國立陽明交通大學

NATIONAL YANG MING CHIAO TUNG UNIVERSITY

Deep Learning
Lab7: Let's Play DDPM

Name: 許子駿

Student ID: 311551166

Institute of Computer Science and Engineering

2023 年 5 月 30 日

Table of Contents

1 Introduction **2**

2 Experiment setups **3**

2.1 DDPM 3

2.1.1 UNet 4

2.1.2 Noise schedule 4

2.1.3 Loss function 5

2.2 Data loader 5

2.3 Hyperparameters 6

3 Experimental results **7**

4 Discussion **10**

Appendices **11**

A Code **12**

Chapter 1

Introduction

This task is to generate synthetic images according to multi-label conditions with conditional Denoising Diffusion Probabilistic Models (DDPM) (See Figures 1.1 and 1.2).

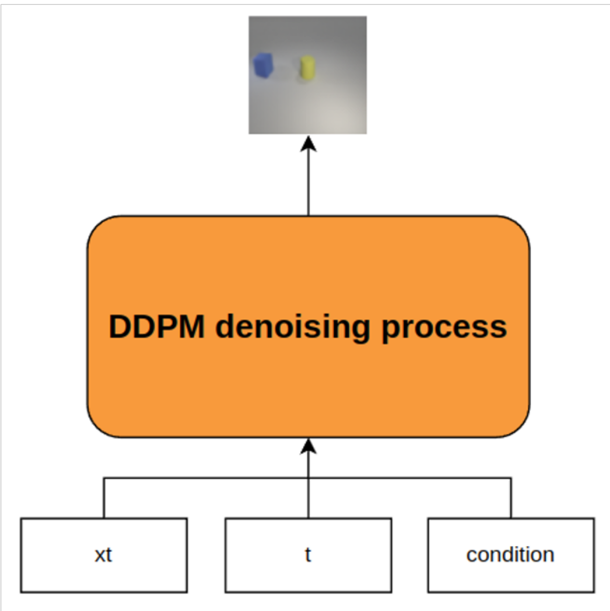

Figure 1.1: Illustration of DDPM process to synthesize images.

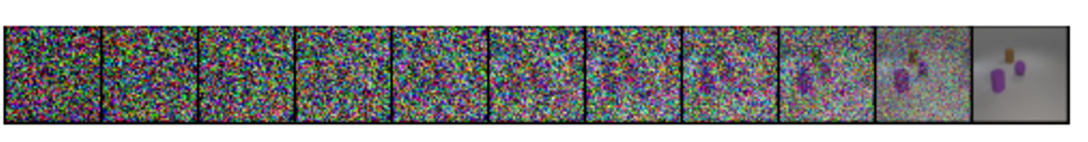

Figure 1.2: Example of DDPM denoising process.

Chapter 2

Experiment setups

Experiment setups are listed in this chapter including 3 parts, DDPM, data loader, and hyperparameters.

2.1 DDPM

The diffusion model is Denoising Diffusion Probabilistic Models[1]. The conditioning follows the method of Classifier-Free Diffusion Guidance[2].
The model infuses timestep embeddings t_e and context embeddings c_e with the U-Net activations at layer a_L via

$$a_{L+1} = c_e a_L + t_e$$

During training, c_e is randomly set to 0 with probability 0.1, and this makes model learn to do unconditional generation, i.e., $\psi(z_t)$ for noise z_t at timestep t, and also conditional generation, i.e., $\psi(z_t, c)$ for context c.
Additionally, a weight $w \geq 0$ is necessary, to guide the model to generate samples with the following equation

$$\hat{\epsilon}_t = (1 + w)\psi(z_t, c) - w\psi(z_t)$$

Increasing w produces images that are more typical but less diverse.
The DDPM details are listed in this section including 3 parts, UNet, noise schedule, and loss function.

