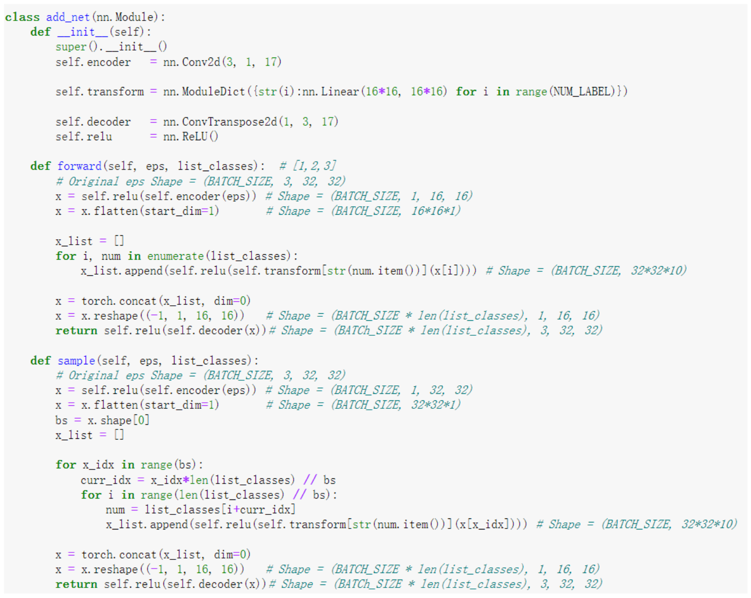
Part 4 (Overall Interpretation)

i) The results obtained from Folder_1, Folder_2 and Folder_3 are placed together for comparison and interpretation. The results are shown at the Validation/Verification Section

4.2 Detailed implementation on our changes

Our main contribution is the changes done on Folder_3. Therefore, in this section, we will explain the changes in detail with the code implementation.

a) The details of additional layer implemented in pytorch is at below:

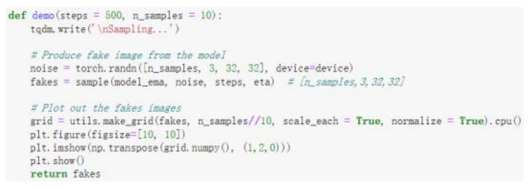
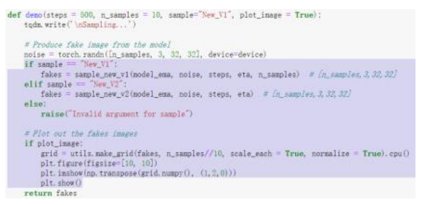


```
class add_net(nn.Module):
    def __init__(self):
        super().__init__()
        self.encoder   = nn.Conv2d(3, 1, 17)

        self.transform = nn.ModuleDict({str(i):nn.Linear(16*16, 16*16) for i in range(NUM_LABEL)})

        self.decoder   = nn.ConvTranspose2d(1, 3, 17)
        self.relu      = nn.ReLU()

    def forward(self, eps, list_classes):  # [1,2,3]
        # Original eps Shape = (BATCH_SIZE, 3, 32, 32)
        x = self.relu(self.encoder(eps)) # Shape = (BATCH_SIZE, 1, 16, 16)
        x = x.flatten(start_dim=1)       # Shape = (BATCH_SIZE, 16*16*1)

        x_list = []
        for i, num in enumerate(list_classes):
            x_list.append(self.relu(self.transform[str(num.item())](x[i]))) # Shape = (BATCH_SIZE, 32*32*10)

        x = torch.concat(x_list, dim=0)
        x = x.reshape((-1, 1, 16, 16))   # Shape = (BATCH_SIZE * len(list_classes), 1, 16, 16)
        return self.relu(self.decoder(x))# Shape = (BATCh_SIZE * len(list_classes), 3, 32, 32)

    def sample(self, eps, list_classes):
        # Original eps Shape = (BATCH_SIZE, 3, 32, 32)
        x = self.relu(self.encoder(eps)) # Shape = (BATCH_SIZE, 1, 32, 32)
        x = x.flatten(start_dim=1)       # Shape = (BATCH_SIZE, 32*32*1)
        bs = x.shape[0]
        x_list = []

        for x_idx in range(bs):
            curr_idx = x_idx*len(list_classes) // bs
            for i in range(len(list_classes) // bs):
                num = list_classes[i+curr_idx]
                x_list.append(self.relu(self.transform[str(num.item())](x[x_idx]))) # Shape = (BATCH_SIZE, 32*32*10)

        x = torch.concat(x_list, dim=0)
        x = x.reshape((-1, 1, 16, 16))   # Shape = (BATCH_SIZE * len(list_classes), 1, 16, 16)
        return self.relu(self.decoder(x))# Shape = (BATCH_SIZE * len(list_classes), 3, 32, 32)
```

- forward() function: This method performs the forward pass through the additional layers. It takes the eps tensor (original input) and list_classes as inputs and outputs the eps tensor that points to the target_classes. The eps tensor is passed through the encoder layer which encodes eps from shape (BATCH_SIZE, 3, 32, 32) to (BATCH_SIZE, 1, 16,16), flattened it, and then transformed by the linear layers in self.transform. The output of each linear layer is concatenated along the batch dimension. Finally, the concatenated tensor is reshaped and passed through the decoder layer, and the result with same size as input is returned.

- sample() function: This method is similar to the forward method but handles sampling when the list_classes length is greater than the batch size. list_classes is usually set to be [0,1,2,3,4,5,6,7,8,9], and eps will be produced by passing n image into U-net. After the sample() function, it will produce 10*n eps tensors that point to 10 difference classes. The procedure of sample function is as follows: it iterates over each element in the batch, selects the corresponding indices from list_classes, and performs the transformation and concatenation steps similar to the forward method.

b) The details of implementation of demo() function is at below:

	
demo() function in Folder_2	demo() function in Folder_3 (Ours)

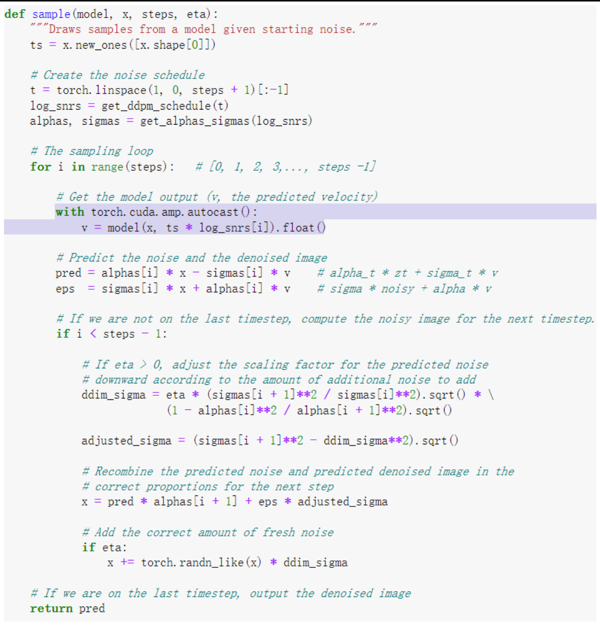
```
def demo(steps = 500, n_samples = 10):
    tqdm.write('\nSampling...')

    # Produce fake image from the model
    noise = torch.randn([n_samples, 3, 32, 32], device=device)
    fakes = sample(model_ema, noise, steps, eta)  # [n_samples, 3, 32, 32]

    # Plot out the fakes images
    grid = utils.make_grid(fakes, n_samples//10, scale_each = True, normalize = True).cpu()
    plt.figure(figsize=[10, 10])
    plt.imshow(np.transpose(grid.numpy(), (1,2,0)))
    plt.show()
    return fakes
```

```
def demo(steps = 500, n_samples = 10, sample="New_V1", plot_image = True):
    tqdm.write('\nSampling...')

    # Produce fake image from the model
    noise = torch.randn([n_samples, 3, 32, 32], device=device)
    if sample == "New_V1":
        fakes = sample_new_v1(model_ema, noise, steps, eta, n_samples)  # [n_samples, 3, 32, 32]
    elif sample == "New_V2":
        fakes = sample_new_v2(model_ema, noise, steps, eta)  # [n_samples, 3, 32, 32]
    else:
        raise("Invalid argument for sample")

    # Plot out the fakes images
    if plot_image:
        grid = utils.make_grid(fakes, n_samples//10, scale_each = True, normalize = True).cpu()
        plt.figure(figsize=[10, 10])
        plt.imshow(np.transpose(grid.numpy(), (1,2,0)))
        plt.show()
    return fakes
```

The `demo` function is used to generate samples from the model and display the resulting images. We add 2 additional parameters:

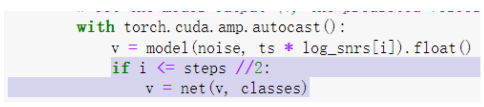
- sample ∈ ("New_V1" or "New_V1") to choose whether use sample_new_v1() function or sample_new_v2() function
- a flag to control whether to plot the images.

c) The details of implementation of sample_new_v1 / v2() function is at below:



