Deep Learning Lab6 Report

陳以瑄 (ID: 313551047)

1 Introduction

In this lab, I implement a conditional Denoising Diffusion Probabilistic Model (DDPM) to generate synthetic images based on multi-label conditions using the iCLEVR dataset. I modified the conditional DDPM code provided by Burak Bayrak on GitHub, which was originally designed for other datasets. In my experiments, the model achieved an accuracy of 0.8611 on the test dataset and 0.8333 on the new_test dataset.

2 Implementation Details

I modified the conditional DDPM implementation by Burak Bayrak, available on GitHub[1], which was originally developed for MNIST, FashionMNIST, and Sprite datasets.

2.1 Model Architecture —ContextUnet

The model architecture is based on the `ContextUnet` defined in `model.py` of the original implementation. The key components are as follows:

- **Input:** RGB images with 3 channels, size 64×64.
- **Encoder:**
  1. Begins with a residual convolutional block (3→64 channels).
  2. Followed by four downsampling blocks, each consisting of ResidualConvBlocks and MaxPool2d. Each block:
     - Doubles the number of channels: 64→128→256→512→1024
     - Halves the spatial dimensions: 64→32→16→8→4
- **Bottleneck:** Applies average pooling to reduce spatial size to 1×1, followed by fully connected layers. A transpose convolution then upsamples the output back to 4×4 with 1024 channels.
- **Decoder:**
  Consists of four upsampling blocks, each incorporating ResidualConvBlocks and skip connections from the corresponding encoder layers. Each block:

[1]`https://github.com/byrkbrk/conditional-ddpm`

  - Halves the number of channels: 1024→512→256→128→64
  - Doubles the spatial dimensions: 4→8→16→32→64

- **Output Layer:** A final 1×1 convolution reduces the output to 3 channels, followed by GroupNorm and GELU activation.

```python
class ContextUnet(nn.Module):
    def __init__(self,in_channels,height,width,n_feat,n_cfeat,n_downs):
        super(ContextUnet, self).__init__()
        self.in_channels = in_channels
        self.height = height
        self.width = width
        self.n_feat = n_feat
        self.n_cfeat = n_cfeat
        self.n_downs = n_downs

        # Define initial convolution
        self.init_conv = ResidualConvBlock(in_channels, n_feat, True)

        # Define downward unet blocks
        self.down_blocks = nn.ModuleList()
        for i in range(n_downs):
            self.down_blocks.append(UNetDown(2**i*n_feat, 2**(i+1)*n_feat))

        # Define the bottelneck layer
        self.to_vec = nn.Sequential(
            nn.AvgPool2d((height//2**len(self.down_blocks),\
                          width//2**len(self.down_blocks))),
            nn.GELU())
        self.up0 = nn.Sequential(
            nn.ConvTranspose2d(
                2**n_downs*n_feat,
                2**n_downs*n_feat,
                (height//2**len(self.down_blocks),\
                 width//2**len(self.down_blocks))),
            nn.GroupNorm(8, 2**n_downs*n_feat),
            nn.GELU()
        )

        # Define upward unet blocks
        self.up_blocks = nn.ModuleList()
        for i in range(n_downs, 0, -1):
            self.up_blocks.append(UNetUp(2**(i+1)*n_feat, 2**(i-1)*n_feat))

        # Define final convolutional layer
        self.final_conv = nn.Sequential(
            nn.Conv2d(2*n_feat, n_feat, 3, 1, 1),
42             nn.GroupNorm(8, n_feat),
43             nn.GELU(),
44             nn.Conv2d(n_feat, in_channels, 1, 1)
45         )
46 
47         # Define time & context embedding blocks
48         self.timeembs = nn.ModuleList([\
49              EmbedFC(1, 2**i*n_feat) for i in range(n_downs, 0, -1)\
50              ])
51         self.contextembs = nn.ModuleList([\
52              EmbedFC(n_cfeat, 2**i*n_feat) for i in range(n_downs, 0, -1)\
53              ])
54 
55     def forward(self, x, t, c):
56         x = self.init_conv(x)
57         downs = []
58         for i, down_block in enumerate(self.down_blocks):
59             if i == 0: downs.append(down_block(x))
60             else: downs.append(down_block(downs[-1]))
61         up = self.up0(self.to_vec(downs[-1]))
62         for up_block, down, contextemb, timeemb in zip(\
63             self.up_blocks,
64             downs[::-1],
65             self.contextembs,
66             self.timeembs):
67             up = up_block(up*contextemb(c) + timeemb(t), down)
68         return self.final_conv(torch.cat([up, x], axis=1))
```

