Spring 2025
Introduction to Artificial Intelligence
Offline Learning Module 3: Generative AI

112550109 楊榮竣

1. Implementation & Results

Part 1: Anime Face Generation

Part 1-1: Denoising Process

```
##############################
# TODO1-1-1: Denoising Process - Initialization
# Begin your code

# raise NotImplementedError('
#
# 1-1-1: Denoising Process - Initialization')

if beta_schedule == 'linear':
    beta_schedule_fn = linear_beta_schedule
else:
    raise ValueError(f'unknown beta schedule {beta_schedule}')

betas = beta_schedule_fn(timesteps, **schedule_fn_kwargs)

alphas = 1.0 - betas

alphas_bar = torch.cumprod(alphas,dim=0)

return betas, alphas, alphas_bar

# End your code
##############################
```

init

```
##############################
# 1-1-2: Denoising Process - Forward function
# Begin your code

# raise NotImplementedError('TODO1-1-2: Denoising Process - Forward function')

alphas_bar_t = extract(self.alphas_bar, t, x_start.shape)
mean = torch.sqrt(alphas_bar_t) * x_start
std = torch.sqrt(1.0-alphas_bar_t)

return mean + std * noise

# End your code
##############################
```

forward

```
##############################
# TODO1-1-3: Denoising Process - Backward function
# Begin your code

# raise NotImplementedError('TODO1-1-3: Denoising Process - Backward function')

alpha_bar_t = self.alphas_bar[t]
alpha_bar_t_prev = self.alphas_bar[t_prev]
var = (eta**2) * (1.0 - alpha_bar_t_prev)/(1.0 - alpha_bar_t) * (1.0 - alpha_bar_t/alpha_bar_t_prev)
mean = torch.sqrt(alpha_bar_t_prev)*pred_x_start + torch.sqrt(1.0-alpha_bar_t_prev-var) * pred_noise

return mean + torch.sqrt(var)*noise
# End your code
##############################
```

backward

init(): given beta_t, alpha is equal to (1 - beta), then using **cumprod()** to compute multiple of alphas form 1 to t. **Forward**:Firstly, it use **extract** function to get alpha_bar_t with corresponding shape from alpha_bar. Then compute mean value is square root of alpha_bar_t multiple x_start. And, std is equal to square root of 1 - alpha_bar_t. Finally, it return a value by computing means + std * noise. **Backward**:Firstly, it get alpha_bar_t and alpha_bar_prev from alpha_bar. Then, var is computed by following function.

$$\sigma_t^2 = \eta^2 \cdot \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t} \cdot \left(1 - \frac{\bar{\alpha}_t}{\bar{\alpha}_{t-1}}\right)$$

Mean is computed using alpha_bar, x_start, alpha_bar_prev, var, and pred_noise. These variable are computed above, so it can use them to compute mean value. Finally, return the value with this equation.

$$mean + \sqrt{var} \cdot noise$$

Part 1-2: Result Visualization

```
# Begin your code

for time, time_prev in tqdm(time_pairs, desc = 'sampling loop time step'):
    time_cond = torch.full((batch,), time, device = device, dtype = torch.long)
    self_cond = x_start if self.self_condition else None
    pred_noise, x_start, *_ = self.model_predictions(img, time_cond, self_cond, clip_x_start = True, rederive_pred_noise = True)
    noise = torch.randn_like(x_start)
    img = self.denoise_backward(x_start, time, time_prev, pred_noise, noise, eta)
    imgs.append(img)
ret = img if not return_all_timesteps else torch.stack(imgs, dim = 1)
ret = self.unnormalize(ret)

num_imgs = 5
num_steps = len(imgs)
fig, axs = plt.subplots(num_imgs, num_steps, figsize=(num_steps * 1.5, num_imgs * 2))
for idx, i in enumerate(torch.randperm(batch)[:num_imgs]):
    for j in range(num_steps):
        img_to_show = self.unnormalize(imgs[j][i].detach().cpu())
        img_to_show = img_to_show.permute(1, 2, 0).numpy()
        img_to_show = np.clip(img_to_show, 0, 1)
        axs[idx, j].imshow(img_to_show)
        axs[idx, j].set_title(f'Step {j+1}')
        axs[idx, j].axis('off')
plt.tight_layout()
plt.savefig('denoising_progress.png')
plt.show()
# End your code
##############################
```

Denoising Progress

There are two part of code in this function. The first part of code is used to compute progress of denoising. It use model_predictions function to get the prediction of noise and x_start. Then, using denoise_backward to compute the denoising image. After computing, it store a new image into imgs and use new image to compute above step until reaching to end time. Finally, using unnormalize function to transform imgs and store it into ret.