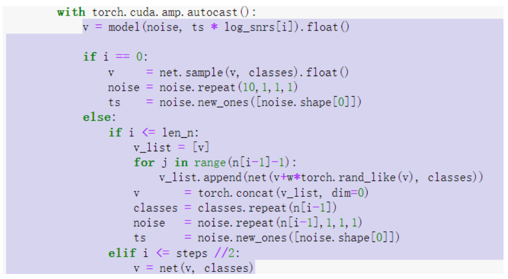
```python
def sample(model, x, steps, eta):
    """Draws samples from a model given starting noise."""
    ts = x.new_ones([x.shape[0]])

    # Create the noise schedule
    t = torch.linspace(1, 0, steps + 1)[:-1]
    log_snrs = get_ddpm_schedule(t)
    alphas, sigmas = get_alphas_sigmas(log_snrs)

    # The sampling loop
    for i in range(steps):   # [0, 1, 2, 3,..., steps -1]

        # Get the model output (v, the predicted velocity)
        with torch.cuda.amp.autocast():
            v = model(x, ts * log_snrs[i]).float()

        # Predict the noise and the denoised image
        pred = alphas[i] * x - sigmas[i] * v    # alpha_t * zt + sigma_t * v
        eps  = sigmas[i] * x + alphas[i] * v    # sigma * noisy + alpha * v

        # If we are not on the last timestep, compute the noisy image for the next timestep.
        if i < steps - 1:

            # If eta > 0, adjust the scaling factor for the predicted noise
            # downward according to the amount of additional noise to add
            ddim_sigma = eta * (sigmas[i + 1]**2 / sigmas[i]**2).sqrt() * \
                         (1 - alphas[i]**2 / alphas[i + 1]**2).sqrt()

            adjusted_sigma = (sigmas[i + 1]**2 - ddim_sigma**2).sqrt()

            # Recombine the predicted noise and predicted denoised image in the
            # correct proportions for the next step
            x = pred * alphas[i + 1] + eps * adjusted_sigma

            # Add the correct amount of fresh noise
            if eta:
                x += torch.randn_like(x) * ddim_sigma

    # If we are on the last timestep, output the denoised image
    return pred
```

Original Ways of sample()



```python
        with torch.cuda.amp.autocast():
            v = model(noise, ts * log_snrs[i]).float()
            if i <= steps //2:
                v = net(v, classes)
```

sample_new_v1()



```python
        with torch.cuda.amp.autocast():
            v = model(noise, ts * log_snrs[i]).float()

            if i == 0:
                v     = net.sample(v, classes).float()
                noise = noise.repeat(10,1,1,1)
                ts    = noise.new_ones([noise.shape[0]])
            else:
                if i <= len_n:
                    v_list = [v]
                    for j in range(n[i-1]-1):
                        v_list.append(net(v+w*torch.rand_like(v), classes))
                    v       = torch.concat(v_list, dim=0)
                    classes = classes.repeat(n[i-1])
                    noise   = noise.repeat(n[i-1],1,1,1)
                    ts      = noise.new_ones([noise.shape[0]])
                elif i <= steps //2:
                    v = net(v, classes)
```

sample_new_v2()

<u>The original ways of sample()</u>

sample() starts reconstructing the image from the random noise,

for i in range(number of reconstructed step, ‘steps’)

- pass the image, x and the noise scheduler log_snrs to the model and obtain the velocity, v.
- After that, v is pass through some mathematical formula to obtain x and the predicted images, pred.

Finally, the predicted images, pred is return.

sample_new_v1()

The only difference between sample() and sample_new_v1() is we passed the output from the U-net, v to the additional layer if i <= 'steps'//2. This will guide the velocity, v, to point to the direction with a specific label for the first 'steps'//2 steps.

sample_new_v2()

For this sample method, it requires

- the number of images generated to be divisible by NUM_LABEL=10
- requires 1 additional hyperparameter, degree of noise added to velocity, w. In our case, we set w = 0.01.

The process of sampling is as below:

```
n = find_all_factor(num_image//10)
n.reverse()
```

First, integer factorization is performed on the integer: (num_image//10). For example, if num_image//10=50, factorizing it will produce 2,5,5.

Then, the list of factors is reverse()

After that, we call called the sample_new_v2() function to do sampling. The difference between sample() and sample_new_v2() is just the block of code inside 'with torch.cuda.amp.autocast():'.

- For each iteration, the images will be passed to the model (U-Net) to obtain velocity, v.
- Then, in the first iteration (i=0), it will use the add_net model's sample method to get input of m number of velocity tensor and output 10*m velocity tensor that points to 10 different labels.
- For the iteration from (i=1 ~ i=steps//2),
  - if i < (total number of factor):

    it will pass v + w*noise to the additional layer for n[i-1] – 1 time for difference noise. Generally, it takes m number of velocity, v and output m*n[i-1] number of velocity where n[i-1] is a factor of num_image//10
  - elif i <= steps//2:

it will pass the output from the U-net, v to the additional layer to guide the velocity

An example of producing 100 images is as below:

1. First, we obtain the factor of 100//10 which are [5,2]
2. Then, we pass 1 noisy image to the U-net and produce 1 velocity, v
3. (Iteration 1), it will get 1 velocity and output 10 velocity towards different labels and produce 10 images after performing some formula
4. (Iteration 2), it will get 10 velocity and output 10*5 = 50 velocity
5. (Iteration 3), it will get 50 velocity and output 50*2 = 100 velocity.
6. For the remaining iteration, it will pass the image to U-Net and then pass through additional layer if i <= steps//2

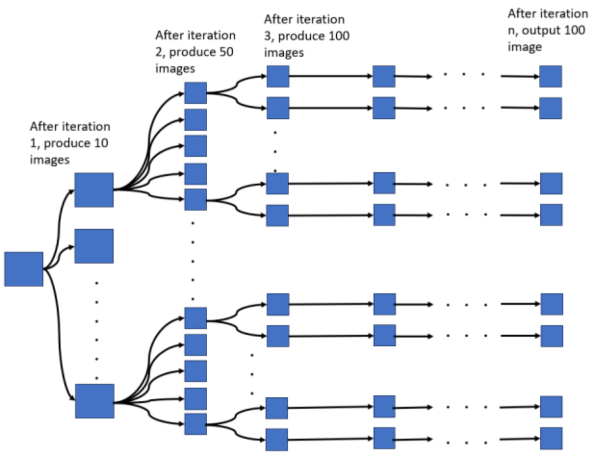
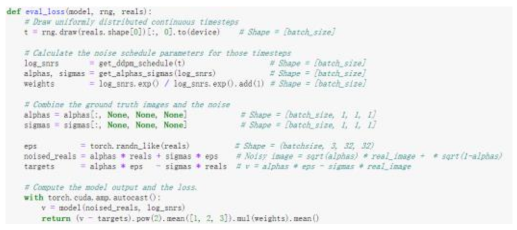
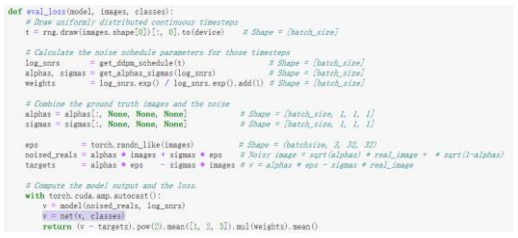

After iteration 1, produce 10 images
After iteration 2, produce 50 images
After iteration 3, produce 100 images
After iteration n, output 100 image

Figure to illustrate the process

d) The details of implementation of eval_loss () function is at below:

	
eval_loss() function in Folder_2	Ours

```python
def eval_loss(model, rng, reals):
    # Draw uniformly distributed continuous timesteps
    t = rng.draw(reals.shape[0])[:, 0].to(device)    # Shape = [batch_size]

    # Calculate the noise schedule parameters for those timesteps
    log_snrs       = get_ddpm_schedule(t)                          # Shape = [batch_size]
    alphas, sigmas = get_alphas_sigmas(log_snrs)                   # Shape = [batch_size]
    weights        = log_snrs.exp() / log_snrs.exp().add(1) # Shape = [batch_size]

    # Combine the ground truth images and the noise
    alphas = alphas[:, None, None, None]              # Shape = [batch_size, 1, 1, 1]
    sigmas = sigmas[:, None, None, None]              # Shape = [batch_size, 1, 1, 1]

    eps          = torch.randn_like(reals)            # Shape = (batchsize, 3, 32, 32)
    noised_reals = alphas * reals + sigmas * eps      # Noisy image = sqrt(alphas) * real_image +  * sqrt(1-alphas)
    targets      = alphas * eps   - sigmas * reals    # v = alphas * eps - sigmas * real_image

    # Compute the model output and the loss.
    with torch.cuda.amp.autocast():
        v = model(noised_reals, log_snrs)
        return (v - targets).pow(2).mean([1, 2, 3]).mul(weights).mean()
```

```python
def eval_loss(model, images, classes):
    # Draw uniformly distributed continuous timesteps
    t = rng.draw(images.shape[0])[:, 0].to(device)    # Shape = [batch_size]

    # Calculate the noise schedule parameters for those timesteps
    log_snrs       = get_ddpm_schedule(t)                          # Shape = [batch_size]
    alphas, sigmas = get_alphas_sigmas(log_snrs)                   # Shape = [batch_size]
    weights        = log_snrs.exp() / log_snrs.exp().add(1) # Shape = [batch_size]

    # Combine the ground truth images and the noise
    alphas = alphas[:, None, None, None]              # Shape = [batch_size, 1, 1, 1]
    sigmas = sigmas[:, None, None, None]              # Shape = [batch_size, 1, 1, 1]

    eps          = torch.randn_like(images)           # Shape = (batchsize, 3, 32, 32)
    noised_reals = alphas * images + sigmas * eps     # Noisy image = sqrt(alphas) * real_image +  * sqrt(1-alphas)
    targets      = alphas * eps   - sigmas * images # v = alphas * eps - sigmas * real_image

    # Compute the model output and the loss.
    with torch.cuda.amp.autocast():
        v = model(noised_reals, log_snrs)
        v = net(v, classes)
        return (v - targets).pow(2).mean([1, 2, 3]).mul(weights).mean()
```

The only change applied to our loss function is just passing the output from the U-net to the additional layer that we implemented.

Work segregated among team members.

Name	Code	Report	PPT
Ng Zheng Jue	Code modifications and constructions of network	Explanations of diffusion models	Slide Design + Overall checking
Ng Rui Qi	Network training and model evaluation of separate folders	Validation/verification	Validation
Tan Hong Guan	Evaluation using Inception score on generated images	Diffusion model mathematical derivations and methodology	Methodology
Ong Ming Jie	Network training and model evaluation of separate folders	Introduction, Background (Literature Review), conclusion	Intro + Background + Conclusion

5. Validation/Verification

5.1 **Result**

After training, Inception score (IS) and human visual evaluation is used to validate the reconstructed image. The reason for using Inception score is because it is generally used to evaluate generated images output by models. According to Tim Salimans, Inception scores are developed to remove subjective human evaluation of images (Salimans et al., 2016). The inception score aims to seek for two different properties of the data, which are Image Quality and Image Diversity. The higher the score, the better the quality of the generated images.