The original parameter settings in Burak Bayrak's code are as follows:

Dataset	**in_channels**	**height × width**	**n_feat**	**n_cfeat**	**n_downs**
MNIST	1	28×28	64	10	2
Sprite	3	16×16	64	5	2
CIFAR-10	3	32×32	64	10	4

For the iCLEVR dataset, which includes 24 object classes, I set `n_cfeat` to 24. Additionally, since the task is more complex, I increased `n_downs` to 4.

```
nn_model = ContextUnet(in_channels=3, height=64, width=64,
                       n_feat=64, n_cfeat=24, n_downs=4)
```

2.2 Time Embedding

Time embeddings are implemented within the `ContextUnet` class, as described above. Specifically, they are defined in line 48:

```
self.timeembs = nn.ModuleList([\
             EmbedFC(1, 2**i*n_feat) for i in range(n_downs, 0, -1)])
```

This line constructs a list of fully connected embedding layers that encode the scalar timestep t into high-dimensional vectors. Each layer produces an embedding with dimensions increasing exponentially, and these embeddings are added to the corresponding decoder blocks during the forward pass.
In the `forward` method, the time embeddings are integrated as follows (line 67):

```
up = up_block(up * contextemb(c) + timeemb(t), down)
```

This operation combines the time embedding with the intermediate feature map before passing it through the upsampling block.

2.3 Cosine Noise Schedule

The original implementation uses a linear noise scheduler, where noise is added at a constant rate over the diffusion steps. In this work, I replaced it with a cosine noise scheduler, which introduces noise more gradually at the beginning and end of the diffusion process, and more rapidly in the middle. This scheduling strategy often leads to more stable training and improved output quality. The cosine schedule can be mathematically expressed as:

$$\beta_t = \beta_{\text{end}} + 0.5 \cdot (\beta_{\text{start}} - \beta_{\text{end}}) \cdot \left(1 + \cos\left(\pi \cdot \frac{t}{T}\right)\right) \tag{1}$$

,where the initial noise is small, while the final noise is large.
The implementation is as follows:

```
def get_cosine_noise_schedule(self, timesteps, beta1, beta2, device):
    t = torch.linspace(0, 1, timesteps+1, device=device)
    b_t = beta2 + 0.5 * (beta1 - beta2) * (1 + torch.cos(t * torch.pi ))
    a_t = 1 - b_t
    ab_t = torch.cumprod(a_t, dim=0)
    return a_t, b_t, ab_t
```

2.4 Training

The `train` function trains a conditional diffusion model using a cosine noise schedule. It manages data loading, noise perturbation, loss computation, parameter optimization, and learning rate scheduling. A detailed explanation of the key steps is as follows:

- **Line 3:** Computes the cosine noise schedule, which determines how noise is added over the diffusion timesteps.
- **Line 11:** Iterates over the training data; `x` represents the input images and `c` represents the class labels.
- **Lines 16—18:** Performs the forward diffusion step by adding Gaussian noise to the images at randomly sampled timesteps.

- **Line 21:** The model predicts the added noise, conditioned on both the timestep and the context label.
- **Lines 24—29:** Calculates the mean squared error (MSE) between the predicted and true noise, and updates the model parameters using backpropagation.

```
1  def train(self, batch_size, n_epoch, lr, timesteps, beta1, beta2, device):
2      self.nn_model.train()
3      _ , _, ab_t = self.get_cosine_noise_schedule(timesteps, beta1, beta2)
4      dataset = IclevrDataset(file_root="../file", data_root="../iclevr")
5      dataloader = self.initialize_dataloader(dataset, batch_size)
6      optim = self.initialize_optimizer(self.nn_model, lr)
7      scheduler = self.initialize_scheduler(optim)
8  
9      for epoch in range(n_epoch):
10         ave_loss = 0
11         for x, c in tqdm(dataloader, mininterval=2, desc=f"Epoch {epoch}"):
12             x = x.to(device)
13             c = c.to(device)
14 
15             # perturb data
16             noise = torch.randn_like(x)
17             t = torch.randint(1, timesteps + 1, (x.shape[0], )).to(device)
18             x_pert = self.perturb_input(x, t, noise, ab_t)
19 
20             # predict noise
21             pred_noise = self.nn_model(x_pert, t / timesteps, c=c)
22 
23             # obtain loss
24             loss = torch.nn.functional.mse_loss(pred_noise, noise)
25 
26             # update params
27             optim.zero_grad()
28             loss.backward()
29             optim.step()
30 
31             ave_loss += loss.item()/len(dataloader)
32         scheduler.step()
```

2.5 Sampling

The `sample_ddpm` function generates samples from a trained conditional diffusion model by starting with pure noise and iteratively denoising it. The key steps are as follows:

- **Line 2:** Computes the cosine noise schedule for the diffusion process.

