Lab6 - Generative Models

Shyang-En Weng 翁祥恩 413551036

May 6, 2025

1 Introduction

Image generation has become one of the hottest topics in recent years, particularly driven by the rapid advancements in diffusion models. These models, especially Conditional Diffusion Models, have shown impressive capabilities in generating synthetic images guided by various forms of prior knowledge, such as class labels, text descriptions, or other modalities. Starting from the basic concept of Diffusion Models, which learn to reverse a noise process to generate realistic images, conditional versions extend this by incorporating external guidance, allowing for more controllable and semantically meaningful outputs. In this lab, we explore the fundamentals of image generation through conditional DDPM [1] and implement the effect of priors to generate corresponding synthetic images, which are images with different objects with corresponding colors and shapes. The highlighted points of the implementation are shown below:

- In this lab, I implemented a Conditional Denoising Diffusion Probabilistic Model (Conditional DDPM) using class labels as guidance, and successfully achieved high classification accuracy with the auxiliary classifier.
- I also compared the effects of different total diffusion timesteps, demonstrating the trade-off between image quality and generation speed across various settings.

2 Implementation Details

In this lab, we aim to generate images containing objects that correspond to given input labels. This is achieved using the Denoising Diffusion Probabilistic Model (DDPM) [1], enhanced with prior knowledge in the form of class labels. Starting from pure noise, the model progressively denoises the input while being guided by the label information, ultimately generating the expected image that aligns with the specified label.

2.1 Denoising Diffusion Probabilistic Models (DDPM)

2.1.1 Theory

Denoising Diffusion Probabilistic Models (DDPM) [1] are a class of generative models that synthesize data by learning to reverse a diffusion process. The forward diffusion process gradually adds Gaussian noise to a data sample $\mathbf{x}_0$ over T timesteps, resulting in a sequence of noisy samples $\mathbf{x}_1, \ldots, \mathbf{x}_T$.

The forward process is defined as a Markov chain with Gaussian transitions:

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

where β_t is a variance schedule controlling the amount of noise added at each timestep.

The marginal distribution of $\mathbf{x}_t$ given $\mathbf{x}_0$ can be derived as:

$$q(\mathbf{x}_t|\mathbf{x}_0) = \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\mathbf{I}) \tag{2}$$

where $\bar{\alpha}_t = \prod_{s=1}^{t}(1-\beta_s)$.

The goal of the model is to learn the reverse process:

$$p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) \tag{3}$$

which is also modeled as a Gaussian:

$$p_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) = \mathcal{N}(\mathbf{x}_{t-1}; \mu_\theta(\mathbf{x}_t, t), \Sigma_\theta(\mathbf{x}_t, t)) \tag{4}$$

In practice, the model learns to predict the added noise ϵ through a neural network ϵ_θ.

The training objective usually compares the difference between the estimated noise and added noise using mean squared error:

$$L_{\text{DDPM}} = \mathbb{E}_{\mathbf{x}_0,\epsilon,t}\left[\|\epsilon - \epsilon_\theta(\mathbf{x}_t, t)\|^2\right] \tag{5}$$

where $\epsilon \sim \mathcal{N}(0, \mathbf{I})$ and $\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon$.

In the conditional variant, the noise prediction network is guided by additional prior knowledge, such as a class label y:

$$\epsilon_\theta(\mathbf{x}_t, t, y) \tag{6}$$

This allows the model to generate images that correspond to specific label conditions.

2.1.2 Implementation of UNet

In this lab, I implemented a Conditional DDPM based on the `diffusers` toolkit [2] from Hugging Face. The core architecture is a UNet with a depth of 4. The first downsampling block and the last upsampling block are standard convolutional blocks (`DownBlock2D` and `UpBlock2D`, respectively), while the intermediate blocks incorporate attention mechanisms (`AttnDownBlock2D` and

`AttnUpBlock2D`) to enhance feature learning and preserve detailed spatial information in deeper layers. The model is designed to condition class labels, enabling the generation of images aligned with the given semantic guidance. The model structure is as follows:

Listing 1: Conditional DDPM using Hugging Face diffusers

```
class cDDPM(nn.Module):
    def __init__(self, num_classes=24,
        n_channel=32, depth=4, sample_size=64):
        super().__init__()
        self.unet = UNet2DModel(
            sample_size=sample_size,
            in_channels=3,
            out_channels=3,
            layers_per_block=2,
            block_out_channels=[n_channel * (i+1) for i in range(depth)],
            down_block_types=["DownBlock2D"] + ["AttnDownBlock2D"] * (depth - 1)
            up_block_types=["AttnUpBlock2D"] * (depth - 1) + ["UpBlock2D"],
            mid_block_type="UNetMidBlock2D",
            class_embed_type="identity",
            num_class_embeds=num_classes,
        )
        self.cls_emb = nn.Linear(num_classes, n_channel * depth)

    def forward(self, x, t, label):
        cls_emb = self.cls_emb(label)
        return self.unet(x, t, cls_emb).sample
```

This architecture enables label-conditioned generation, where class embeddings are projected and injected into the UNet to guide the denoising process. This setup ensures that the generated images correspond to the desired object categories, with enhanced attention to critical features.