To calculate the inception score of a generative model G, the following equation is applied:

$$IS(G) = \exp\left(E_{\boldsymbol{x}}\left[KL\left(p(y|\boldsymbol{x})||p(y)\right)\right]\right)$$

$$= \exp\left(\int_{\boldsymbol{x}} p(y|\boldsymbol{x}) \left[\ln p(y|\boldsymbol{x}) - \ln \int_{\boldsymbol{x}} p(y|x) d\boldsymbol{x}\right] d\boldsymbol{x}\right)$$

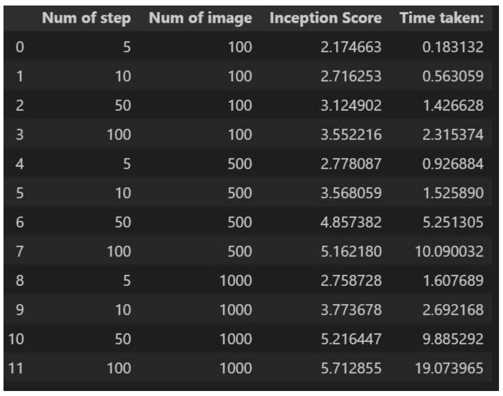
By Monte Carlo, we can rewrite the integral as summation form (Barratt et al., 2018). Therefore,

$$\boxed{IS(G) = \frac{1}{N}\sum_{i=1}^{N} p(y|\boldsymbol{x}_i) \left[\ln p(y|\boldsymbol{x}_i) - \ln \sum_{j=i}^{N} p(y|\boldsymbol{x}_i)\right]}$$

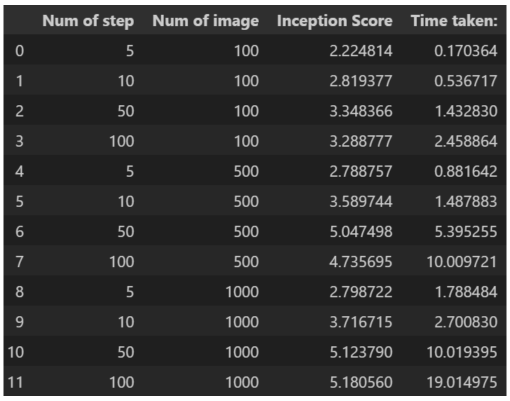
In practice, the equation by Monte Carlo is used because the integral of the probability distribution is hardly obtained usually.

To obtain the best result, the program will run multiple times with different seeds and compute its average. This is because different seeds will have a huge impact on the IS and time taken. For example, when running model in Folder_2(no label),

Seed (0)	Seed (1)



	Num of step	Num of image	Inception Score	Time taken:
0	5	100	2.174663	0.183132
1	10	100	2.716253	0.563059
2	50	100	3.124902	1.426628
3	100	100	3.552216	2.315374
4	5	500	2.778087	0.926884
5	10	500	3.568059	1.525890
6	50	500	4.857382	5.251305
7	100	500	5.162180	10.090032
8	5	1000	2.758728	1.607689
9	10	1000	3.773678	2.692168
10	50	1000	5.216447	9.885292
11	100	1000	5.712855	19.073965



	Num of step	Num of image	Inception Score	Time taken:
0	5	100	2.224814	0.170364
1	10	100	2.819377	0.536717
2	50	100	3.348366	1.432830
3	100	100	3.288777	2.458864
4	5	500	2.788757	0.881642
5	10	500	3.589744	1.487883
6	50	500	5.047498	5.395255
7	100	500	4.735695	10.009721
8	5	1000	2.798722	1.788484
9	10	1000	3.716715	2.700830
10	50	1000	5.123790	10.019395
11	100	1000	5.180560	19.014975

From both seeds, different results will be obtained. This result may lead to opposite conclusions being drawn. Thus, to ensure we obtain a more accurate result, our team chose to use an average Inception score using 5 different seeds. The table below show the result for each Folder:

Information / Folder	Number of steps	Number of Image	Inception Score (↑ better)	Time taken (↓ better)
1 (have label)	5	100	**2.174714**	1.326927
2 (No label)			2.174663	0.625429
3 (Ours – V1)			2.064425	0.624506
3 (Ours – V2)			2.156775	**0.594132**
1 (have label)	10		2.475825	1.752192
2 (No label)			**2.716253**	0.738410
3 (Ours – V1)			2.480563	0.801699
3 (Ours – V2)			2.539255	**0.717480**
1 (have label)	50		3.046099	3.742809
2 (No label)			**3.124902**	1.716626
3 (Ours – V1)			2.989483	1.783177
3 (Ours – V2)			3.075978	**1.688391**

Table 5.1: 100 image inception score

Information / Folder	Number of steps	Number of Image	Inception Score (↑ better)	Time taken (↓ better)
1 (have label)	5	500	2.352235	1.354344
2 (No label)			**2.608718**	1.175104
3 (Ours – V1)			2.443938	1.138037
3 (Ours – V2)			2.164406	**1.074470**
1 (have label)	10		2.860620	2.305094
2 (No label)			**3.459932**	1.544977
3 (Ours – V1)			2.913189	1.571032
3 (Ours – V2)			2.848407	**1.327559**
1 (have label)	50		3.338510	6.289643
2 (No label)			**4.666190**	**3.871089**
3 (Ours – V1)			4.245681	4.210478
3 (Ours – V2)			4.216586	4.071746

Table 5.2: 300 image inception score

Information / Folder	Number of steps	Number of Image	Inception Score (↑ better)	Time taken (↓ better)
1 (have label)	5	500	2.465517	1.688617
2 (No label)			**2.778087**	1.236471
3 (Ours – V1)			2.394432	1.248334
3 (Ours – V2)			2.156054	**0.924747**
1 (have label)	10		3.014049	2.982358
2 (No label)			**3.568059**	1.696497
3 (Ours – V1)			3.077052	1.871502
3 (Ours – V2)			3.050585	**1.501928**
1 (have label)	50		3.556916	8.873584
2 (No label)			**4.857382**	**5.506475**
3 (Ours – V1)			4.557778	6.286274
3 (Ours – V2)			4.417679	6.007638

Table 5.3: 500 image inception score

From the table above, we can know that:

- Most of the time, the model in Folder_2 generates the highest inception score. This is because inception score aims to calculate the diversity of the generated image. This means that the diversity of model_2 will be wider as the data has no label. In section 5.2, we will see that the image produced by model_2 will have many 'merged images' (e.g., car merging with a deer). This kind of diversity will lead to an increase in Inception Score.
- Our sampling of Version 1 generally has a better inception score than Version 2 but requires a longer time to generate an image.

- Our sampling of Version 2 has the shortest time taken to generate images except when the number of steps = 50. Thus, our model could only increase the time taken to generate images when the number of steps is small. However, this problem can be solved using samplers such as DPM-Solvers to reduce the number of sampling steps.

5.2 Example of reconstructed image of CIFAR-10

From Section 5.1, we can clearly observe that Model_2 has the highest Inception Score, but does it really mean better images? The figure below is the sample output image for the different models and different number of reconstructed steps.

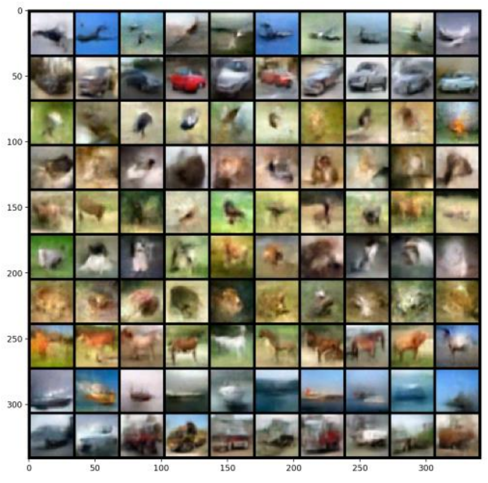
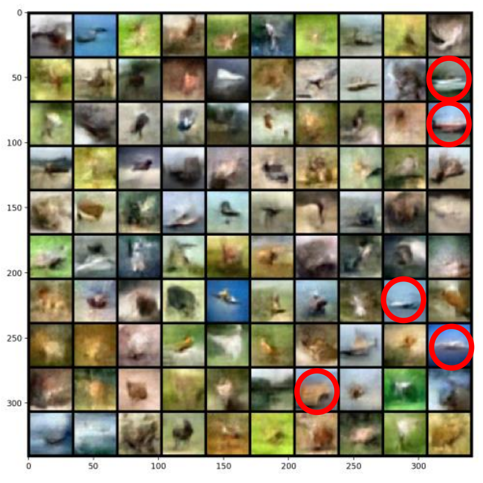
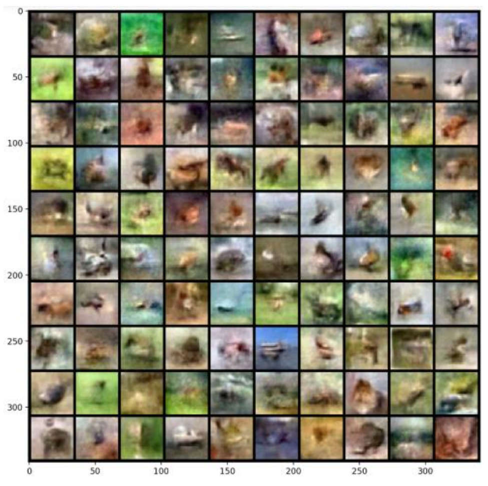
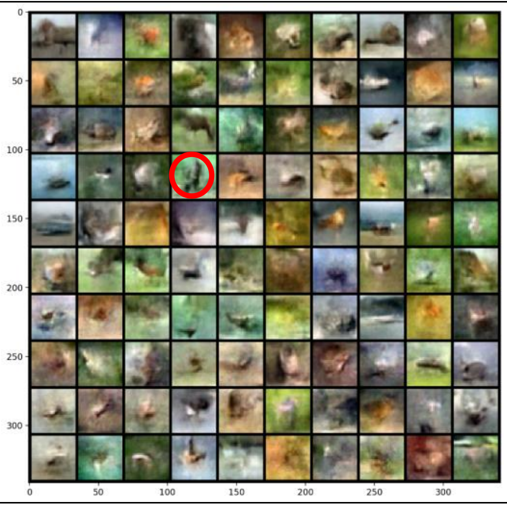
	