The second part of code is used to show the all progress of denoising. It will show five different image with number of time step. Firstly, **subplot** create a graph with a grid of subplots which set according to num_imgs and num_steps. Then, randomly go through five image and their progress of denoising. It store progress into graph with correspond position. Using **unnormalize** function, **premute().numpy()**, and **clip()** function to change image into visualization form. Finally, use tight_layout() to adjusts the padding between subplots to ensure they don't overlap and use the available figure space efficiently, and savefig to save figure as 'denoising_progress.png'.

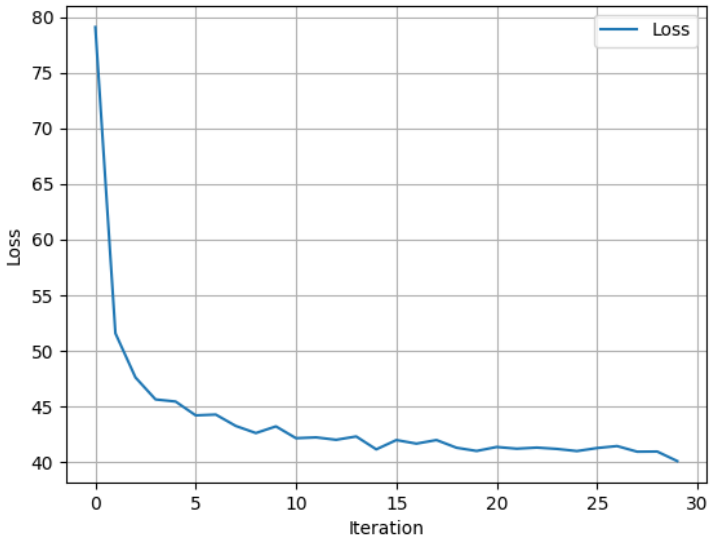
denoising_progress

```python
def plot_losses(self):
    ##############################
    # TODO1-2-2:  Result Visualization - Loss curve
    # You can use self.losses directly to plot the loss curve.
    # Begin your code

    # raise NotImplementedError('TODO1-2-2:  Result Visualization - Loss curve')
    plt.figure()
    plt.plot(self.losses,label='Loss')
    plt.xlabel('Iteration')
    plt.ylabel('Loss')
    plt.legend()
    plt.grid(True)
    plt.savefig('Loss curve')
    plt.show()

    # End your code
    ##############################
```

plot_loss

In this part, I use function in matplotlib.pyplot(plt) to complete plotting the loss of training.

- figure() : create a new figure to plot the graph.
- plot() : the main function to plot the graph with input list. It will set the label of plot with input 'label'.
- ylabel() : setting the label of y-axis in the graph.
- legend() : place a legend on the axes.
- savefig() : save graph with input name.
- show() : show the graph after plotting.



Loss Curve

Part 1-3: Evaluation Baseline

```
########## Hyper-Parameters ##########
seed = 114514
path = '/kaggle/input/diffusion/faces/faces'
IMG_SIZE = 64
batch_size = 32
train_num_steps = 30000
lr = 2e-4
grad_steps = 1
ema_decay = 0.995

channels = 128
dim_mults = (1, 2, 3, 4)

timesteps = 10000
sampling_timesteps = 50
beta_schedule = 'cosine'
# beta_schedule = 'linear'

# End your code
##############################
```

hyper-parameter setting

```
FID score: 72.38
AFD rate:  94.30%
Final score: 18/20
```

Result Score

Result & Discussion

Upon examining the generated images from the first test, I observed that some images exhibited blurred edges in middle. Additionally, the facial images appeared to be asymmetrically generated, with the left and right sides looking as though they originated from different sources.

Example images of first test

To solve observed problem, I try some strategies to improve performance of model.

Parameter improve

- Decrease learning rate → keep model more stable
- Increase ema_decay → Smooths model parameter updates over time
- Large train steps → Allows the model to see more data and fine-tune its generative capacity.
- Large sample time step → Allows the model to see more data and fine-tune its generative capacity.

Unet model improve

- Large channel size → Increases the number of filters at each layer, improving the model's ability to capture
- Deeper model → Enables hierarchical feature learning and a better understanding of spatial structures.

