LAB 6

B1126006 郭兆揚

Introduction

This lab focuses on implementing a Conditional Denoising Diffusion Probabilistic Model (DDPM). Given a set of multi-label conditions provided in test.json and new_test.json, the model is required to generate images containing the specified objects, as well as visualize the denoising process. The synthesized images are evaluated using a pretrained classifier provided by the TA to assess their quality and alignment with the given labels.

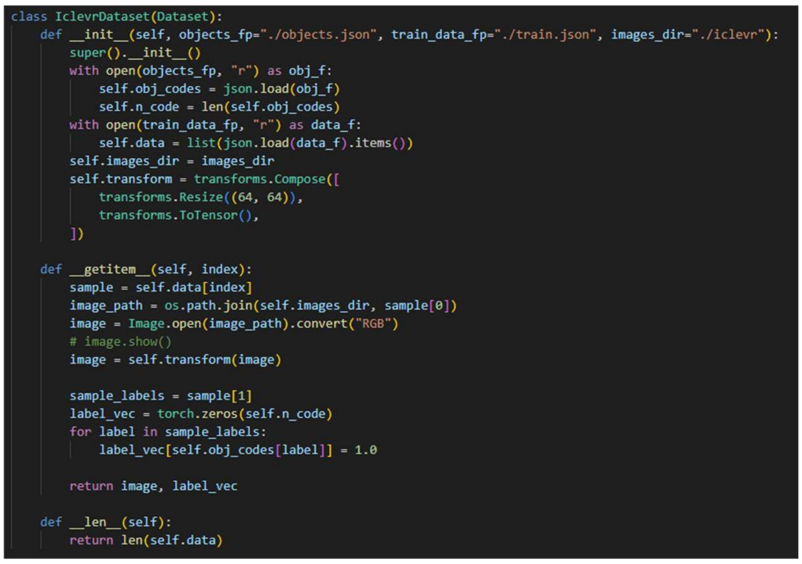
In this work, I referred to the original DDPM paper and implemented three variants of the UNet architecture, exploring different designs for embedding strategies, normalization methods, and activation functions. All models were evaluated using the provided pretrained ResNet18-based evaluator. Among the designs, the best-performing model — which closely follows the original DDPM architecture by applying time embeddings at every layer and using group normalization — achieved the highest classification accuracy (test: 0.86; new_test: 0.87).

Implementation details

1. Data Preprocessing

To prepare the dataset, I utilized the iclevr.py script and the Python Imaging Library (PIL) to load and process images. Each image was converted into a 64×64 RGB tensor for input into the model.

For label processing, I used the objects.json file provided by the TA, which defines a dictionary mapping object names to class indices. Each label corresponding to an image was transformed into a multi-label one-hot encoded vector, which was later used as the condition input to the DDPM model.



```python
class IclevrDataset(Dataset):
    def __init__(self, objects_fp="./objects.json", train_data_fp="./train.json", images_dir="./iclevr"):
        super().__init__()
        with open(objects_fp, "r") as obj_f:
            self.obj_codes = json.load(obj_f)
            self.n_code = len(self.obj_codes)
        with open(train_data_fp, "r") as data_f:
            self.data = list(json.load(data_f).items())
        self.images_dir = images_dir
        self.transform = transforms.Compose([
            transforms.Resize((64, 64)),
            transforms.ToTensor(),
        ])

    def __getitem__(self, index):
        sample = self.data[index]
        image_path = os.path.join(self.images_dir, sample[0])
        image = Image.open(image_path).convert("RGB")
        # image.show()
        image = self.transform(image)

        sample_labels = sample[1]
        label_vec = torch.zeros(self.n_code)
        for label in sample_labels:
            label_vec[self.obj_codes[label]] = 1.0

        return image, label_vec

    def __len__(self):
        return len(self.data)
```

iclevr dataset

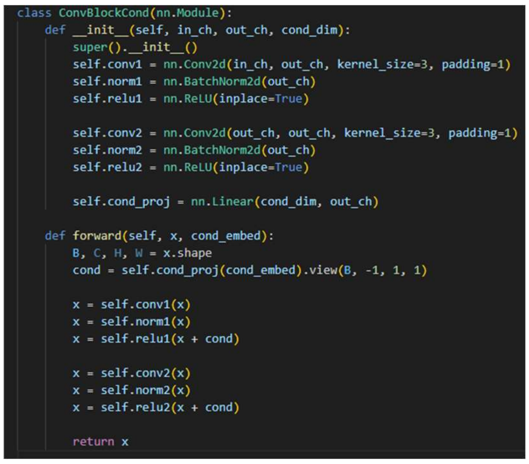
2. Model Design

To explore different architectural choices for conditional DDPMs, I implemented three UNet variants. Below, I describe each design in detail.