1 (Have Label)	2 (No label)
	
3 (Ours – V1)	3 (Ours – V2)

Table 5.4 Images for different models when **num of reconstructed step = 5**

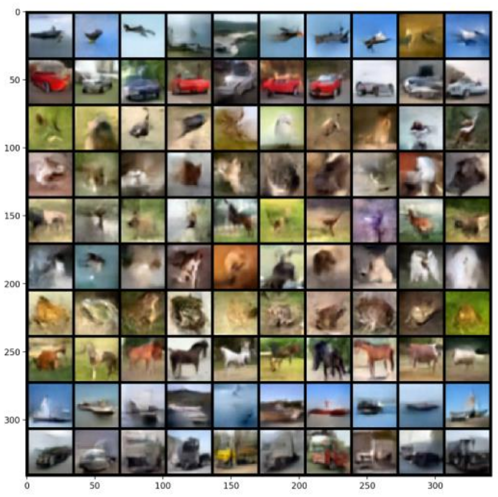
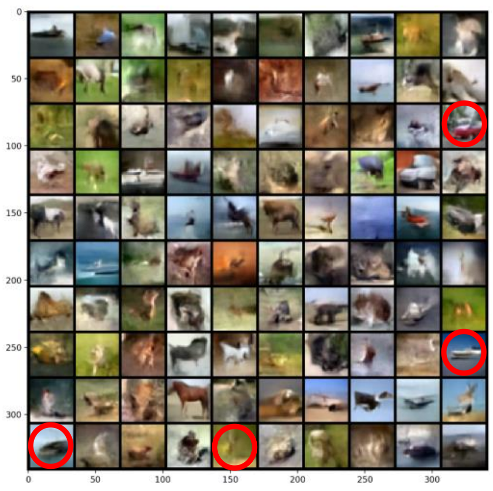
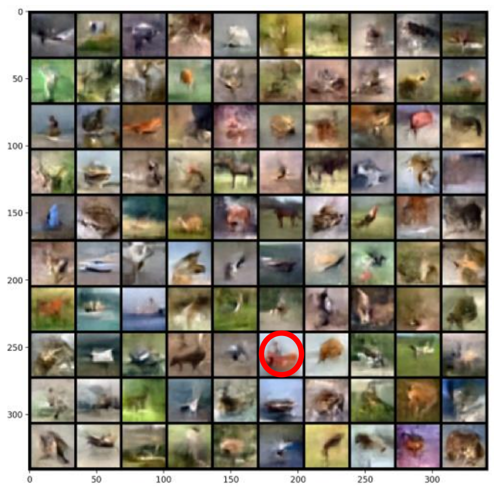
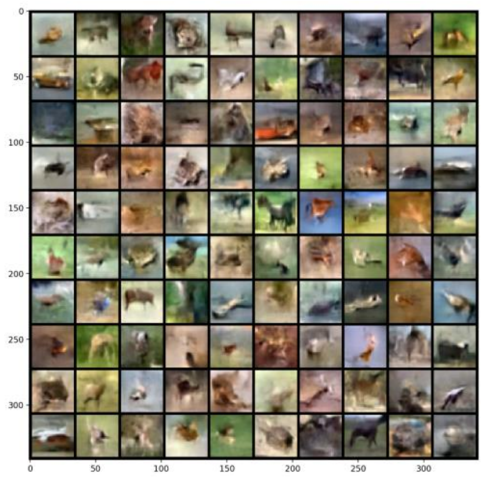
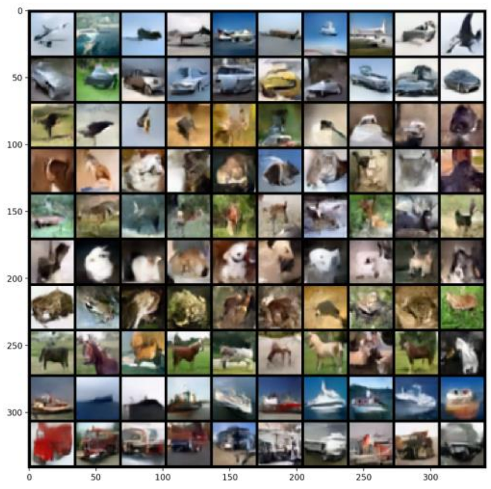
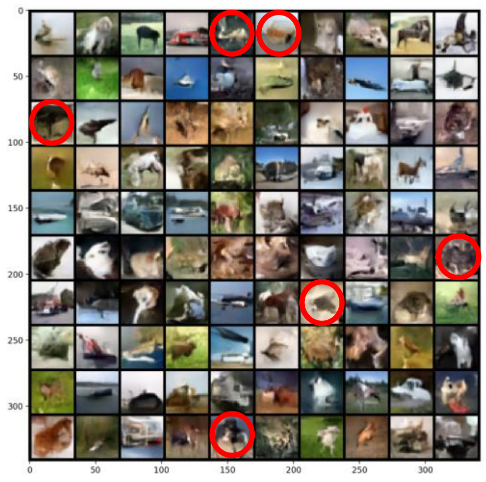
	
1 (Have Label)	2 (No label)
	
3 (Ours – V1)	3 (Ours – V2)

Table 5.5 Images for difference model when **num of reconstructed step = 10**

	
1 (Have Label)	2 (No label)

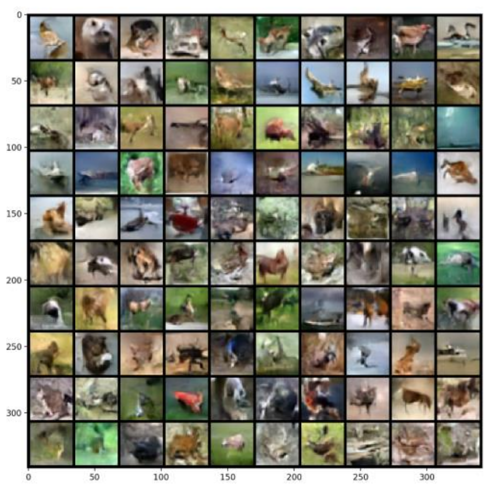
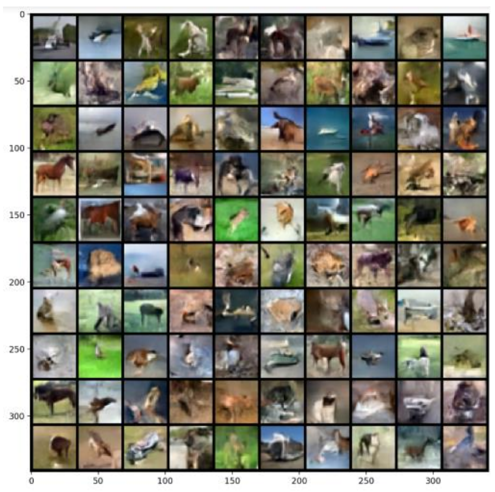
	
3 (Ours – V1)	3 (Ours – V2)

Table 5.6 Images for difference model when **num of reconstructed step = 50**

Interpretation of table

- The images generated by model_1 have the clearest images corresponding to a label.
- The images generated by model_2 have some weird images such as a combination of a horse and a car (This can be found from the red circle in the table above). It is possible that due to the lack of labelling in the model, merged images will be generated, thus increasing the inception score (realistic & diversity of the generated image).
- The images generated by model_3 have quality slightly worse than Folder_1 but slightly better than Folder_2. This is because the results of Folder_3 will mainly depend on pre-trained model on Folder_2.

5.3 Overall Conclusion

Overall, we can conclude that:

- When the model is trained with images and its corresponding label, it will produce clearer images corresponding to a label, but a longer time taken to generate images.

- When the model is trained with images only, followed by passing through additional layers that is trained using the label to guide the images toward a label, it will produce slightly better images than the model that is trained with images only. Consequently, this method can generate large number of images with shorter time if and only if the number of reconstructed steps is relatively small (<10).

5.4 Explanation of why our training methods work.

As we know that diffusion model is a score-based model, it is learning the score function $\nabla_x \log p(x)$. Ideally, by multiplying the score function with an orthogonal matrix, we can change the direction/gradient/score function towards a higher distribution vector space that contains an image with a specific label. Recall that all n x n orthogonal matrices can be decomposed to at most $O(d^2)$ rotation matrix and at most 1 elementary reflection matrix. Therefore, by multiplying the score function with the orthogonal matrix, the properties of the score function remains unchanged, and we can control the direction of the score function towards images with specific label.

Multiplication with the orthogonal matrix is done by the linear layer in the additional layer. As we know, passing an input to a fully connected layer is just performing a matrix multiplication to the input (FunCry, 2023). Ideally, if the model learns well, the matrix represented by the fully connected layer will be similar to orthogonal matrix. Additionally, we can also obtain the matrix that performs the rotation by getting the weight of each linear layer corresponding to the label.

5 Conclusion

In conclusion, we have managed to build a model that is able to generate images with quick speed. However, a major issue is that the generated samples are not of equal quality to other diffusion models. We perform a trade-off between the quality of the images and the speed at which the images are generated. Though we have managed to successfully produce the images at a faster speed, we aim to generate higher quality images in the future by modifying other aspects of the diffusion model's architecture and integrating an improved sampler into the diffusion model.