- **Line 4:** Randomly samples the initial pure standard Gaussian noise, and no noise is added to the final image.
- **Line 14:** For each timestep (iteratively from T down to 1), the model predicts the noise component that was added to the image at the current timestep.
- **Line 19:** Calls the `denoise_add_noise` function to remove the predicted noise and optionally add Gaussian noise to prevent collapse.

```
 1 def sample_ddpm(self, n_samples, context, timesteps, beta1, beta2):
 2     a_t, b_t, ab_t = self.get_cosine_noise_schedule(timesteps,beta1,beta2)
 3     self.nn_model.eval()
 4     samples = torch.randn(n_samples, self.nn_model.in_channels,
 5                           self.nn_model.height, self.nn_model.width,
 6                           device=self.device)
 7     intermediate_samples = [samples.detach().cpu()]
 8     t_steps = [timesteps]
 9     for t in range(timesteps, 0, -1):
10         print(f"Sampling timestep {t}", end="\r")
11         if t % 50 == 0: print(f"Sampling timestep {t}")
12 
13         z = torch.randn_like(samples) if t > 1 else 0
14         pred_noise = self.nn_model(samples,
15                                    torch.tensor([t/timesteps],
16                                    device=self.device)
17                                    [:, None, None, None],
18                                    context)
19         samples=self.denoise_add_noise(samples,t,pred_noise,a_t,b_t,ab_t,z)
20 
21     return intermediate_samples[-1], intermediate_samples, t_steps
```

The `denoise_add_noise` function implements a key part of the DDPM sampling procedure, following the equation:

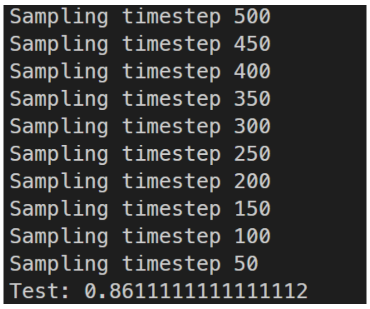
$$x_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}} \cdot \epsilon_\theta(x_t, t)\right) + \sigma_t z \tag{2}$$

```
def denoise_add_noise(self, x, t, pred_noise, a_t, b_t, ab_t, z):
    noise = b_t.sqrt()[t]*z
    denoised_x=(x-pred_noise*((1-a_t[t])/(1-ab_t[t]).sqrt()))/a_t[t].sqrt()
    return denoised_x + noise
```

3 Results and Discussion

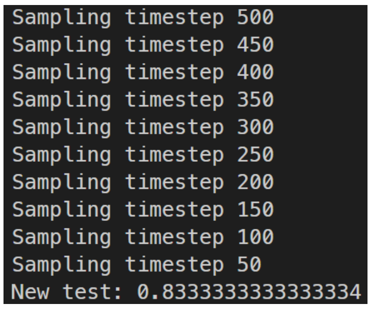
The following images show the results of my DDPM model, trained over 270 epochs with an initial learning rate of 1e-3, using a sampling timestep of 500.

3.1 Accuracy



```
Sampling timestep 500
Sampling timestep 450
Sampling timestep 400
Sampling timestep 350
Sampling timestep 300
Sampling timestep 250
Sampling timestep 200
Sampling timestep 150
Sampling timestep 100
Sampling timestep 50
Test: 0.8611111111111112
```

(a) test



```
Sampling timestep 500
Sampling timestep 450
Sampling timestep 400
Sampling timestep 350
Sampling timestep 300
Sampling timestep 250
Sampling timestep 200
Sampling timestep 150
Sampling timestep 100
Sampling timestep 50
New test: 0.8333333333333334
```