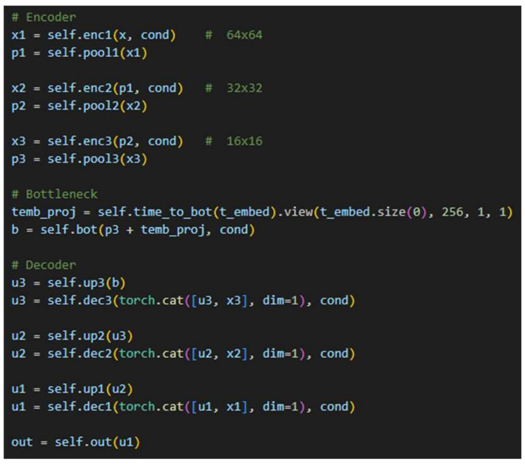
Version 1: Simplified UNet (Baseline)

Considering the relatively moderate complexity of this lab task, I first implemented a simplified UNet architecture as a baseline to test the conditional image generation task without introducing excessive structural complexity. This version is adapted from the standard UNet architecture, with extensions to support time and condition embeddings.

The core building block is ConvBlockCond, which consists of two convolutional layers, each followed by Batch Normalization and a ReLU activation. Downsampling is achieved using a combination of ConvBlockCond modules and max-pooling layers, while upsampling is done using transpose convolutions followed by additional ConvBlockCond modules. Finally, a 1×1 convolution maps the output to a 3-channel RGB image. See figure below for the code structure of ConvBlockCond and the overall UNet.



```python
class ConvBlockCond(nn.Module):
    def __init__(self, in_ch, out_ch, cond_dim):
        super().__init__()
        self.conv1 = nn.Conv2d(in_ch, out_ch, kernel_size=3, padding=1)
        self.norm1 = nn.BatchNorm2d(out_ch)
        self.relu1 = nn.ReLU(inplace=True)

        self.conv2 = nn.Conv2d(out_ch, out_ch, kernel_size=3, padding=1)
        self.norm2 = nn.BatchNorm2d(out_ch)
        self.relu2 = nn.ReLU(inplace=True)

        self.cond_proj = nn.Linear(cond_dim, out_ch)

    def forward(self, x, cond_embed):
        B, C, H, W = x.shape
        cond = self.cond_proj(cond_embed).view(B, -1, 1, 1)

        x = self.conv1(x)
        x = self.norm1(x)
        x = self.relu1(x + cond)

        x = self.conv2(x)
        x = self.norm2(x)
        x = self.relu2(x + cond)

        return x
```



```python
# Encoder
x1 = self.enc1(x, cond)    #  64x64
p1 = self.pool1(x1)

x2 = self.enc2(p1, cond)   #  32x32
p2 = self.pool2(x2)

x3 = self.enc3(p2, cond)   #  16x16
p3 = self.pool3(x3)

# Bottleneck
temb_proj = self.time_to_bot(t_embed).view(t_embed.size(0), 256, 1, 1)
b = self.bot(p3 + temb_proj, cond)

# Decoder
u3 = self.up3(b)
u3 = self.dec3(torch.cat([u3, x3], dim=1), cond)

u2 = self.up2(u3)
u2 = self.dec2(torch.cat([u2, x2], dim=1), cond)

u1 = self.up1(u2)
u1 = self.dec1(torch.cat([u1, x1], dim=1), cond)

out = self.out(u1)
```

version 1: Structure of ConvBlockCond and Unet

Condition Embedding in v1: The input condition vector (a multi-label one-hot vector) is projected via a linear layer to match the output channel dimension of the convolutional block. This projected condition vector is treated as a bias-like term and is added to the activations. To avoid interfering with Batch Normalization's internal statistics (mean and variance computed before affine transformation), the condition embedding is injected *after* normalization. This placement helps prevent the condition vector from skewing the normalized distribution of activations.

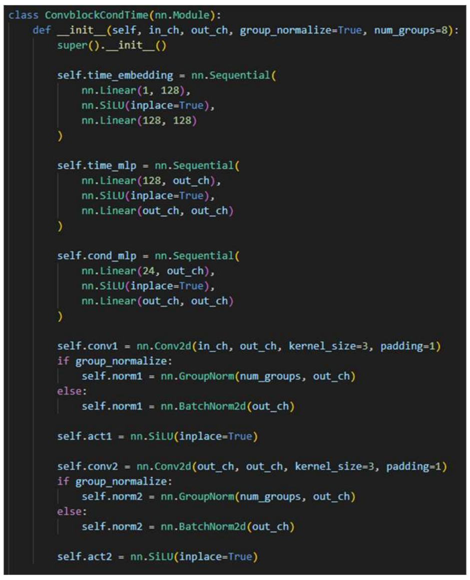
Time Embedding in v1: The time step t is first projected into a higher-dimensional space using a two-layer MLP with ReLU activation. This helps encode more semantic and nonlinear information about the timestep. However, since the value range of t (typically from 0 to T, e.g., 0 to 1000) is significantly different from the binary label vectors (0 or 1), I chose to apply the time embedding only in the bottleneck layer in this version. This design avoids potential interference between the large variance of the timestep embedding and the condition embedding, and also serves to isolate the temporal information where it may be most useful.