6 Reference

Andrew. (2023, June 10). *Stable diffusion samplers: A comprehensive guide*. Stable Diffusion Art. https://stable-diffusion-art.com/samplers/#Image_Convergence

Ayan Das. (2021, July 14). *Generative modelling with score functions*. Ayan Das. https://ayandas.me/blog-tut/2021/07/14/generative-model-score-function.html

FunCry. (2023, February 21). *Using matrix to represent fully connected layer and its gradient*. Medium. https://medium.com/@funcry/using-matrix-to-represent-fully-connected-layer-and-its-gradient-35604d99d138

Ho, J., Jain, A., & Abbeel, P. (2020, December 16). *Denoising Diffusion Probabilistic models*. arXiv.org. https://arxiv.org/abs/2006.11239

Lu, C., Zhou, Y., Bao, F., Chen, J., Li, C., & Zhu, J. (2022, October 13). *DPM-solver: A fast ode solver for diffusion probabilistic model sampling in around 10 steps*. arXiv.org. https://arxiv.org/abs/2206.00927

Nichol, A., & Dhariwal, P. (2021, February 18). *Improved denoising diffusion probabilistic models*. arXiv.org. https://arxiv.org/abs/2102.09672

Salimans, T., & Ho, J. (2022, June 7). *Progressive distillation for fast sampling of diffusion models*. arXiv.org. https://arxiv.org/abs/2202.00512

Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Radford, A., & Chen, X. (2016, June 10). *Improved techniques for training gans*. arXiv.org. https://arxiv.org/abs/1606.03498

Barratt, S., & Sharma, R. (2018b, June 21). *A note on the inception score*. arXiv.org. https://arxiv.org/abs/1801.01973

Luo, C. (2022, August 25). *Understanding diffusion models: A unified perspective*. arXiv.org. https://arxiv.org/abs/2208.11970

Soch, J. (2020, May 5). *Kullback-Leibler divergence for the Multivariate Normal Distribution*. The Book of Statistical Proofs. https://statproofbook.github.io/P/mvn-kl.html

7 Appendix

A. Diffusion model

i. Proof for formula (1.3)

Firstly, let $\alpha_t = 1 - \beta_t$ and let $\boldsymbol{\epsilon}_{t-1}, \boldsymbol{\epsilon}_{t-2}, \dots, \boldsymbol{\epsilon}_0 \sim \mathcal{N}_{i.i.d}(\mathbf{0}, \boldsymbol{I})$ be the set of standardization of posterior distributions $\boldsymbol{x}_t|\boldsymbol{x}_{t-1},\ \boldsymbol{x}_{t-1}|\boldsymbol{x}_{t-2} \dots,\ \boldsymbol{x}_1|\boldsymbol{x}_0$ respectively. Then the standardization of $\boldsymbol{x}_t|\boldsymbol{x}_{t-1} \sim \mathcal{N}(\sqrt{\alpha_t}\boldsymbol{x}_{t-1}, (1-\alpha_t)\boldsymbol{I})$ is given by

$$\boldsymbol{\epsilon}_{t-1} = \frac{\boldsymbol{x}_t - \sqrt{\alpha_t}\,\boldsymbol{x}_{t-1}}{\sqrt{1-\alpha_t}}$$

The term $1 - \alpha_t$ in the denominator is simply the variance of the distribution because it is in the diagonal of the covariance matrix $(1-\alpha_t)\boldsymbol{I}$.

Rearranging the equation above will give us

$$\begin{aligned}
\boldsymbol{x}_t &= \sqrt{\alpha_t}\,\boldsymbol{x}_{t-1} + \sqrt{1-\alpha_t}\,\boldsymbol{\epsilon}_{t-1} \\
&= \sqrt{\alpha_t}\left(\sqrt{\alpha_{t-1}}\,\boldsymbol{x}_{t-2} + \sqrt{1-\alpha_{t-1}}\,\boldsymbol{\epsilon}_{t-2}\right) + \sqrt{1-\alpha_t}\,\boldsymbol{\epsilon}_{t-1} \\
&= \sqrt{\alpha_t\alpha_{t-1}}\,\boldsymbol{x}_{t-2} + \sqrt{\alpha_t(1-\alpha_{t-1})}\,\boldsymbol{\epsilon}_{t-2} + \sqrt{1-\alpha_t}\,\boldsymbol{\epsilon}_{t-1}
\end{aligned}$$

Notice that $\sqrt{\alpha_t(1-\alpha_{t-1})}\,\boldsymbol{\epsilon}_{t-2} + \sqrt{1-\alpha_t}\,\boldsymbol{\epsilon}_{t-1}$ is the addition of a pair of ***i.i.d*** standard Gaussian distribution. Since $\boldsymbol{\epsilon}_{t-2}, \boldsymbol{\epsilon}_{t-1} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$, then $\boldsymbol{x}_t$ can be written in terms of $\mathcal{N}(\mathbf{0}, \boldsymbol{I})$ where $\mathcal{N}$ is a standard normal density function. Therefore, we can deduce that

$$\begin{aligned}
\boldsymbol{x}_t &= \sqrt{\alpha_t\alpha_{t-1}}\,\boldsymbol{x}_{t-2} + \mathcal{N}(\mathbf{0}, \alpha_t(1-\alpha_{t-1})\boldsymbol{I}) + \mathcal{N}(\mathbf{0}, (1-\alpha_t)\boldsymbol{I}) \\
&= \sqrt{\alpha_t\alpha_{t-1}}\,\boldsymbol{x}_{t-2} + \mathcal{N}(\mathbf{0}, (\alpha_t - \alpha_t\alpha_{t-1} + 1 - \alpha_t)\boldsymbol{I}) \\
&= \sqrt{\alpha_t\alpha_{t-1}}\,\boldsymbol{x}_{t-2} + \mathcal{N}(\mathbf{0}, (1-\alpha_t\alpha_{t-1})\boldsymbol{I})
\end{aligned}$$

$$= \sqrt{\alpha_t \alpha_{t-1}}\, \boldsymbol{x}_{t-2} + \sqrt{(1-\alpha_t\alpha_{t-1})}\mathcal{N}(\boldsymbol{0},\boldsymbol{I})$$

If we continue to expand until $\boldsymbol{x}_1$, we will eventually get the form

$$\boldsymbol{x}_t = \sqrt{\alpha_t\alpha_{t-1}\alpha_{t-2}\ldots\alpha_1}\boldsymbol{x}_0 + \sqrt{(1-\alpha_t\alpha_{t-1}\alpha_{t-2}\ldots\alpha_1)}\cdot\boldsymbol{\epsilon}_t$$

$$= \sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \left(\sqrt{1-\bar{\alpha}_t}\right)\cdot\boldsymbol{\epsilon}_t$$

where $\bar{\alpha}_t = \prod_{r=0}^{t-1}\alpha_{t-r}$ and $\epsilon = \mathcal{N}(\boldsymbol{0},\boldsymbol{I})$ which is a standard normal distribution.

∴ Finally, we can conclude that

$$\boldsymbol{x}_t|\boldsymbol{x}_0 \sim \mathcal{N}\left(\sqrt{\bar{\alpha}_t}\boldsymbol{x}_0, (1-\bar{\alpha}_t)\boldsymbol{I}\right)$$

and the closed form solution of $q(\boldsymbol{x}_t|\boldsymbol{x}_0)$ is then given by

$$\boxed{q(\boldsymbol{x}_t|\boldsymbol{x}_0) = \mathcal{N}\left(\boldsymbol{x}_t; \sqrt{\bar{\alpha}_t}\boldsymbol{x}_0, (1-\bar{\alpha}_t)\boldsymbol{I}\right)}$$

ii. Proof for formula (1.6)

Before moving on, note that the multivariate Gaussian density function of a $d \times 1$ random vector $\boldsymbol{x}$ is defined by

$$\mathcal{N}(\boldsymbol{x};\boldsymbol{\mu},\boldsymbol{\Sigma}) \overset{\text{def}}{=} \frac{1}{(2\pi)^{\frac{d}{2}}\det(\boldsymbol{\Sigma})^{\frac{1}{2}}} e^{-\frac{1}{2}\Delta_{\boldsymbol{x}}^2}$$

(1.6.1)

where $\Delta_{\boldsymbol{x}}^2 = (\boldsymbol{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\boldsymbol{x}-\boldsymbol{\mu})$ is scalar.

From equation $q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}) = \mathcal{N}\left(\boldsymbol{x}_t; \boldsymbol{\mu} = \sqrt{1-\beta_t}\boldsymbol{x}_{t-1}, \boldsymbol{\Sigma} = \beta_t\boldsymbol{I}\right)$, $\Delta_{\boldsymbol{x}_t|\boldsymbol{x}_{t-1}}^2$ is given by

$$\Delta_{\boldsymbol{x}_t|\boldsymbol{x}_{t-1}}^2 = \left(\boldsymbol{x}_t - \sqrt{\alpha_t}\boldsymbol{x}_{t-1}\right)^T(\beta_t\boldsymbol{I})^{-1}\left(\boldsymbol{x}_t - \sqrt{\alpha_t}\boldsymbol{x}_{t-1}\right)$$

Recall that $\alpha_t = 1-\beta_t$ and $(\beta_t\boldsymbol{I})^{-1}$ can be simplified to $\frac{1}{\beta_t}\boldsymbol{I}$. Then, we deduce that

$$\Delta_{\boldsymbol{x}_t|\boldsymbol{x}_{t-1}}^2 = \frac{\left|\left|\boldsymbol{x}_t - \sqrt{\alpha_t}\boldsymbol{x}_{t-1}\right|\right|^2}{\beta_t}$$

(1.6.2)