2.2 Noise Schedule

The noise schedule plays a crucial role in the performance of DDPMs, as it determines how noise is added during the forward process and subsequently reversed during sampling. In this implementation, we use a linear noise schedule for the variance β_t over T timesteps:

$$\beta_t = \beta_{\min} + \frac{t}{T}(\beta_{\max} - \beta_{\min}), \quad t = 1, \ldots, T \tag{7}$$

This schedule gradually increases the amount of noise added at each step, which helps the model learn a smooth denoising trajectory. From the variance values, we compute the corresponding $\alpha_t = 1 - \beta_t$ and the cumulative product $\bar{\alpha}_t = \prod_{s=1}^{t} \alpha_s$, which are used during training and sampling:

$$\alpha_t = 1 - \beta_t \tag{8}$$

$$\bar{\alpha}_t = \prod_{s=1}^{t} \alpha_s \tag{9}$$

These quantities are used to reparameterize the noisy inputs at each timestep as:

$$\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1 - \bar{\alpha}_t}\,\epsilon, \quad \epsilon \sim \mathcal{N}(0, \mathbf{I}) \tag{10}$$

where ϵ is sampled noise. The model learns to predict ϵ to recover the original image through the reverse process.

The implementation is simple and modular, as shown in Listing 2:

Listing 2: Noise schedule.

```
noise_scheduler = DDPMScheduler(
    num_train_timesteps=args.timesteps,
    beta_start=0.0001,
    beta_end=0.02,
    beta_schedule=args.noise_schedule,
)
```

After understanding how diffusion works with the noise scheduling, we can implement the training and testing procedures for the Conditional DDPM. During training, we randomly select a timestep $t \in [1, T]$ for each image in a batch, and then generate the corresponding noisy image $\mathbf{x}_t$ using the forward diffusion process. The model is trained to predict the added noise ϵ given the noisy input $\mathbf{x}_t$, the timestep t, and the conditioning label. The objective is to minimize the difference between the predicted noise and the true noise at the corresponding timestep. For testing, we start from a pure Gaussian noise image $\mathbf{x}_T$ and iteratively apply the learned denoising model in reverse order of timesteps from T to 1. At each step, the model predicts the noise to be removed, and the image is updated accordingly using the noise scheduler. This guided sampling procedure results in a final synthetic image $\mathbf{x}_0$ that reflects the conditioning label and the learned data distribution.

The implementation of training and testing procedures is shown in Listings 3 and 4.

Listing 3: Training loop for Conditional DDPM

```
for epoch in range(args.start_epoch, args.num_epochs):
    model.train()
    total_loss = 0.
    loader = tqdm(train_loader)
    for i, (images, labels) in enumerate(loader):
        optimizer.zero_grad()
```

```
            images = images.to(device)
            labels = labels.float().to(device)
            # Forward
            noise = torch.randn_like(images)
            t = torch.randint(0, args.timesteps, (images.size(0),),
                device=device).long()
            noisy_images = noise_scheduler.add_noise(images, noise, t)

            with autocast(device_type=device.type, enabled=args.amp):
                outputs = model(noisy_images, t, labels)
                loss = criterion(outputs, noise)

            if scaler is not None:
                scaler.scale(loss).backward()
                scaler.step(optimizer)
                scaler.update()
            else:
                loss.backward()
                optimizer.step()

            total_loss += loss.item()

            avg_loss = total_loss / len(train_loader)

        scheduler.step()
```

Listing 4: Testing loop for Conditional DDPM

```
for i, labels in enumerate(loader):
    images = torch.randn((labels.size(0), 3,
        args.image_size, args.image_size)).to(device)
    labels = labels.float().to(device)
    denosing_process = []
    for t in tqdm(noise_scheduler.timesteps):
        n_t = model(images, t, labels)
        images = noise_scheduler.step(n_t, t, images).prev_sample

        if (t + 1) % args.save_interval == 0 or t == 0:
            denosing_process.append(denormalize(images).cpu())

    denosing_process = torch.stack(denosing_process, dim=0)
    denosing_process = denosing_process.reshape(-1, 3, args.image_size, args.ima
    save_image(denosing_process,
        os.path.join(save_dir, f"denosing_process_{i}.png"), nrow=denosing_proce
    save_image(denormalize(images).cpu(),
        os.path.join(save_dir, f"final_{i}.png"), nrow=denosing_process.size(0))
```

```
acc = eval_model.eval(images, labels)
total_acc += acc

loader.set_postfix({"acc": total_acc/(i+1)})
```