Although Version 1 achieved reasonable classification accuracy at 500 training epochs (test: 0.71; new_test: 0.82), I observed that extending training to 1000 epochs did not lead to significant improvements (test: 0.78; new_test: 0.79). The model exhibited signs of convergence stagnation and still fell short of the full score benchmark set by the TA. In addition, the denoising process was visually unstable and inconsistent across runs.

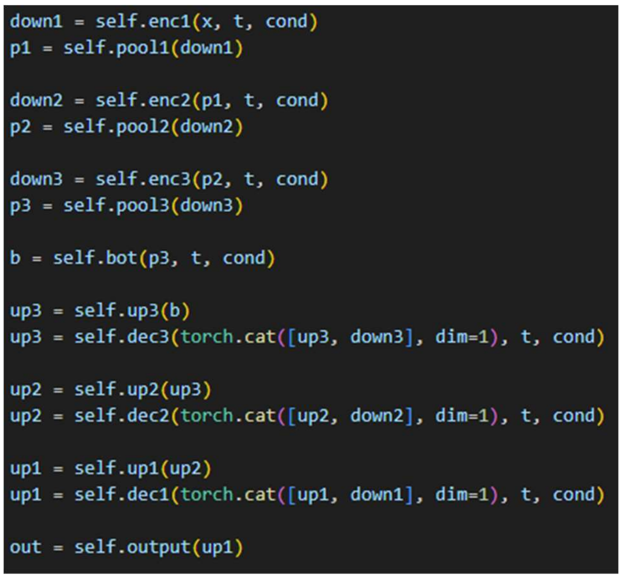
Based on these observations, I decided to revise the model architecture by incorporating design principles from the original DDPM paper, with the goal of improving both quantitative performance and qualitative sample stability.

Version 2: DDPM-Aligned UNet with Batch Normalization

In Version 2, I redesigned the model architecture to align more closely with the structural design proposed in the original DDPM paper. This version introduces a new building block, ConvBlockCondTime, which serves as the foundation for the full model implementation, UnetCondTime.



```python
class ConvblockCondTime(nn.Module):
    def __init__(self, in_ch, out_ch, group_normalize=True, num_groups=8):
        super().__init__()

        self.time_embedding = nn.Sequential(
            nn.Linear(1, 128),
            nn.SiLU(inplace=True),
            nn.Linear(128, 128)
        )

        self.time_mlp = nn.Sequential(
            nn.Linear(128, out_ch),
            nn.SiLU(inplace=True),
            nn.Linear(out_ch, out_ch)
        )

        self.cond_mlp = nn.Sequential(
            nn.Linear(24, out_ch),
            nn.SiLU(inplace=True),
            nn.Linear(out_ch, out_ch)
        )

        self.conv1 = nn.Conv2d(in_ch, out_ch, kernel_size=3, padding=1)
        if group_normalize:
            self.norm1 = nn.GroupNorm(num_groups, out_ch)
        else:
            self.norm1 = nn.BatchNorm2d(out_ch)

        self.act1 = nn.SiLU(inplace=True)

        self.conv2 = nn.Conv2d(out_ch, out_ch, kernel_size=3, padding=1)
        if group_normalize:
            self.norm2 = nn.GroupNorm(num_groups, out_ch)
        else:
            self.norm2 = nn.BatchNorm2d(out_ch)

        self.act2 = nn.SiLU(inplace=True)
```



```python
down1 = self.enc1(x, t, cond)
p1 = self.pool1(down1)

down2 = self.enc2(p1, t, cond)
p2 = self.pool2(down2)

down3 = self.enc3(p2, t, cond)
p3 = self.pool3(down3)

b = self.bot(p3, t, cond)

up3 = self.up3(b)
up3 = self.dec3(torch.cat([up3, down3], dim=1), t, cond)

up2 = self.up2(up3)
up2 = self.dec2(torch.cat([up2, down2], dim=1), t, cond)

up1 = self.up1(up2)
up1 = self.dec1(torch.cat([up1, down1], dim=1), t, cond)

out = self.output(up1)
```

version 2: Structure of ConvBlockCondTime and UnetCondTime

Key modifications in this version include:

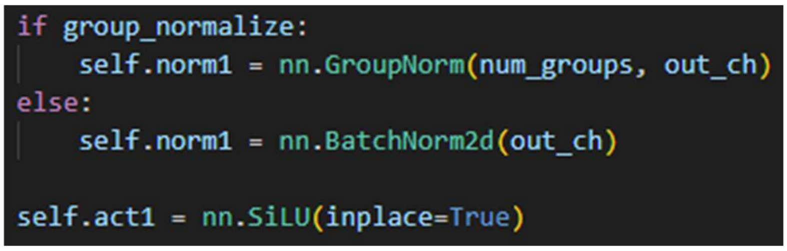
- Both the **timestep** and **condition vector** are passed through separate two-layer MLPs to obtain higher-dimensional embeddings with greater representational capacity.
- The **activation function** was changed from ReLU to **SiLU** (Sigmoid-weighted Linear Unit), as adopted in the original DDPM.
- The **time and condition embeddings** are injected into *every convolutional block*, rather than only at the bottleneck as in Version 1.
- The **embedding is applied before normalization**, with the intuition that this positioning would allow the embeddings to directly modulate feature distributions.