Similarly, from equation $q(\boldsymbol{x}_t|\boldsymbol{x}_0) = \mathcal{N}\left(\boldsymbol{x}_t; \sqrt{\bar{\alpha}_t}\boldsymbol{x}_0, (1-\bar{\alpha}_t)\boldsymbol{I}\right)$, $\Delta^2_{\boldsymbol{x}_t|\boldsymbol{x}_0}$ is given by

$$\Delta^2_{\boldsymbol{x}_t|\boldsymbol{x}_0} = \frac{\left|\left|\boldsymbol{x}_t - \sqrt{\bar{\alpha}_t}\boldsymbol{x}_0\right|\right|^2}{1-\bar{\alpha}_t} \tag{1.6.3}$$

Then by Bayes' Theorem, $q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \boldsymbol{x}_0)$ can be written as

$$\begin{aligned} q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \boldsymbol{x}_0) &= q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1}, \boldsymbol{x}_0)\frac{q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_0)}{q(\boldsymbol{x}_t|\boldsymbol{x}_0)} \\ &= \frac{1}{(2\pi)^{\frac{d}{2}}\sqrt{\frac{\beta_t(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}}} e^{\left(-\frac{1}{2}\Delta^2_{\boldsymbol{x}_{t-1}|\boldsymbol{x}_t,\boldsymbol{x}_0}\right)} \end{aligned} \tag{1.6.4}$$

where $\Delta^2_{\boldsymbol{x}_{t-1}|\boldsymbol{x}_t,\boldsymbol{x}_0} = \Delta^2_{\boldsymbol{x}_t|\boldsymbol{x}_{t-1}} + \Delta^2_{\boldsymbol{x}_{t-1}|\boldsymbol{x}_0} - \Delta^2_{\boldsymbol{x}_t|\boldsymbol{x}_0}$. By the definition of (1.6.1), the estimated covariance matrix, $\tilde{\beta}_t\boldsymbol{I}$ and mean vector, $\tilde{\boldsymbol{\mu}}(\boldsymbol{x}_t, \boldsymbol{x}_0)$ can be found in the coefficient and $\Delta^2_{\boldsymbol{x}_{t-1}|\boldsymbol{x}_t,\boldsymbol{x}_0}$ from the density function in (1.6.4) respectively. Therefore, by comparing (1.6.1) and (1.6.4) we can have

$$\tilde{\beta}_{\mathrm{t}} = \det(\boldsymbol{\Sigma}) = \left(\sqrt{\frac{\beta_t(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}}\right)^2 = \frac{\beta_t(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}$$

As a result, $\tilde{\beta}_{\mathrm{t}}\boldsymbol{I} = \frac{\beta_t(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\boldsymbol{I}$. For $\tilde{\boldsymbol{\mu}}(\boldsymbol{x}_t, \boldsymbol{x}_0)$ on the other hand, we should consider the term $\Delta^2_{\boldsymbol{x}_{t-1}|\boldsymbol{x}_t,\boldsymbol{x}_0}$. Hence, by (1.6.2) and (1.6.3), $\Delta^2_{\boldsymbol{x}_{t-1}|\boldsymbol{x}_t,\boldsymbol{x}_0}$ is given by

$$\begin{aligned} \Delta^2_{\boldsymbol{x}_{t-1}|\boldsymbol{x}_t,\boldsymbol{x}_0} &= \Delta^2_{\boldsymbol{x}_t|\boldsymbol{x}_{t-1}} + \Delta^2_{\boldsymbol{x}_{t-1}|\boldsymbol{x}_0} - \Delta^2_{\boldsymbol{x}_t|\boldsymbol{x}_0} \\ &= \frac{\left|\left|\boldsymbol{x}_t - \sqrt{\alpha_t}\boldsymbol{x}_{t-1}\right|\right|^2}{\beta_t} + \frac{\left|\left|\boldsymbol{x}_{t-1} - \sqrt{\bar{\alpha}_{t-1}}\boldsymbol{x}_0\right|\right|^2}{1-\bar{\alpha}_{t-1}} - \frac{\left|\left|\boldsymbol{x}_t - \sqrt{\bar{\alpha}_t}\boldsymbol{x}_0\right|\right|^2}{1-\bar{\alpha}_t} \end{aligned}$$

$$= \frac{||\boldsymbol{x}_t||^2 - 2\sqrt{\alpha_t}\boldsymbol{x}_t \cdot \boldsymbol{x}_{t-1} + \alpha_t||\boldsymbol{x}_{t-1}||^2}{\beta_t}$$

$$+ \frac{||\boldsymbol{x}_{t-1}||^2 - 2\sqrt{\bar{\alpha}_{t-1}}\boldsymbol{x}_0 \cdot \boldsymbol{x}_{t-1} + \bar{\alpha}_{t-1}||\boldsymbol{x}_0||^2}{1-\bar{\alpha}_{t-1}} - \frac{||\boldsymbol{x}_t - \sqrt{\bar{\alpha}_t}\boldsymbol{x}_0||^2}{1-\bar{\alpha}_t}$$

Grouping the coefficients of $\boldsymbol{x}_{t-1}$ will give us

$$\Delta^2_{\boldsymbol{x}_{t-1}|\boldsymbol{x}_t,\boldsymbol{x}_0} = \left(\frac{\alpha_t}{\beta_t} + \frac{1}{1-\bar{\alpha}_{t-1}}\right)||\boldsymbol{x}_{t-1}||^2 - \left(\frac{2\sqrt{\alpha_t}}{\beta_t}\boldsymbol{x}_t + \frac{2\sqrt{\bar{\alpha}_{t-1}}}{1-\bar{\alpha}_{t-1}}\boldsymbol{x}_0\right) \cdot \boldsymbol{x}_{t-1} + \xi(\boldsymbol{x}_t, \boldsymbol{x}_0)$$

(1.6.5)

The remaining terms that are not coefficients of $\boldsymbol{x}_{t-1}$ are represented as $\xi(\boldsymbol{x}_t, \boldsymbol{x}_0)$. At this point, it is hard to tell which part of $\Delta^2_{\boldsymbol{x}_{t-1}|\boldsymbol{x}_t,\boldsymbol{x}_0}$ belongs to the mean vector $\tilde{\boldsymbol{\mu}}(\boldsymbol{x}_t, \boldsymbol{x}_0)$. However, from the quadratic form of $\Delta^2_{\boldsymbol{x}_{t-1}|\boldsymbol{x}_t,\boldsymbol{x}_0}$, we can tell which particular term of (1.6.5) associates with the mean vector as shown below. For simplicity, $\tilde{\boldsymbol{\mu}}(\boldsymbol{x}_t, \boldsymbol{x}_0)$ will be replaced by $\tilde{\boldsymbol{\mu}}$.

$$\begin{aligned}
\Delta^2_{\boldsymbol{x}_{t-1}|\boldsymbol{x}_t,\boldsymbol{x}_0} &= (\boldsymbol{x}_{t-1} - \tilde{\boldsymbol{\mu}})^T\left(\tilde{\beta}_t\boldsymbol{I}\right)^{-1}(\boldsymbol{x}_{t-1} - \tilde{\boldsymbol{\mu}}) \\
&= \frac{1}{\tilde{\beta}_t}(\boldsymbol{x}_{t-1} - \tilde{\boldsymbol{\mu}})^T(\boldsymbol{x}_{t-1} - \tilde{\boldsymbol{\mu}}) \\
&= \frac{1}{\tilde{\beta}_t}\left(||\boldsymbol{x}_{t-1}||^2 - 2\tilde{\boldsymbol{\mu}} \cdot \boldsymbol{x}_{t-1} + ||\tilde{\boldsymbol{\mu}}||^2\right) \\
&= \frac{||\boldsymbol{x}_{t-1}||^2}{\tilde{\beta}_t} - \frac{2\tilde{\boldsymbol{\mu}} \cdot \boldsymbol{x}_{t-1}}{\tilde{\beta}_t} + \frac{||\tilde{\boldsymbol{\mu}}||^2}{\tilde{\beta}_t}
\end{aligned}$$

(1.6.6)

From the final result shown in (1.6.6), it shows that we can evaluate $\tilde{\boldsymbol{\mu}}$ by equating the coefficients of $\boldsymbol{x}_{t-1}$ in (1.6.5) to $\frac{-2\tilde{\boldsymbol{\mu}}}{\tilde{\beta}_t}$. i.e.,

$$\frac{-2\tilde{\boldsymbol{\mu}}}{\tilde{\beta}_t} = -\left(\frac{2\sqrt{\alpha_t}}{\beta_t}\boldsymbol{x}_t + \frac{2\sqrt{\bar{\alpha}_{t-1}}}{1-\bar{\alpha}_{t-1}}\boldsymbol{x}_0\right)$$

$$\tilde{\boldsymbol{\mu}} = \tilde{\beta}_t\left(\frac{\sqrt{\alpha_t}}{\beta_t}\boldsymbol{x}_t + \frac{\sqrt{\bar{\alpha}_{t-1}}}{1-\bar{\alpha}_{t-1}}\boldsymbol{x}_0\right)$$