[1] `https://arxiv.org/abs/2006.11239`

[2] `https://arxiv.org/abs/2207.12598`

2.1.1 UNet

The model uses asymmetric UNet, composed of 2 down-sampling layers and 3 up-sampling layers, and the final output is output of third up-sampling layer with input image concatenated. Timestep and context are embedded with 2 linear layers with one `GELU` activation layer in between. (See Listings 2.1, A.1, and A.2 for details).

```python
class ContextUnet(nn.Module):
    def forward(self,
                x: torch.FloatTensor, c: torch.FloatTensor, t: torch.FloatTensor,
                context_mask: torch.FloatTensor) -> torch.FloatTensor:
        # x: image, c: context, t: timestep.
        x = self.init_conv(x)
        down1 = self.down1(x)
        down2 = self.down2(down1)
        hiddenvec = self.to_vec(down2)

        context_mask = (-1 * (1 - context_mask)) # flip 0 <-> 1
        c = c * context_mask

        # embed context, time step
        cemb1 = self.contextembed1(c).view(-1, self.n_feat * 2, 1, 1)
        temb1 = self.timeembed1(t).view(-1, self.n_feat * 2, 1, 1)
        cemb2 = self.contextembed2(c).view(-1, self.n_feat, 1, 1)
        temb2 = self.timeembed2(t).view(-1, self.n_feat, 1, 1)

        up1 = self.up0(hiddenvec)
        up2 = self.up1(cemb1 * up1 + temb1, down2)
        up3 = self.up2(cemb2 * up2 + temb2, down1)
        out = self.out(torch.cat((up3, x), 1))
        return out
```

Listing 2.1: Python code of **ContextUnet** of DDPM (some code is omitted).

2.1.2 Noise schedule

Noise schedule follows equation

$$\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(\mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\epsilon_\theta(\mathbf{x}_t, t)\right) + \sigma_t \mathbf{z}$$

Listing 2.2 shows the noise schedule function calculates each part of above equation. During sampling, the pre-calculated values are used to sample images.

```python
def ddpm_schedules(beta1, beta2, T):
    assert beta1 < beta2 < 1.0, 'beta1 and beta2 must be in (0, 1)'

    beta_t = (beta2 - beta1) * torch.arange(0, T + 1, dtype=torch.float32) / T
    + beta1
    sqrt_beta_t = torch.sqrt(beta_t)
    alpha_t = 1 - beta_t
    log_alpha_t = torch.log(alpha_t)
    alphabar_t = torch.cumsum(log_alpha_t, dim=0).exp()

    sqrtab = torch.sqrt(alphabar_t)
    oneover_sqrta = 1 / torch.sqrt(alpha_t)

    sqrtmab = torch.sqrt(1 - alphabar_t)
    mab_over_sqrtmab_inv = (1 - alpha_t) / sqrtmab

    return {
        'alpha_t': alpha_t,  # \alpha_t
        'oneover_sqrta': oneover_sqrta,  # 1/\sqrt{\alpha_t}
        'sqrt_beta_t': sqrt_beta_t,  # \sqrt{\beta_t}
        'alphabar_t': alphabar_t,  # \bar{\alpha_t}
        'sqrtab': sqrtab,  # \sqrt{\bar{\alpha_t}}
        'sqrtmab': sqrtmab,  # \sqrt{1-\bar{\alpha_t}}
        'mab_over_sqrtmab': mab_over_sqrtmab_inv,  # (1-\alpha_t)/\sqrt{1-\bar
{\alpha_t}}
    }
```

Listing 2.2: Python code of **ddpm_schedules** of DDPM.

2.1.3 Loss function

MSE loss is used to calculate loss between input image and generated image.