Although this version more closely follows the DDPM paper in terms of embedding design and activation function, I initially assumed that the normalization type would have a limited impact, and therefore retained **Batch Normalization** rather than switching to **Group Normalization**. However, the results showed that this version performed worse than the baseline in terms of classification accuracy (test: 0.60; new_test: 0.73). I speculate that this may be due to BatchNorm's sensitivity to added embeddings prior to normalization, which could disrupt its internal statistics and make training less stable.

Version 3: DDPM-Aligned UNet with Group Normalization

In Version 3, I addressed the compatibility issues between **Batch Normalization** and the injected **time/condition embeddings** encountered in Version 2. To resolve this, I replaced all BatchNorm layers in the model with **Group Normalization**.

GroupNorm is independent of batch size and provides more stable behavior when dealing with conditioning information, as it normalizes across groups of channels instead of across the batch dimension. In this version, I retained the UnetCondTime architecture, where both time and condition embeddings are applied to **every convolutional block before normalization**. The only change was replacing all normalization layers with GroupNorm. Different group sizes (e.g., 8, 16, 32, 64) were assigned based on the number of channels in each layer, ensuring proper adaptation at different resolutions.



```
if group_normalize:
    self.norm1 = nn.GroupNorm(num_groups, out_ch)
else:
    self.norm1 = nn.BatchNorm2d(out_ch)

self.act1 = nn.SiLU(inplace=True)
```

version 3: using Group Normalization to replace Batch Normalization

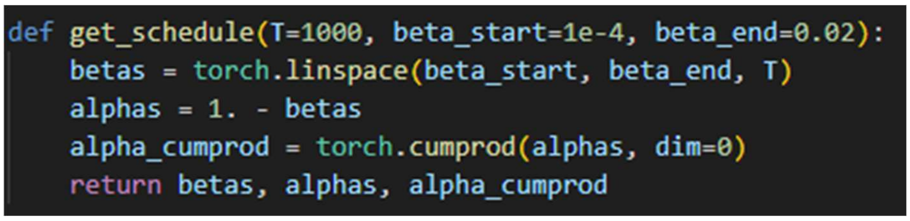
This adjustment significantly improved training stability and the visual quality of the generated images. Compared to Versions 1 and 2, Version 3 achieved **the highest classification accuracy** (test: 0.86; new_test: 0.87), reaching the maximum score threshold specified by the TA. In addition, the denoising process became noticeably more stable and consistent. These results suggest that **Group Normalization is a more suitable normalization strategy** for DDPMs with time and condition embeddings.

3. Noise Schedule

I adopted a **linear noise schedule**, where the noise variance βt\beta_tβt increases linearly from 1e-4 to 0.02 over T=1000T = 1000T=1000 timesteps. The noise scaling factors are computed based on:

$$\alpha_t = 1 - \beta_t \ , \quad \bar{\alpha} = \prod_{s=0}^{t} \alpha_s$$

This defines the progressive corruption of images over time. The corresponding implementation is shown below:



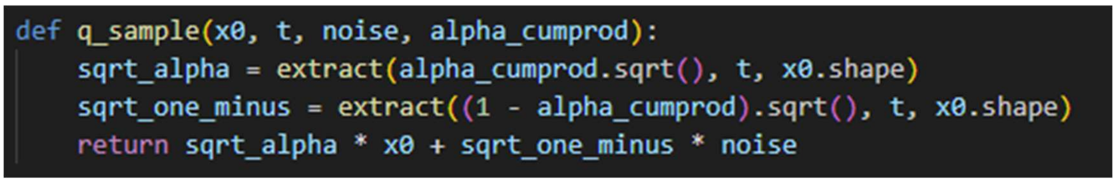
```
def get_schedule(T=1000, beta_start=1e-4, beta_end=0.02):
    betas = torch.linspace(beta_start, beta_end, T)
    alphas = 1. - betas
    alpha_cumprod = torch.cumprod(alphas, dim=0)
    return betas, alphas, alpha_cumprod
```

Noise schedule implementation code

4. Forward Process

The forward process adds Gaussian noise to the original image x_0 to obtain a noisy version x_t at an arbitrary timestep t, using the following formula:

$$x_t = \sqrt{\bar{\alpha}} * x_0 + \sqrt{1 - \bar{\alpha}} * \epsilon$$



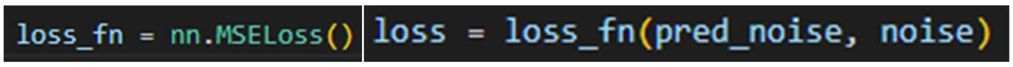
```
def q_sample(x0, t, noise, alpha_cumprod):
    sqrt_alpha = extract(alpha_cumprod.sqrt(), t, x0.shape)
    sqrt_one_minus = extract((1 - alpha_cumprod).sqrt(), t, x0.shape)
    return sqrt_alpha * x0 + sqrt_one_minus * noise
```