Since $\tilde{\beta}_t = \frac{\beta_t(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}$ and from $\sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \left(\sqrt{1-\bar{\alpha}_t}\right)\cdot\boldsymbol{\epsilon}_t$, $\boldsymbol{x}_0 = \frac{\boldsymbol{x}_t - \left(\sqrt{1-\bar{\alpha}_t}\right)\cdot\boldsymbol{\epsilon}_t}{\sqrt{\bar{\alpha}_t}}$, we deduce that

$$\begin{aligned}
\tilde{\boldsymbol{\mu}} &= \left(\frac{\beta_t(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\right)\left(\frac{\sqrt{\alpha_t}}{\beta_t}\boldsymbol{x}_t + \frac{\sqrt{\bar{\alpha}_{t-1}}}{1-\bar{\alpha}_{t-1}}\left(\frac{\boldsymbol{x}_t - \left(\sqrt{1-\bar{\alpha}_t}\right)\cdot\boldsymbol{\epsilon}_t}{\sqrt{\bar{\alpha}_t}}\right)\right) \\
&= \left(\frac{\beta_t(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\right)\left(\frac{\sqrt{\alpha_t}}{\beta_t}\boldsymbol{x}_t + \frac{\sqrt{\bar{\alpha}_{t-1}}}{(1-\bar{\alpha}_{t-1})\sqrt{\bar{\alpha}_t}}\boldsymbol{x}_t + \frac{\sqrt{\bar{\alpha}_{t-1}}\left(\sqrt{1-\bar{\alpha}_t}\right)}{(1-\bar{\alpha}_{t-1})\sqrt{\bar{\alpha}_t}}\boldsymbol{\epsilon}_t\right) \\
&= \left(\frac{(1-\bar{\alpha}_{t-1})\sqrt{\alpha_t}}{1-\bar{\alpha}_t} + \frac{\beta_t\sqrt{\bar{\alpha}_{t-1}}}{(1-\bar{\alpha}_t)\sqrt{\bar{\alpha}_t}}\right)\boldsymbol{x}_t + \frac{\beta_t\left(\sqrt{1-\bar{\alpha}_t}\right)\sqrt{\bar{\alpha}_{t-1}}}{(1-\bar{\alpha}_t)\sqrt{\bar{\alpha}_t}}\boldsymbol{\epsilon}_t \\
&= \left(\frac{(1-\bar{\alpha}_{t-1})\sqrt{\alpha_t}}{1-\bar{\alpha}_t} + \frac{\beta_t\sqrt{\bar{\alpha}_{t-1}}}{(1-\bar{\alpha}_t)\sqrt{\alpha_t\bar{\alpha}_{t-1}}}\right)\boldsymbol{x}_t + \frac{\beta_t\sqrt{\bar{\alpha}_{t-1}}}{\sqrt{1-\bar{\alpha}_t}\cdot\sqrt{\alpha_t\bar{\alpha}_{t-1}}}\boldsymbol{\epsilon}_t \\
&= \left(\frac{(1-\bar{\alpha}_{t-1})\sqrt{\alpha_t}}{1-\bar{\alpha}_t} + \frac{\beta_t}{(1-\bar{\alpha}_t)\sqrt{\alpha_t}}\right)\boldsymbol{x}_t + \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}\cdot\sqrt{\alpha_t}}\boldsymbol{\epsilon}_t \\
&= \frac{(1-\bar{\alpha}_{t-1})\alpha_t + (1-\alpha_t)}{(1-\bar{\alpha}_t)\sqrt{\alpha_t}}\boldsymbol{x}_t + \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}\cdot\sqrt{\alpha_t}}\boldsymbol{\epsilon}_t \\
&= \frac{1}{\sqrt{\alpha_t}}\left(\frac{\alpha_t - \bar{\alpha}_t + 1 - \alpha_t}{1-\bar{\alpha}_t}\boldsymbol{x}_t + \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_t\right) \\
&= \frac{1}{\sqrt{\alpha_t}}\left(\boldsymbol{x}_t + \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_t\right)
\end{aligned}$$

$\therefore$ The mean vector $\tilde{\boldsymbol{\mu}}(\boldsymbol{x}_t, \boldsymbol{x}_0)$ and covariance matrix $\tilde{\beta}_t\boldsymbol{I}$ are given by

$$\tilde{\boldsymbol{\mu}}(\boldsymbol{x}_t, \boldsymbol{x}_0) = \frac{1}{\sqrt{\alpha_t}}\left(\boldsymbol{x}_t + \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_t\right), \qquad \tilde{\beta}_t\boldsymbol{I} = \frac{\beta_t(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\boldsymbol{I}$$

where $\boldsymbol{\epsilon}_t = \frac{\boldsymbol{x}_0\sqrt{\bar{\alpha}_t} - \boldsymbol{x}_t}{\sqrt{1-\bar{\alpha}_t}}$ and finally, we can conclude that

$$\boxed{q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t,\boldsymbol{x}_0) = \mathcal{N}\left(\boldsymbol{x}_{t-1}; \frac{1}{\sqrt{\alpha_t}}\left(\boldsymbol{x}_t + \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_t\right), \frac{\beta_t(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\boldsymbol{I}\right)}$$

iii. Proof for formula (1.7)

In this section, we will formulate the loss function using the properties we have obtained so far. To formulate the loss function, we will first need to find the evidence lower bound (ELBO) of the estimating model

$p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t) \sim \mathcal{N}\big(\boldsymbol{x}_{t-1}; \boldsymbol{\mu}_\theta(\boldsymbol{x}_t,t), \boldsymbol{\Sigma}_\theta(\boldsymbol{x}_t,t)\big)$ and optimize it. The ELBO of the distribution can be found in the evidence (log likelihood) of the distribution $p_\theta(\boldsymbol{x}_0)$ (Luo, 2022) which is given by

$$evidence = ELBO + KL\ divergence$$

$$\begin{aligned}
\ln p_\theta(\boldsymbol{x}_0) &= \ln\big(p_\theta(\boldsymbol{x}_0)\big)\overbrace{\int q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)d\boldsymbol{x}_{1:T}}^{1} \\
&= \int q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)\ln p_\theta(\boldsymbol{x}_0)\,d\boldsymbol{x}_{1:T} \\
&= \mathbb{E}_{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)}[\ln p_\theta(\boldsymbol{x}_0)] \\
&= \mathbb{E}_{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)}\left[\ln\frac{p_\theta(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)p_\theta(\boldsymbol{x}_0)}{p_\theta(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)}\right] \\
&= \mathbb{E}_{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)}\left[\ln\frac{p_\theta(\boldsymbol{x}_{0:T})}{p_\theta(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)}\right] \\
&= \mathbb{E}_{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)}\left[\ln\frac{p_\theta(\boldsymbol{x}_{0:T})q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)}{p_\theta(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)}\right] \\
&= \mathbb{E}_{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)}\left[\ln\frac{p_\theta(\boldsymbol{x}_{0:T})}{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)}\right] + \mathbb{E}_{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)}\left[\ln\frac{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)}{p_\theta(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)}\right] \\
&= \mathbb{E}_{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)}\left[\ln\frac{p_\theta(\boldsymbol{x}_{0:T})}{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)}\right] + D_{KL}\big(q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)\parallel p_\theta(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)\big)
\end{aligned}$$

Since $D_{KL}\big(q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)\parallel p_\theta(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)\big) \geq 0$, we can conclude that

$$ELBO_{p_\theta(x_0)} = \mathbb{E}_{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)}\left[\ln \frac{p_\theta(\boldsymbol{x}_{0:T})}{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)}\right]$$

where $\ln p_\theta(\boldsymbol{x}_0) \geq ELBO_{p_\theta(x_0)}$. We also label $ELBO_{p_\theta(x_0)}$ as the variational lower bound. However, in practice, the training involves minimization of the variational upper bound (L_{vlb}) on the negative log likelihood, i.e., $L_{vlb} = -ELBO_{p_\theta(x_0)}$ because we want to find the reverse Markov Transitions that maximize the likelihood of the training data. We also want L_{vlb} in terms of the Kullback-Leibler (KL) Divergences and entropy which is given in the form

$$\boxed{L_{vlb} = \underbrace{D_{KL}\big(q(\boldsymbol{x}_T|\boldsymbol{x}_0) \parallel p_\theta(\boldsymbol{x}_T)\big)}_{L_T} + \sum_{t=2}^{T} \underbrace{D_{KL}\big(q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \boldsymbol{x}_0) \parallel p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)\big)}_{L_{t-1}} \underbrace{-\mathbb{E}_q[\ln p_\theta(\boldsymbol{x}_0|\boldsymbol{x}_1)]}_{L_0}}$$

Let L_T, L_{t-1} and L_0 to denote the separate terms as shown above, then we can rewrite L_{vlb} as

$$L_{vlb} = L_T + L_{T-1} + \cdots + L_{t-1} + \cdots + L_1 + L_0$$

where that $2 \leq t \leq T$.

Proof:

$$L_{vlb} = \mathbb{E}_q\left[\ln \frac{q(\boldsymbol{x}_{1:T}|\boldsymbol{x}_0)}{p_\theta(\boldsymbol{x}_{0:T})}\right] = \mathbb{E}_q\left[\ln\left(\frac{1}{p_\theta(\boldsymbol{x}_T)} \prod_{t=1}^{T} \frac{q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})}{p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)}\right)\right]$$

$$= \mathbb{E}_q\left[-\ln p_\theta(\boldsymbol{x}_T) + \sum_{t=1}^{T} \ln \frac{q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})}{p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)}\right]$$

$$= \mathbb{E}_q\left[-\ln p_\theta(\boldsymbol{x}_T) + \sum_{t=2}^{T} \ln \frac{q(\boldsymbol{x}_t|\boldsymbol{x}_{t-1})}{p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)} + \ln \frac{q(\boldsymbol{x}_1|\boldsymbol{x}_0)}{p_\theta(\boldsymbol{x}_0|\boldsymbol{x}_1)}\right]$$

$$= \mathbb{E}_q\left[-\ln p_\theta(\boldsymbol{x}_T) + \sum_{t=2}^{T} \ln\left(\frac{q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \boldsymbol{x}_0)}{p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)} \cdot \frac{q(\boldsymbol{x}_t|\boldsymbol{x}_0)}{q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_0)}\right) + \ln \frac{q(\boldsymbol{x}_1|\boldsymbol{x}_0)}{p_\theta(\boldsymbol{x}_0|\boldsymbol{x}_1)}\right]$$

$$= \mathbb{E}_q\left[-\ln p_\theta(\boldsymbol{x}_T) + \sum_{t=2}^{T} \ln\left(\frac{q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t, \boldsymbol{x}_0)}{p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)}\right) + \sum_{t=2}^{T} \ln \frac{q(\boldsymbol{x}_t|\boldsymbol{x}_0)}{q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_0)} + \ln \frac{q(\boldsymbol{x}_1|\boldsymbol{x}_0)}{p_\theta(\boldsymbol{x}_0|\boldsymbol{x}_1)}\right]$$

