DL-Lab7

0816095 Hsuan Wang

May 27, 2023

1 Introduction

In this lab we are asked to implement a conditional diffusion model. The model is trained on iclevr data set, which contains approximately 18000 images of geometric objects of various color, for example *redcube*, *bluecylinder*, it can also contain multiple object in one image. Our goal is to train the model to generate the right images based on the conditions.

2 Implementation details

2.1 Choice of DDPM

2.1.1 Pixel domain diffusion model

I chose to implement a pixel domain diffusion model, because it is the most intuitive to me, you just need to diffuse and denoise on the pixel space, it's also easier to see the process of denoising.

2.1.2 Condition embedding

I tried two ways of embedding time, one-hot encoding with a single linear layer, and the sinusoidal positional encoding method in the transformer papers. As we will see in the results later, sinusoidal positional encoding perform much better then just plain one-hot encoding with linear layer.As to the labels, I just embed them with a single fully connected layer. Finally I combine the time embedding and label embedding to form condition embedding by adding them together and passing them through every layer of Unet.

```
def pos_encoding(self, t, channels):
    inv_freq = 1.0 / (
        10000
        ** (torch.arange(0, channels, 2, device=self.device).float() / channels)
    )
    pos_enc_a = torch.sin(t.repeat(1, channels // 2) * inv_freq)
    pos_enc_b = torch.cos(t.repeat(1, channels // 2) * inv_freq)
    pos_enc = torch.cat([pos_enc_a, pos_enc_b], dim=-1)
    return pos_enc
```

```
self.label_emb = nn.Linear(num_conditions, time_dim)
```

```
t = self.pos_encoding(t, self.time_dim)
if y is not None:
    t = t + self.label_emb(y)
```

2.1.3 Noise schedule

I used linear schedule for my noise as proposed in the 2020 paper.

```
    def prepare_noise_schedule(self):
        return torch.linspace(self.beta_start, self.beta_end, self.noise_steps)
```

2.1.4 Classifier-free guidance scale

Some times the model may ignore the condition term, or over-fit on it, generating undesirable images. Thus people use Classifier-free guidance, or cfg scale in short, to interpolate the results of model with and without conditions.

```
    predicted_noise = model(x, t, conds)
    if args.cfg_scale > 0:
        uncond_predicted_noise = model(x, t, None)
        predicted_noise = torch.lerp(uncond_predicted_noise, predicted_noise, args.cfg_scale)
```

2.2 UNet architectures

The Unet has an encoder a bottle-neck and a decoder. The *DoubleConv* is just a module that consist of two convolutional layers, *Down* is just a down sampling module that consist of maxpooling layers, and *sa* is a self attention layer. Our encoder starts with a *DoubleConv* then followed by three down sampling block and self attention layers in between. The bottle-neck consist of three *DoubleConv* layers stacked together. The decoder is basically the same as the encoder just in reverse.

```
class UNet_conditional(nn.Module):
    def __init__(self, c_in=3, c_out=3, time_dim=256, num_conditions=None, device="cuda"):
        super().__init__()
        self.device = device
        self.time_dim = time_dim
        self.inc = DoubleConv(c_in, 64)
        self.down1 = Down(64, 128)
        self.sa1 = SelfAttention(128, 32)
        self.down2 = Down(128, 256)
        self.sa2 = SelfAttention(256, 16)
        self.down3 = Down(256, 256)
        self.sa3 = SelfAttention(256, 8)

        self.bot1 = DoubleConv(256, 512)
        self.bot2 = DoubleConv(512, 512)
        self.bot3 = DoubleConv(512, 256)

        self.up1 = Up(512, 128)
        self.sa4 = SelfAttention(128, 16)
        self.up2 = Up(256, 64)
        self.sa5 = SelfAttention(64, 32)
        self.up3 = Up(128, 64)
        self.sa6 = SelfAttention(64, 64)
        self.outc = nn.Conv2d(64, c_out, kernel_size=1)
```

2.3 Loss function

I chose the simplified loss function, this makes the training process really simple to implement as well as having good results. All the model has to do is take in x_t and t (In which we can calculate x_t directly using $\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon$ where ϵ is just the noise sampled from standard Gaussian), predict the noise and calculate the mse between the predicted noise and the actual noise.

$$L_{t-1} = \mathbb{E}_{\mathbf{x}_0,\,\epsilon}\left[\left\|\epsilon - \epsilon_\theta\left(\sqrt{\bar{\alpha}}_t\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon,\, t\right)\right\|_2^2\right]$$

Figure 1: Equation of loss function

2.4 Hyperparameters

- Epochs : 300
- Batch size : 8
- Learning rate : 3e-4
- Noise steps : 300
- cfg scale : 3
- Linear β schedule start : 1e-4
- Linear β schedule end : 2e-2
- Time embedding dimension : 256

3 Results and discussion

I tried three different implementation:

- A : Sinusoidal positional encoding with cfg=3
- B : One-hot positional encoding with cfg=3
- C : Sinusoidal positional encoding with cfg=0 (no classifier-free guidance)

	test	new_test
A	0.8586	0.8893
B	0.8200	0.7927
C	0.7516	0.7646

Table 1: Comparison of best performance between different models

I trained each of the different implementation for around 35 epoch. As we can see from the results in the next page, One-hot positional encoding performs slightly worse then sinusoidal positional encoding, the image quality is also visibly inferior, there's often some residual noise in the final image.