Forward process implementation code

5. Loss Function

The model is trained to predict the added noise ϵ, and the loss function used is **mean squared error (MSE)**:

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



```
loss_fn = nn.MSELoss()
```

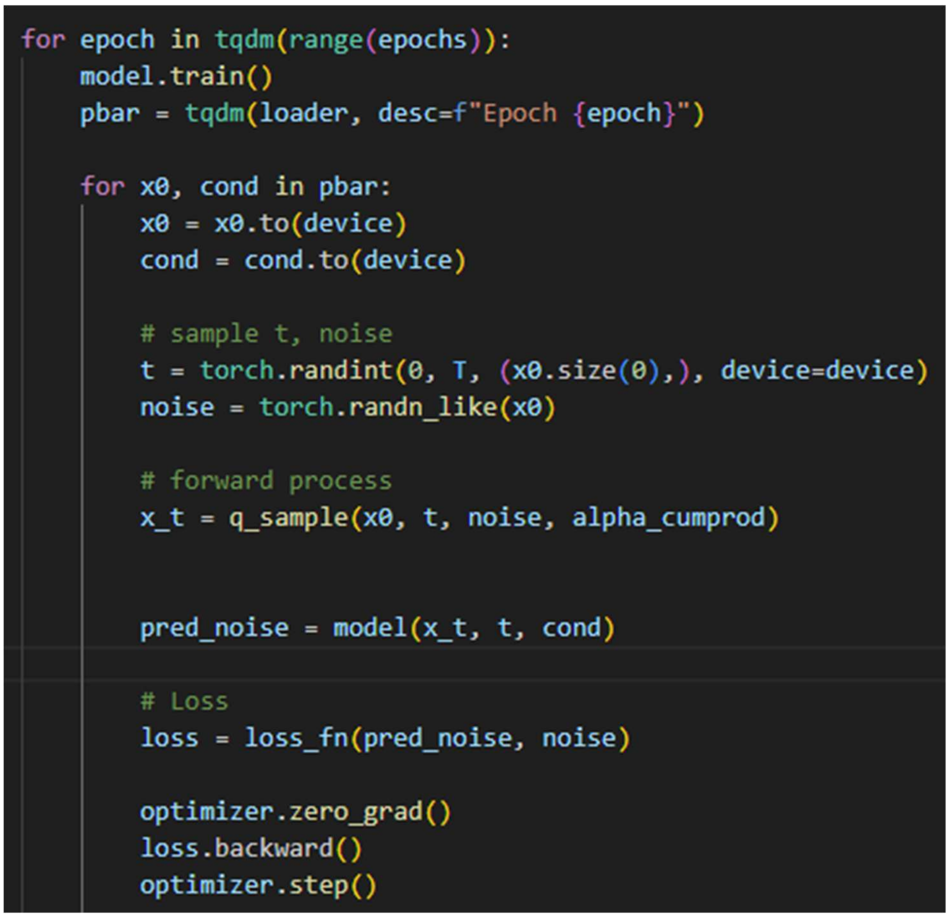
```
loss = loss_fn(pred_noise, noise)
```

Loss function implementation code

6. Whole Training Procedure

The full training pipeline proceeds as follows:

1. Sample a clean image x_0 and the corresponding condition label vector $cond$
2. Uniformly sample a timestep t and Gaussian noise ϵ
3. Use x_0, t, ϵ and the precomputed noise schedule to generate x_t via the forward process
4. Input x_t, t, $cond$ into the model to predict the noise to be removed
5. Compute the loss using MSE between the predicted and true noise
6. Perform backpropagation and update the model parameters



```python
for epoch in tqdm(range(epochs)):
    model.train()
    pbar = tqdm(loader, desc=f"Epoch {epoch}")

    for x0, cond in pbar:
        x0 = x0.to(device)
        cond = cond.to(device)

        # sample t, noise
        t = torch.randint(0, T, (x0.size(0),), device=device)
        noise = torch.randn_like(x0)

        # forward process
        x_t = q_sample(x0, t, noise, alpha_cumprod)


        pred_noise = model(x_t, t, cond)

        # Loss
        loss = loss_fn(pred_noise, noise)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

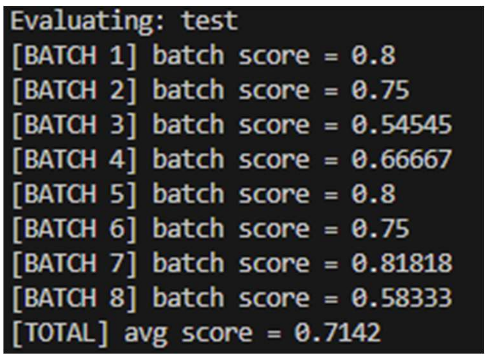
Training Procedure

Results and Discussion

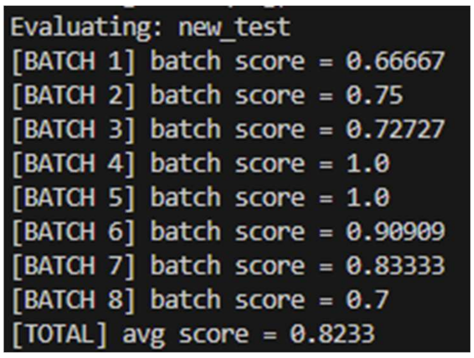
To quantitatively compare the performance of different model designs, I used the pretrained ResNet18 evaluator provided by the TA to assess the classification accuracy of generated images. The evaluation was conducted on both the test.json and new_test.json label sets. The classification results for each model version are shown below:

Model Version	**test**	**new_test**
Version 1 (Simplified UNet, 500ep)	0.71	0.82
Version 1 (1000ep)	0.78	0.78
Version 2 (DDPM aligned + BN, 500ep)	0.60	0.73
Version 3 (DDPM aligned + GN, 500ep)	**0.86**	**0.87**

classification accuracy of each model version

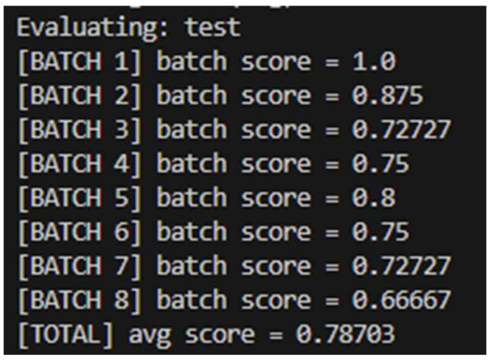