$$= \mathbb{E}_q\left[-\ln p_\theta(\boldsymbol{x}_T) + \sum_{t=2}^{T} \ln\left(\frac{q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t,\boldsymbol{x}_0)}{p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)}\right) + \ln\left(\prod_{t=2}^{T}\frac{q(\boldsymbol{x}_t|\boldsymbol{x}_0)}{q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_0)}\right)\right.$$
$$\left. + \ln\frac{q(\boldsymbol{x}_1|\boldsymbol{x}_0)}{p_\theta(\boldsymbol{x}_0|\boldsymbol{x}_1)}\right]$$

$$= \mathbb{E}_q\left[-\ln p_\theta(\boldsymbol{x}_T) + \sum_{t=2}^{T} \ln\left(\frac{q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t,\boldsymbol{x}_0)}{p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)}\right)\right.$$
$$\left. + \ln\left(\frac{q(\boldsymbol{x}_T|\boldsymbol{x}_0)q(\boldsymbol{x}_{T-1}|\boldsymbol{x}_0)\ldots q(\boldsymbol{x}_2|\boldsymbol{x}_0)}{q(\boldsymbol{x}_{T-1}|\boldsymbol{x}_0)\ldots q(\boldsymbol{x}_2|\boldsymbol{x}_0)q(\boldsymbol{x}_1|\boldsymbol{x}_0)}\right)\right] + \ln\frac{q(\boldsymbol{x}_1|\boldsymbol{x}_0)}{p_\theta(\boldsymbol{x}_0|\boldsymbol{x}_1)}$$

$$= \mathbb{E}_q\left[-\ln p_\theta(\boldsymbol{x}_T) + \sum_{t=2}^{T} \ln\left(\frac{q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t,\boldsymbol{x}_0)}{p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)}\right) + \ln\frac{q(\boldsymbol{x}_T|\boldsymbol{x}_0)}{q(\boldsymbol{x}_1|\boldsymbol{x}_0)} + \ln\frac{q(\boldsymbol{x}_1|\boldsymbol{x}_0)}{p_\theta(\boldsymbol{x}_0|\boldsymbol{x}_1)}\right]$$

$$= \mathbb{E}_q\left[\ln\frac{q(\boldsymbol{x}_T|\boldsymbol{x}_0)}{p_\theta(\boldsymbol{x}_T)} + \sum_{t=2}^{T} \ln\left(\frac{q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t,\boldsymbol{x}_0)}{p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)}\right) - \ln p_\theta(\boldsymbol{x}_0|\boldsymbol{x}_1)\right]$$

$$= D_{KL}\big(q(\boldsymbol{x}_T|\boldsymbol{x}_0) \parallel p_\theta(\boldsymbol{x}_T)\big) + \sum_{t=2}^{T} D_{KL}\big(q(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t,\boldsymbol{x}_0) \parallel p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t)\big)$$
$$- \mathbb{E}_q[\ln p_\theta(\boldsymbol{x}_0|\boldsymbol{x}_1)]$$

From the result shown above, each KL divergences term in L_{vlb} measures the difference in information between two Gaussian distribution. Since $\boldsymbol{x}_T$ is a Gaussian noise, L_T turns out to be a constant because q has no learnable parameters.

Recall that we want to learn a neural network to approximate

$$p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t) \sim \mathcal{N}\big(\boldsymbol{x}_{t-1}; \boldsymbol{\mu}_\theta(\boldsymbol{x}_t,t), \boldsymbol{\Sigma}_\theta(\boldsymbol{x}_t,t)\big)$$

in reverse diffusion process to train $\boldsymbol{\mu}_\theta(\boldsymbol{x}_t,t)$ to predict $\tilde{\boldsymbol{\mu}}(\boldsymbol{x}_t,\boldsymbol{x}_0) = \frac{1}{\sqrt{\alpha_t}}\Big(\boldsymbol{x}_t + \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_t\Big)$. Because $\boldsymbol{x}_t$ is available as input during training time, we can re-parameterize the Gaussian noise term instead to make it predict $\boldsymbol{\epsilon}_t$ from the input $\boldsymbol{x}_t$ at time step t. In other word, we assume that

$$p_\theta(\boldsymbol{x}_{t-1}|\boldsymbol{x}_t) \sim \mathcal{N}\left(\boldsymbol{x}_{t-1}; \boldsymbol{\mu}_\theta(\boldsymbol{x}_t,t) = \frac{1}{\sqrt{\alpha_t}}\left(\boldsymbol{x}_t + \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t,t)\right), \boldsymbol{\Sigma}_\theta(\boldsymbol{x}_t,t) = \sigma_\theta^2\boldsymbol{I}\right)$$

Note that the assumption of $\boldsymbol{\Sigma}_\theta(\boldsymbol{x}_t,t) = \sigma_\theta^2\boldsymbol{I}$ will be helpful when we want to find its inverse. Let $\boldsymbol{z}$ be a $n \times 1$ random vector with the two multivariate Gaussian distribution $\mathcal{N}_1: \boldsymbol{z} \sim \mathcal{N}(\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1)$ and $\mathcal{N}_2: \boldsymbol{z} \sim \mathcal{N}(\boldsymbol{\mu}_2, \boldsymbol{\Sigma}_2)$. Then, the Kullback-Leibler divergence of $\mathcal{N}_1$ and $\mathcal{N}_2$ (Soch ,2020) is given by

$$D_{KL}(\mathcal{N}_1 \parallel \mathcal{N}_2) = \frac{1}{2}\left((\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)\boldsymbol{\Sigma}_2^{-1}(\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1) + tr(\boldsymbol{\Sigma}_2^{-1}\boldsymbol{\Sigma}_1) - \ln\frac{\det(\boldsymbol{\Sigma}_2)}{\det(\boldsymbol{\Sigma}_1)} - n \right)$$

However, in our case, we only need the term with mean vector (Ho et al, 2020), i.e., $\frac{1}{2}(\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)\boldsymbol{\Sigma}_2^{-1}(\boldsymbol{\mu}_2 - \boldsymbol{\mu}_1)$ because our objective is to train $\boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t)$ while other terms are just constant. Therefore, we will use this property to parameterize L_{t-1}. We rewrite L_{t-1} as

$$\begin{aligned}
\mathrm{L}^{(t)} &= \frac{1}{2}\left(\boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t) - \tilde{\boldsymbol{\mu}}(\boldsymbol{x}_t, \boldsymbol{x}_0)\right)\boldsymbol{\Sigma}_\theta^{-1}(\boldsymbol{x}_t, t)\left(\boldsymbol{\mu}_\theta(\boldsymbol{x}_t, t) - \tilde{\boldsymbol{\mu}}(\boldsymbol{x}_t, \boldsymbol{x}_0)\right) \\
&= \frac{1}{2\sigma_\theta^2}\left\|\frac{1}{\sqrt{\alpha_t}}\left(\boldsymbol{x}_t + \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t)\right) - \frac{1}{\sqrt{\alpha_t}}\left(\boldsymbol{x}_t + \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\boldsymbol{\epsilon}_t\right)\right\|^2 \\
&= \frac{1}{2\alpha_t\sigma_\theta^2}\left\|\frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} \cdot (\boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t) - \boldsymbol{\epsilon}_t)\right\|^2 \\
&= \frac{(1-\alpha_t)^2}{2\alpha_t(1-\bar{\alpha}_t)\sigma_\theta^2}\left\|\boldsymbol{\epsilon}_\theta(\boldsymbol{x}_t, t) - \boldsymbol{\epsilon}_t\right\|^2 \\
&= \frac{(1-\alpha_t)^2}{2\alpha_t(1-\bar{\alpha}_t)\sigma_\theta^2}\left\|\boldsymbol{\epsilon}_\theta\left(\sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \left(\sqrt{1-\bar{\alpha}_t}\right) \cdot \boldsymbol{\epsilon}_t, t\right) - \boldsymbol{\epsilon}_t\right\|^2
\end{aligned}$$

Nevertheless, the training performance using the loss without the weight term is way better (Ho et al, 2020). In short,

$$L_{simple}^{(t)} = \left\|\boldsymbol{\epsilon}_\theta\left(\sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \left(\sqrt{1-\bar{\alpha}_t}\right) \cdot \boldsymbol{\epsilon}_t, t\right) - \boldsymbol{\epsilon}_t\right\|^2$$

Then, the final objective is simply just:

$$\boxed{L_{simple} = \sum_{t=2}^{T}\left\|\boldsymbol{\epsilon}_\theta\left(\sqrt{\bar{\alpha}_t}\boldsymbol{x}_0 + \left(\sqrt{1-\bar{\alpha}_t}\right) \cdot \boldsymbol{\epsilon}_t, t\right) - \boldsymbol{\epsilon}_t\right\|^2 + C}$$

where C is the constant term that is not related to θ.

B. Proof of Score function of diffusion model

Using the diffusion process annotation, the score approximates

$$s_\theta(x_t, t) \approx \nabla_{x_t} \log q(x_t)$$

Given a Gaussian distribution $x \sim N(\mu, \sigma^2 I)$, we can write the derivative of the logarithm of its density function as

$$\nabla_x \log p(x) = \nabla_x \left(-\frac{1}{2\sigma^2}(x-\mu)^2\right) = -\frac{x-\mu}{\sigma^2} = -\frac{\epsilon}{\sigma} \text{ where } \epsilon \sim N(0, I)$$

Recall that $q(x_t|x_0) \sim N\left(\sqrt{\bar{\alpha}_t} x_0, (1-\bar{\alpha}_t) I\right)$, and therefore:

$$s_\theta(x_t, t) \approx \nabla_{x_t} \log q(x_t)$$

$$= E_{q(x_0)}\left[\nabla_{x_t} q(x_t|x_0)\right]$$

$$= E_{q(x_0)}\left[-\frac{\epsilon_\theta(x_t,t)}{\sqrt{1-\bar{\alpha}_t}}\right]$$

$$= -\frac{\epsilon_\theta(x_t,t)}{\sqrt{1-\bar{\alpha}_t}}$$