Cosine Beta Schedule

I use cosine beta schedule to make model more stable in progressing. This schedule can avoid add too little noise in the begin and add too much noise in the end. Below picture is formula for cosine beta schedule.

$$\bar{\alpha}_t = \frac{f(t)}{f(0)} \quad \text{where } f(t) = \cos^2\left(\left(\frac{t/T + s}{1+s}\right) \cdot \frac{\pi}{2}\right)$$

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

```python
def cosine_beta_schedule(timesteps, s=0.008):
    steps = timesteps + 1
    x = torch.linspace(0, timesteps, steps, dtype=torch.float64) / timesteps
    alphas_cumprod = torch.cos(((x + s) / (1 + s)) * math.pi * 0.5) ** 2
    alphas_cumprod = alphas_cumprod / alphas_cumprod[0]
    betas = 1 - (alphas_cumprod[1:] / alphas_cumprod[:-1])
    return torch.clip(betas, 1e-8, 0.9999)
```

Implement of cosine beta schedule

Reference:
`https://www.zainnasir.com/blog/cosine-beta-schedule-for-denoising-diffusion-models/`

	Test 1	**Test 2**	**Test 3**	**Test 4**
FID	126.69	88.94	82.81	72.38
AFD(%)	72.20	91.90	93.40	94.30

FID and AFD score of each Test

Parameter	**Test 1**	**Test 2**	**Test 3**	**Test 4**
Train Steps	20000	20000	30000	30000
Learning Rate	1×10^{-4}	2×10^{-4}	1×10^{-4}	2×10^{-4}
EMA Decay	0.997	0.997	0.995	0.995
Channels	32	64	64	128
Dim Mults	(1, 2, 4)	(1, 2, 4, 8)	(1, 2, 4, 8)	(1, 2, 3, 4)
Timesteps	1000	1000	4000	10000
Sampling Timesteps	50	50	50	50
Beta Schedule	cosine	Linear	cosine	cosine

Parameter Setting

In **Test 1**, I modified the initial hyper-parameter settings by increasing the number of timesteps and sampling timesteps, as well as raising the EMA decay to 0.997. However, the model had relatively low capacity (32 channels and fewer dimension multipliers), which likely limited its ability to capture complex features, resulting in poor performance (FID: 126.69, AFD: 72.20%).

In **Test 2**, I increased the learning rate to 2×10^{-4}, used a deeper model with more dimension multipliers $(1, 2, 4, 8)$, and doubled the number of channels to 64. These changes significantly improved the model's learning efficiency and representational power, leading to a notable improvement in both FID (88.94) and AFD (91.90%).

For **Test 3**, I reverted to the initial learning rate (1×10^{-4}) and further increased the number of timesteps to 4000 while keeping the model structure from Test 2. The increased timesteps allowed for finer denoising steps during training, enhancing image detail and quality. As a result, FID improved to 82.81 and AFD rose to 93.40%, showing better image realism.

In **Test 4**, I further increased the channel size to 128 and retained the higher timestep count. Additionally, I adjusted the EMA decay to 0.995, allowing for more responsive weight updates. These changes provided

greater model capacity and training stability, which led to the best performance among all tests, with the lowest FID (72.38) and highest AFD (94.30%).

Overall, the trend suggests that increasing model complexity, training duration, and timesteps, combined with careful tuning of EMA decay and beta schedules, leads to steady and significant improvements in generative performance as measured by both FID and AFD. Sample images of final test are show on below, I observe that there still exist some face are strange. However, most of them have significant improvement.

Example images of final test

Part 2: Optical Illusion Generation

Part 2-1: Prompt Design

- prompt 1 : A cyberpunk-style alien spaceship hovers above a rural Earth farm. In the foreground, aliens exit the ship and capture a horse and a cow for food. The background shows a glowing city skyline with neon lights and flying drones.
- prompt 2 : In a futuristic city, people drive flying cars above the roads. The sky is filled with bright aerial highways, while the ground level is lined with glowing skyscrapers and digital billboards. The scene is viewed from a rooftop at sunset.

In this part, I aimed to create an image with a cyberpunk and science fiction style, focusing on two main elements: aliens and vehicles. To achieve this, I developed two distinct prompts—one centered on the alien theme and the other on futuristic vehicles. First prompt describe aliens and second prompt describe vehicles. a To ensure the model generates the desired visual context, I included detailed background descriptions such as neon lights, drones, digital billboards, and glowing skylines. These settings enhance the science fiction

atmosphere and support the overall visual complexity, helping to create compelling imagery suitable for an optical illusion task.

Part 2-2: Viewing Transformation

```python
class IdentityView:
    def __init__(self):
        pass

    def view(self, im):
        return im

    def inverse_view(self, noise):
        return noise


class Rotate180View:
    def __init__(self):
        pass

    def view(self, im):
        return T.functional.rotate(im, 180)

    def inverse_view(self, noise):
        return T.functional.rotate(noise, 180)
```

views

Identity View will return original image and noise. Rotate180View will return a image with rotating with 180 degree. I use T.function.rotate to complete this function.