```
Evaluating: test
[BATCH 1] batch score = 0.8
[BATCH 2] batch score = 0.75
[BATCH 3] batch score = 0.54545
[BATCH 4] batch score = 0.66667
[BATCH 5] batch score = 0.8
[BATCH 6] batch score = 0.75
[BATCH 7] batch score = 0.81818
[BATCH 8] batch score = 0.58333
[TOTAL] avg score = 0.7142
```

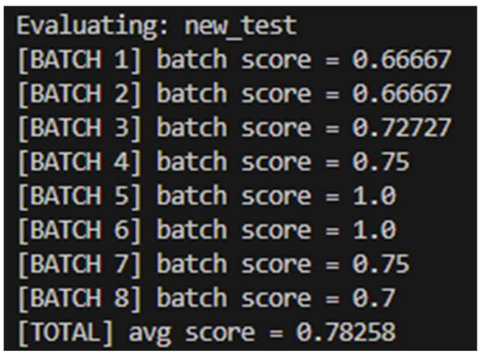


```
Evaluating: new_test
[BATCH 1] batch score = 0.66667
[BATCH 2] batch score = 0.75
[BATCH 3] batch score = 0.72727
[BATCH 4] batch score = 1.0
[BATCH 5] batch score = 1.0
[BATCH 6] batch score = 0.90909
[BATCH 7] batch score = 0.83333
[BATCH 8] batch score = 0.7
[TOTAL] avg score = 0.8233
```

classification accuracy of v1 (500 epochs)

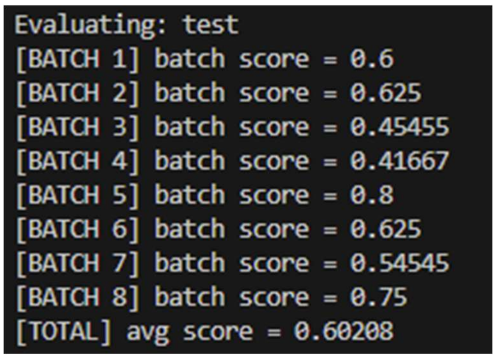


```
Evaluating: test
[BATCH 1] batch score = 1.0
[BATCH 2] batch score = 0.875
[BATCH 3] batch score = 0.72727
[BATCH 4] batch score = 0.75
[BATCH 5] batch score = 0.8
[BATCH 6] batch score = 0.75
[BATCH 7] batch score = 0.72727
[BATCH 8] batch score = 0.66667
[TOTAL] avg score = 0.78703
```

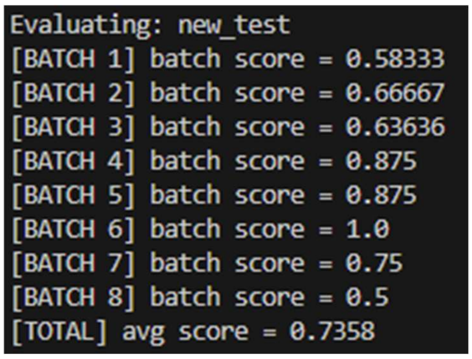