3 Results and discussion

3.1 Experiment results

The evaluation results of my implementation are shown in Figs. 1 and 2, the denoising results are shown in Fig. 3, and the demonstration of 32 images on test set and new test set are shown in Figs 4 and 5; both results look good where achieving an accuracy over 90%, 92% for the test set and 95% for the new test set, respectively.

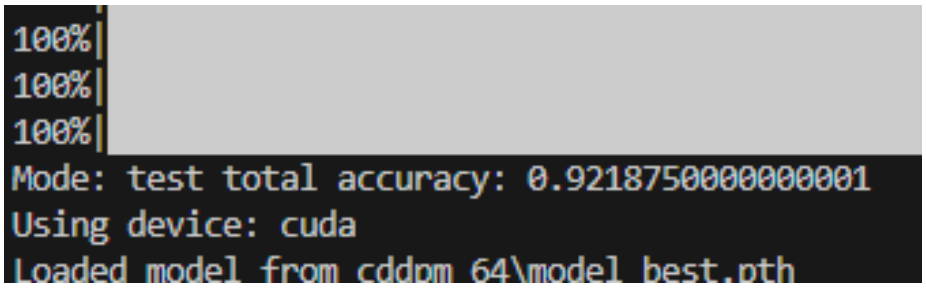

Figure 1: Screenshot of evaluation results on the test set.

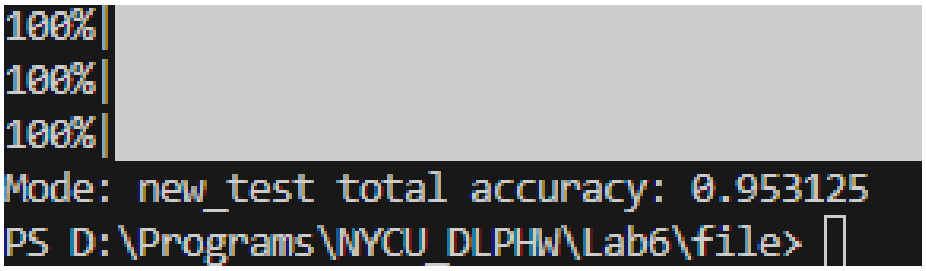

Figure 2: Screenshot of evaluation results on the new test set.

3.2 Extra implementations or experiments

In the extra experiments, I further compared the performance of the model using 300 timesteps versus 1000 timesteps. To accelerate training for this comparison, the number of channels was reduced to half of the size used in the previous

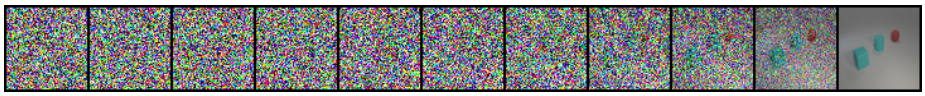

Figure 3: Denoising process image with the label set ["red sphere", "cyan cylinder", "cyan cube"].

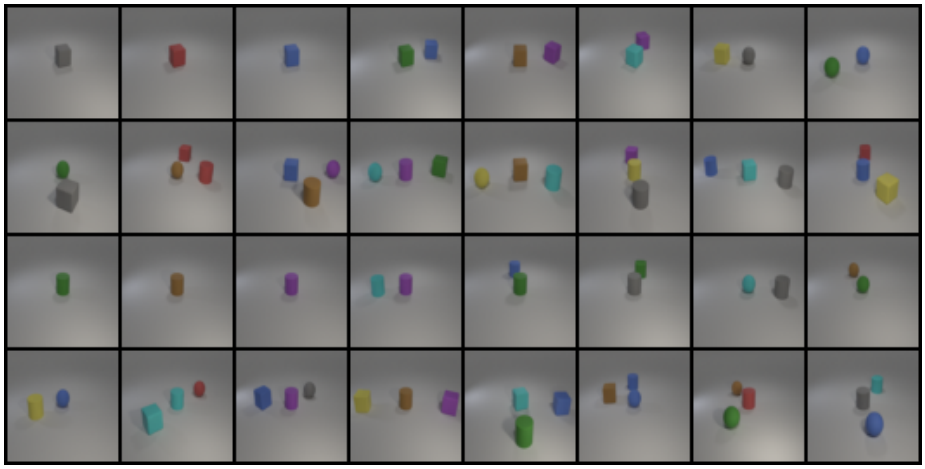

Figure 4: Demonstration of the conditional DDPM on the test set.

experiments. The resulting learning curves are shown in Fig. 6. From the figure, we observe that the number of diffusion timesteps significantly affects the model's performance. Specifically, models trained with a higher number of timesteps tend to converge more smoothly and achieve better reconstruction quality, highlighting the importance of choosing an appropriate timestep setting for balancing computational efficiency and generation quality.