Part 2-3: Denoising Operation

```python
@torch.no_grad()
def denoising_loop(model, noisy_images, prompt_embeds, views,
                   timesteps, guidance_scale, generator, noise_level=None, upscaled=None):

    # Use the first prompt embedding
    num_prompts = len(views)

    # Condition on noise level, for each model input (used in stage 2)
    if noise_level is not None:
        noise_level = torch.cat([noise_level] * num_prompts * 2)

    for i, t in enumerate(tqdm(timesteps)):

        # If upscaled is provided (stage 2), concatenate with noisy_images
        if upscaled is not None:
            model_input = torch.cat([noisy_images, upscaled], dim=1)
        else:
            model_input = noisy_images

        viewed_imgs = [view.view(model_input[0]) for view in views]
        model_input = torch.stack(viewed_imgs)

        model_input = torch.cat([model_input] * 2)
        model_input = model.scheduler.scale_model_input(model_input, t)

        # Predict noise estimate
        noise_pred = model.unet(
            model_input, t, encoder_hidden_states=prompt_embeds,
            class_labels=noise_level, cross_attention_kwargs=None, return_dict=False
        )[0]
```

(a) First image

```python
        # Extract uncond (neg) and cond noise estimates
        noise_pred_uncond, noise_pred_text = noise_pred.chunk(2)

        # invert pred result
        inverted_preds_uncond = [view.inverse_view(pred) for pred, view in zip(noise_pred_uncond, views)]
        inverted_preds_text = [view.inverse_view(pred) for pred, view in zip(noise_pred_text, views)]
        noise_pred_uncond = torch.stack(inverted_preds_uncond)
        noise_pred_text = torch.stack(inverted_preds_text)

        # Split into noise estimate and variance estimates
        # Split predicted noise and predicted variances
        splited_size = model_input.shape[1] // (2 if upscaled is not None else 1)
        noise_pred_uncond, _ = noise_pred_uncond.split(splited_size, dim=1)
        noise_pred_text, predicted_variance = noise_pred_text.split(splited_size, dim=1)
        noise_pred = noise_pred_uncond + guidance_scale * (noise_pred_text - noise_pred_uncond)

        # Reduce predicted noise and variances
        # Combine noise estimates (and variance estimates)
        img_size = 256 if upscaled is not None else 64
        noise_pred = noise_pred.view(-1,num_prompts,3,img_size,img_size)
        predicted_variance = predicted_variance.view(-1,num_prompts,3,img_size,img_size)
        noise_pred = noise_pred.mean(1)
        predicted_variance = predicted_variance.mean(1)

        noise_pred = torch.cat([noise_pred, predicted_variance], dim=1)
        # Compute the previous noisy sample x_t -> x_t-1
        noisy_images = model.scheduler.step(noise_pred, t, noisy_images, generator=generator, return_dict=False)[0]

    return noisy_images

# End your code
##############################
```

(b) Second image

Denoising Operation

Firstly, I assign num_prompts as length of views. Then, delete following two lines of code to make the prompt_embeds as original prompt_embeds.

- **pos_idx, neg_idx = 0, prompt_embeds.shape[0]//2**
- **prompt_embeds = torch.stack([prompt_embeds[pos_idx],prompt_embeds[neg_idx]])**

Then, this two lines of code is to use view on model_input.

- **viewed_imgs = [view.view(model_input[0]) for view in views]**
- **model_input = torch.stack(viewed_imgs)**

Inverting the transformation applied earlier by following codes to make the denoised outputs to be in the original image space before averaging them.

- **inverted_preds_uncond = [view.inverse_view(pred) for pred, view in zip(noise_pred_uncond, views)]**
- **inverted_preds_text = [view.inverse_view(pred) for pred, view in zip(noise_pred_text, views)]**
- **noise_pred_uncond = torch.stack(inverted_preds_uncond)**
- **noise_pred_text = torch.stack(inverted_preds_text)**

After reverting the views, I reshape and average across the num_prompts views. This fuses information from multiple views into a single coherent denoised image.

- **noise_pred = noise_pred.view(-1,num_prompts,3,img_size,img_size)**
- **predicted_variance = predicted_variance.view(-1,num_prompts,3,img_size,img_size)**
- **noise_pred = noise_pred.mean(1)**

Part 2-4: Evaluation Baseline

Result

```
Prompt: A cyberpunk-style alien spaceship hov
CLIP Score: 0.3487

Prompt: In a futuristic city, people drive fl
CLIP Score: 0.3574
```

CLIP score

Result & Discussion

In this part, I totally try four pair of prompts to achieve the CLIP score up to baseline. Below, I present each prompt pair along with their generated image results and corresponding CLIP scores.