```
Evaluating: new_test
[BATCH 1] batch score = 0.66667
[BATCH 2] batch score = 0.66667
[BATCH 3] batch score = 0.72727
[BATCH 4] batch score = 0.75
[BATCH 5] batch score = 1.0
[BATCH 6] batch score = 1.0
[BATCH 7] batch score = 0.75
[BATCH 8] batch score = 0.7
[TOTAL] avg score = 0.78258
```

classification accuracy of v1 (1000 epochs)

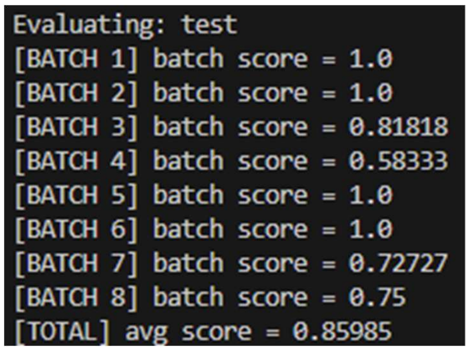


```
Evaluating: test
[BATCH 1] batch score = 0.6
[BATCH 2] batch score = 0.625
[BATCH 3] batch score = 0.45455
[BATCH 4] batch score = 0.41667
[BATCH 5] batch score = 0.8
[BATCH 6] batch score = 0.625
[BATCH 7] batch score = 0.54545
[BATCH 8] batch score = 0.75
[TOTAL] avg score = 0.60208
```

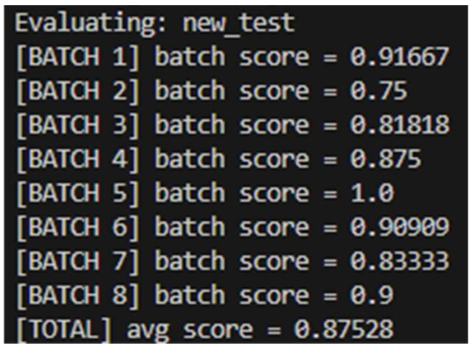


```
Evaluating: new_test
[BATCH 1] batch score = 0.58333
[BATCH 2] batch score = 0.66667
[BATCH 3] batch score = 0.63636
[BATCH 4] batch score = 0.875
[BATCH 5] batch score = 0.875
[BATCH 6] batch score = 1.0
[BATCH 7] batch score = 0.75
[BATCH 8] batch score = 0.5
[TOTAL] avg score = 0.7358
```

classification accuracy of v2