Without implementing cfg scale, the accuracy drop significantly, about 10% lower then my best results, this suggest that the model does need interpolation between none label and label prediction, this makes the image generation more diverse, higher quality while still following the guidance of labels.

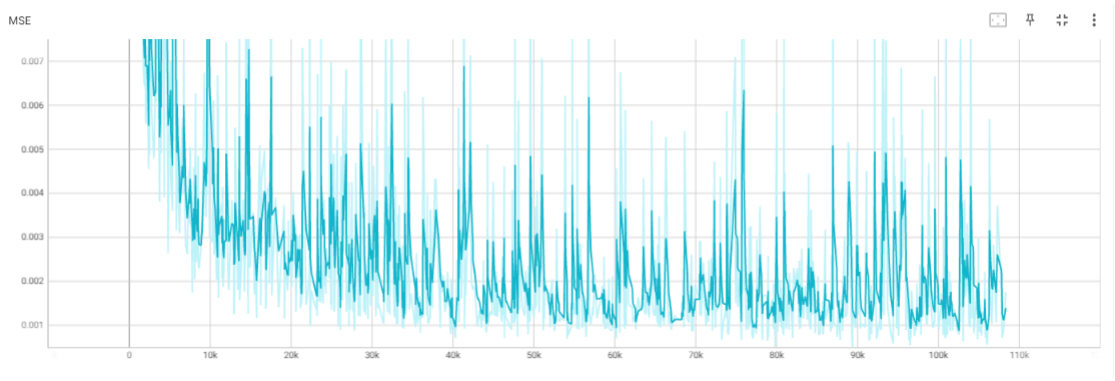

Figure 2: MSE loss training curve of experiment A

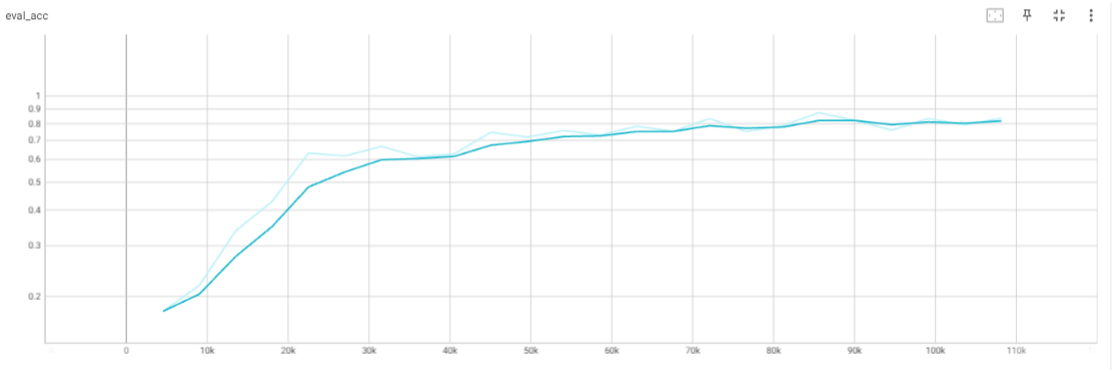

Figure 3: eval acc curve of experiment A

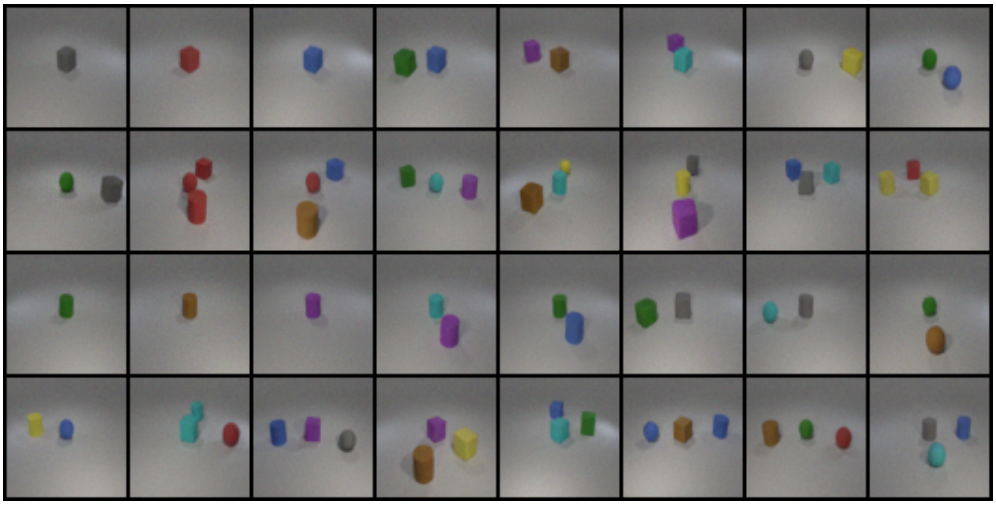

Figure 4: The synthetic images of accuracy of 0.8586

Figure 5: The synthetic images of accuracy of 0.8893