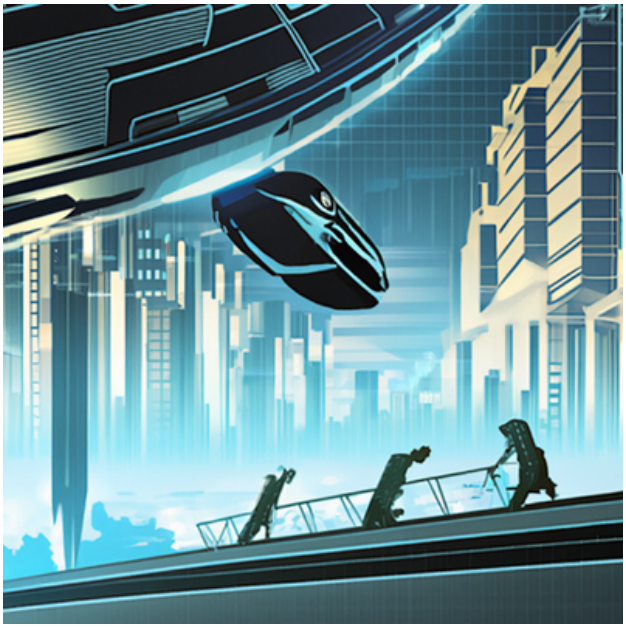
Test	Prompt 1 (Alien)	Prompt 2 (Vehicle)	Design Idea
1	An aliens land space ship on earth, and they catch animal to feed as food.	A future picture with advanced technology. Human can drive cars in the sky.	Simple description on target element.
2	A cyberpunk picture with aliens driving the space ship, landing on the earth, and catching horses and cow to feed them as food.	A future scene with human have high-advanced technology which can make people drive car in the sky and city will fill with skyscraper.	Continue to prompts of test 1. Added cyberpunk style and slightly more detail about the spaceship and actions.
3	A detailed cyberpunk scene featuring aliens piloting a futuristic spaceship, landing on Earth, and capturing horses and cows to use as food. The landscape is neon-lit, with glowing cityscape and dystopian skies.	A futuristic city where advanced human technology allows people to drive flying cars. The skyline is filled with towering skyscrapers, holographic billboards, and glowing aerial highways.	Continue to prompts of test 2. Expanded scene with glowing city elements and clearer futuristic atmosphere.
4	A cyberpunk-style alien spaceship hovers above a rural Earth farm. In the foreground, aliens exit the ship and capture a horse and a cow for food. The background shows a glowing city skyline with neon lights and flying drones.	In a futuristic city, people drive flying cars above the roads. The sky is filled with bright aerial highways, while the ground level is lined with glowing skyscrapers and digital billboards. The scene is viewed from a rooftop at sunset.	Modify from prompts of test 3. Added foreground setting (farm), sunset lighting, and stronger scene composition.

Pair of prompts for each test

	Test 1	Test 2	Test 3	Test 4-1	Test 4-2
Prompt 1 Score	0.2682	0.2564	0.3248	0.3890	0.3487
Prompt 2 Score	0.2848	0.3011	0.3413	0.3536	0.3574

CLIP scores of four tests.

(a) Test 1



(b) Test 2

(c) Test 3

(d) Test 4-1

Result Picture

Initially, I start from simple prompts in test 1, but I observe that the generated image can not focus on target elements. For example, the result of test 1 only spaceship and vehicle in the image. I think the reason is that prompts is too simple to make model difficult to catch feature on target elements.

To improve model, I added more detail information about target elements. The result pf test 2 is better, however, it is not enough. The CLIP score only one of then exceed 0.3. We can observe that there are something like alien exist in the image.

Then, I try some method like add some description about scene and background, such as skyscraper and highways. Furthermore, I add some objective, such as detailed and futuristic, on the scene and city to make model can catch more feature of image. In this part, the CLIP of both prompt are higher than 0.3, and target

elements appear in the image. However, I feel this image is not clear, and somewhere in the picture is vague.

Finally, I modify prompts of test 3, I try to use another sentence to describe the same scene, and generate two times(Test 4-1 and result picture in part 2-4). The all CLIP score of result is higher than 0.3. We can observe that the vague scene problem will be solve in this test, and both of score are higher than test 3.

2. Results Link

Result Link: `https://drive.google.com/drive/folders/1VindpOw4-V2e459pNesRKdaBDXVo_b8c?usp=drive_link`