(b) new test

3.2 Synthetic Image Grids

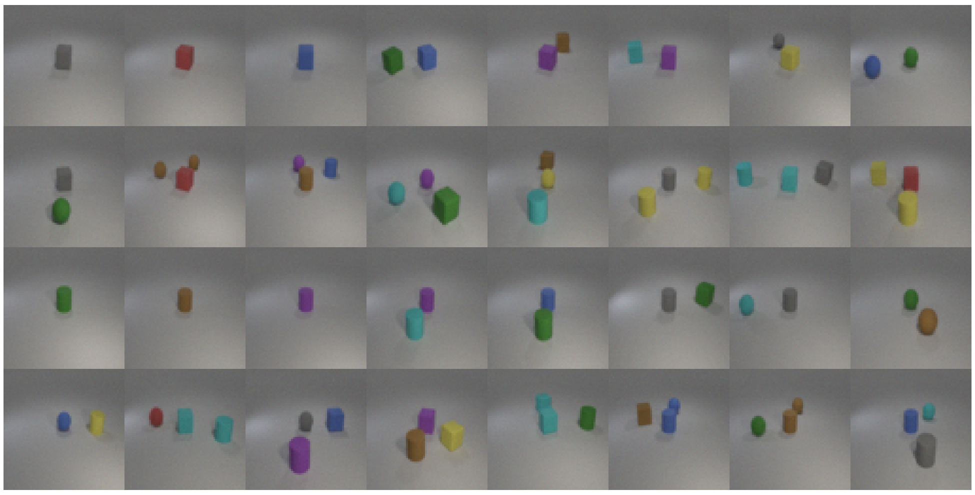


Figure 2: Result on test.json

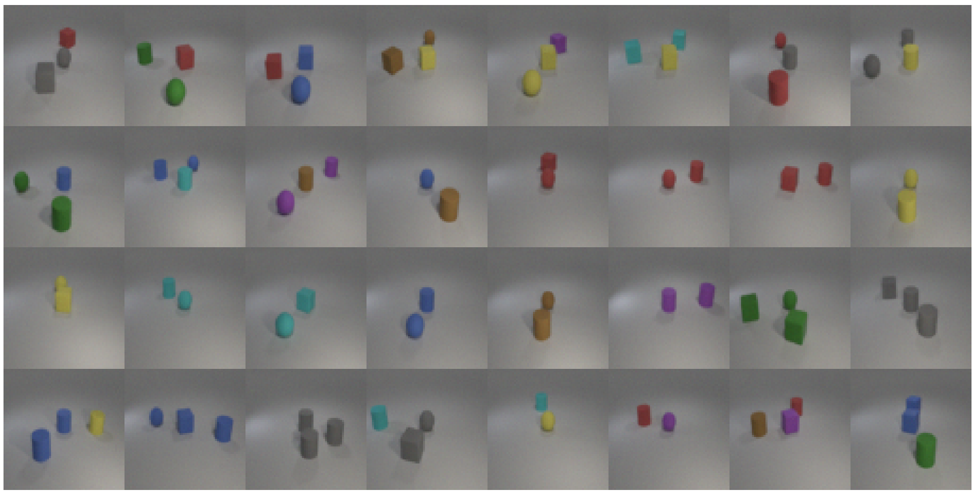


Figure 3: Result on new_test.json

3.3 Denoising Process Image

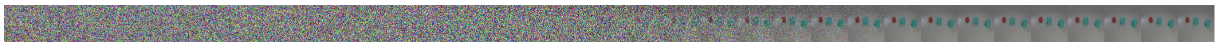


Figure 4: Denoising process of label = ["red sphere", "cyan cylinder", "cyan cube"]

3.4 Discussion of your extra implementations or experiments

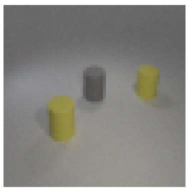
3.4.1 Cosine Noise Scheduler v.s. Linear Noise Scheduler

I initially used the linear noise scheduler provided by the original code. However, the results showed that out of 500 sampling steps, around 400 steps consisted of pure noise. This suggests that the linear noise scheduler may be inefficient for this task, as it introduces a large number of uninformative steps.

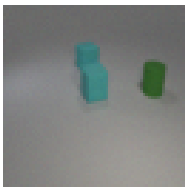
3.4.2 Findings from the Results

I observed that the DDPM performs well when the label contains one or two objects. However, when the label includes three objects, the model tends to generate incorrect outputs. For example:

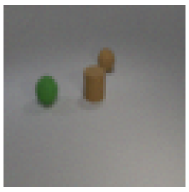
1. test - 13
label: [”gray cylinder”, ”yellow cylinder”, ”purple cube”]
result: no purple cube



2. test - 28
label: [”cyan cube”, ”green cylinder”, ”blue cube”]
result: no blue cube



3. test - 30
label: [”green sphere”, ”red cylinder”, ”brown sphere”]
result: no red cylinder



3.4.3 Findings from the Training Process

I also saved the generated results during the training phase and observed that, before the model converged—for instance, around epoch 94—it occasionally produced more objects than indicated by the label. For example, it might generate four objects even when the label specifies only three. This behavior suggests that the model has not yet matured as a reliable conditional generator.