References

[1] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *Advances in neural information processing systems*, 33:6840–6851, 2020.

[2] Patrick von Platen, Suraj Patil, Anton Lozhkov, Pedro Cuenca, Nathan Lambert, Kashif Rasul, Mishig Davaadorj, Dhruv Nair, Sayak Paul, William Berman, Yiyi Xu, Steven Liu, and Thomas Wolf. Diffusers: State-of-the-art diffusion models. https://github.com/huggingface/diffusers, 2022.

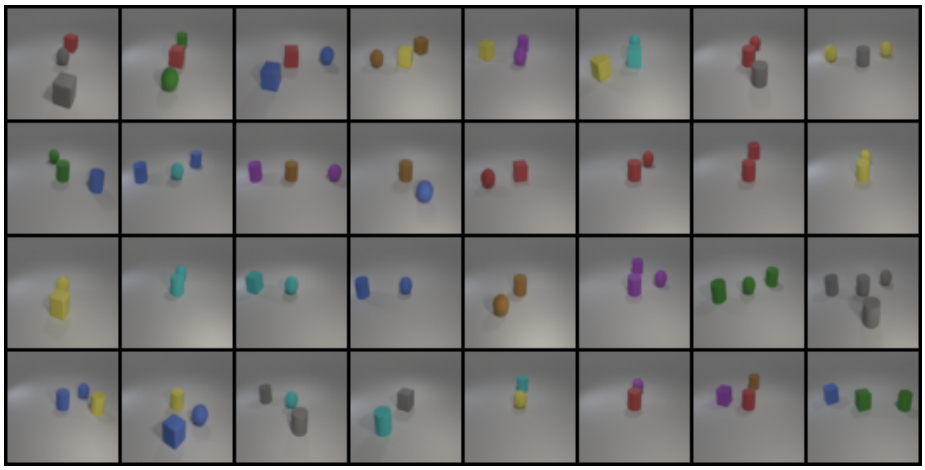

Figure 5: Demonstration of the conditional DDPM on the new test set.

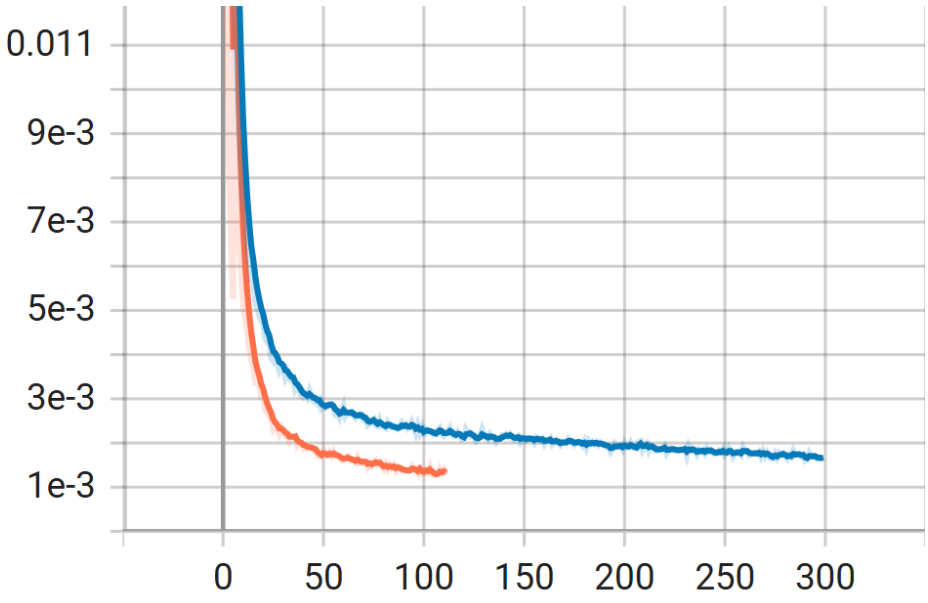

Figure 6: Learning curves of different timesteps. Orange: 1000 timesteps; blue: 300 timesteps.