```
Evaluating: test
[BATCH 1] batch score = 1.0
[BATCH 2] batch score = 1.0
[BATCH 3] batch score = 0.81818
[BATCH 4] batch score = 0.58333
[BATCH 5] batch score = 1.0
[BATCH 6] batch score = 1.0
[BATCH 7] batch score = 0.72727
[BATCH 8] batch score = 0.75
[TOTAL] avg score = 0.85985
```



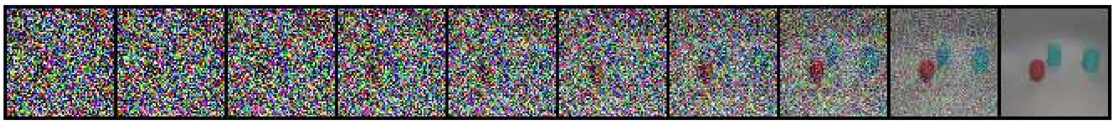
```
Evaluating: new_test
[BATCH 1] batch score = 0.91667
[BATCH 2] batch score = 0.75
[BATCH 3] batch score = 0.81818
[BATCH 4] batch score = 0.875
[BATCH 5] batch score = 1.0
[BATCH 6] batch score = 0.90909
[BATCH 7] batch score = 0.83333
[BATCH 8] batch score = 0.9
[TOTAL] avg score = 0.87528
```

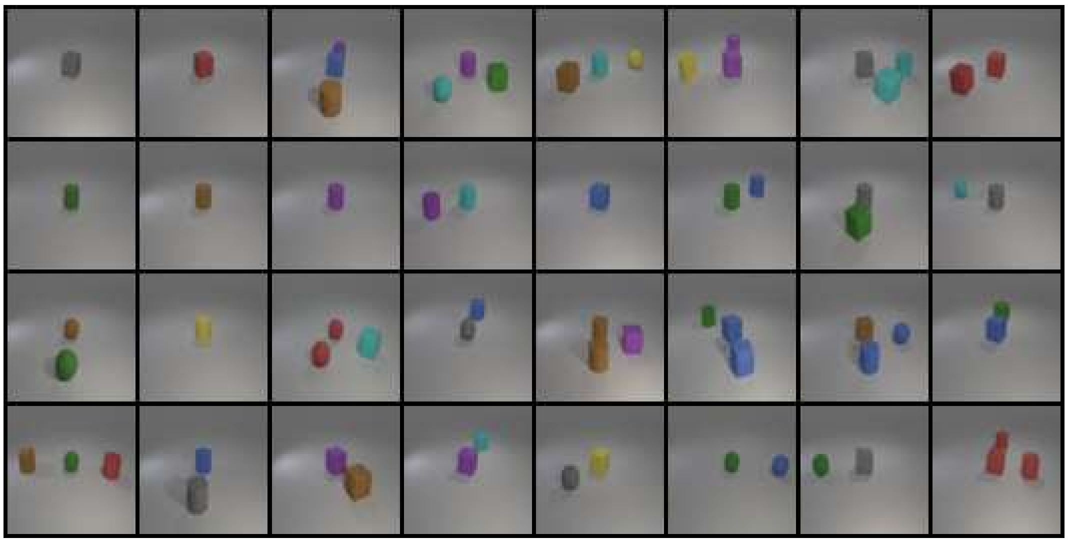
classification accuracy of v3

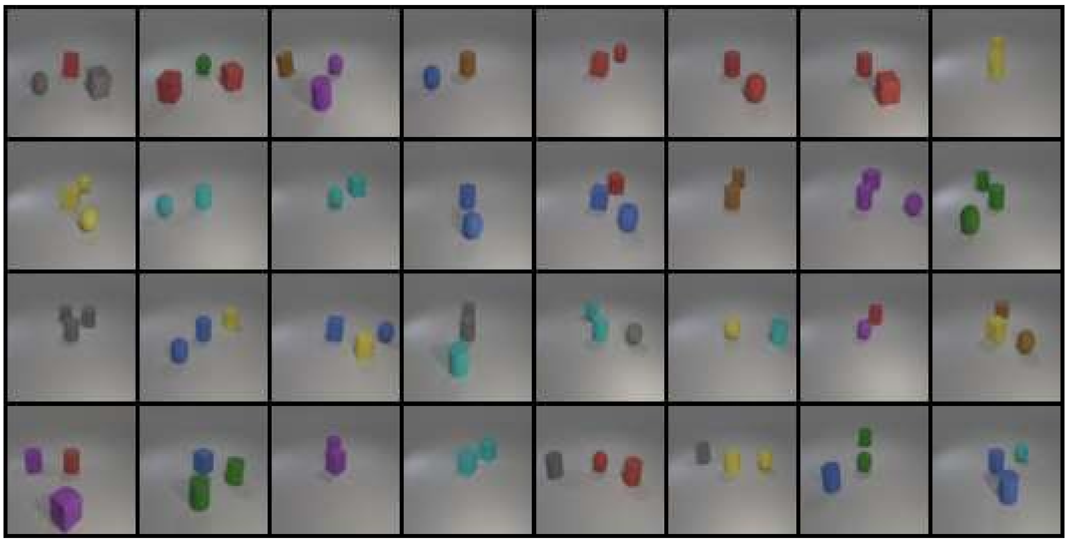
Version 1, as the baseline, achieved moderate classification accuracy but plateaued after 500 epochs, with no significant improvements observed even after extending training to 1000 epochs. Although Version 2 incorporated full-layer time and condition embeddings and replaced ReLU with SiLU, its continued use of Batch Normalization likely introduced instability. When embeddings are injected before normalization, BatchNorm may produce inconsistent internal statistics, leading to degraded performance.

Version 3 addressed this by replacing all BatchNorm layers with GroupNorm, which improved both training stability and classification accuracy, ultimately achieving the maximum score required. These findings indicate that Group Normalization is better suited for diffusion models with time and condition embeddings.

As Version 3 demonstrated the best overall performance, all subsequent visualizations, including denoising processes and image grids, are generated using this model.


Version 3: denoising process


Version 3: images grid of test


Version 3: images grid of new_test

Based on the results, two key architectural factors can be identified:

- **Embedding Depth:** Injecting time and condition embeddings at every convolutional block (as in Versions 2 and 3) is more effective than only embedding at the bottleneck (as in Version 1), leading to better conditioning representation.

- **Normalization Strategy:** Replacing Batch Normalization with Group Normalization (Version 3) significantly improves training stability and generation quality, especially when embeddings are added before normalization.

Overall, Version 3 aligns most closely with the original DDPM architecture and achieved the best performance in both classification accuracy and visual fidelity.

Conclusion

Through the progressive design and evaluation of three UNet-based conditional diffusion models, this study demonstrated the critical impact of architectural choices on both quantitative and qualitative performance. While the baseline model provided a reasonable starting point, further enhancements—such as full-layer time and condition embeddings, and the replacement of Batch Normalization with Group Normalization—led to significant improvements in training stability and classification accuracy.

Among all versions, Version 3 achieved the best results, attaining full classification scores on both test sets and producing stable, interpretable denoising trajectories. These findings underscore the importance of embedding placement and normalization strategies in the effective implementation of conditional denoising diffusion models.

Usage & Command

Below are the steps to operate the files, which can be followed to reproduce the best-performing model:

1. Using `train.py` to train the model:

```
python train.py ^
--saved-dir dir_name ^
--epochs 500 ^
--use-improv-unet ^
--GN
```

2. Using `generate.py` with a trained model to generate image directory

```
python generate.py ^
--model-path model_path ^
--output-dir images_dir_name ^
--use-improv-unet ^
--GN
```

3. Using `evaluate.py` to evaluating on image directory

```
python evaluate.py ^
--test-name test or new_test ^
--images-dir images_dir_name
```

4. Using `denoise_process.py` to make denoise process

```
python denoise_process.py ^
--model-path model_path
--use-improv-unet
--GN
```