2.2 Data loader

Images are pre-processed with `Resize` to 64×64 and then `Normalize` with mean $(0.5, 0.5, 0.5)$ and standard deviation $(0.5, 0.5, 0.5)$. And labels are converted to one-hot vectors with total 24 classes (See Listings 2.3 and A.3 for details).

```python
class _ClevrTrainingDataset(Dataset):
    def __init__(self, dir_dataset: str, dir_root='iclevr'):
        self.transforms = transforms.Compose([
            transforms.Resize((64, 64)),
            transforms.ToTensor(),
```

```
            transforms.Normalize((0.5, 0.5, 0.5),(0.5, 0.5, 0.5))])

    def __getitem__(self, index):
        img = Image.open(
            Path(self.dir_dataset, self.dir_root, self.img_names[index])).
convert('RGB')
        img = self.transforms(img)
        cond = self.int2one_hot(self.img_conds[index])

        return img, cond

    def int2one_hot(self, int_list):
        one_hot = torch.zeros(self.num_classes)
        for i in int_list:
            one_hot[i] = 1.

        return one_hot
```

Listing 2.3: Python code of data pre-process and loader.

2.3 Hyperparameters

Hyperparameters of this experiment are listed below:

- Embedding size: 128.
- Betas of noise schedule: $(10^{-4}, 0.02)$.
- Dropout of context: 0.1.
- Sampling timestep: 400.
- Batch size: 128.
- Number of training epochs: 300.
- Optimizer: Adam.
- Learning rate: 10^{-4}.
- Seed: 42.

Chapter 3

Experimental results

Testing labels are evaluated at once, i.e., generate 32 images with total size $[32, 3, 64, 64]$, and evaluate by pre-trained ResNet. Two different guidance w 0.5 and 2.0 are used.
Scores are listed in Table 3.1.

Labels \ w	test.json	new_test.json
0.5	0.85 (Figure 3.1)	0.83 (Figure 3.3)
2.0	0.82 (Figure 3.2)	0.81 (Figure 3.4)

Table 3.1: Scores with guidance w 0.5 and 2.0.

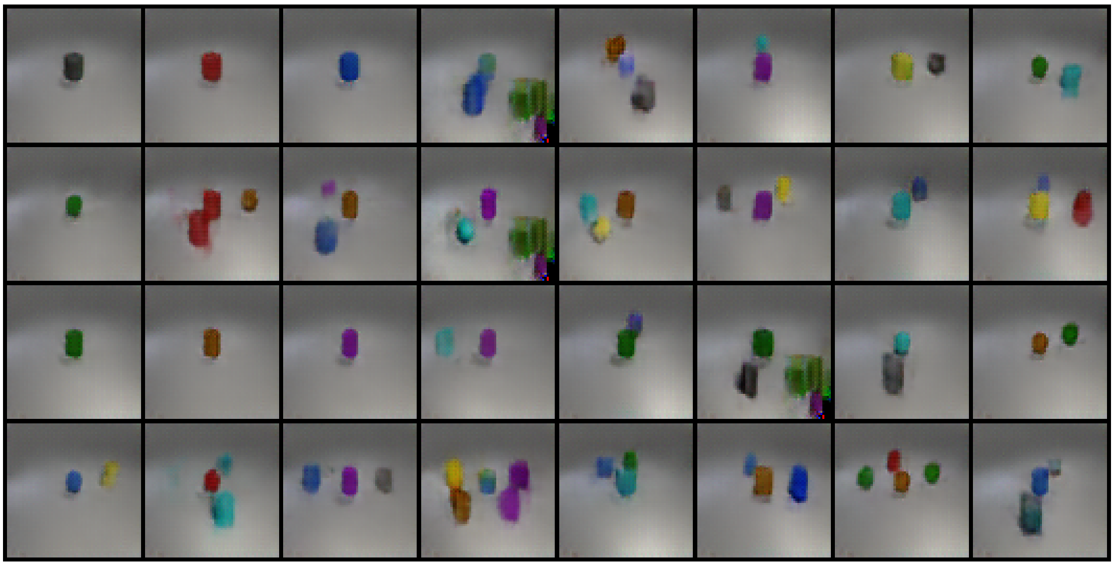

Figure 3.1: Results of test.json with guidance w 0.5.

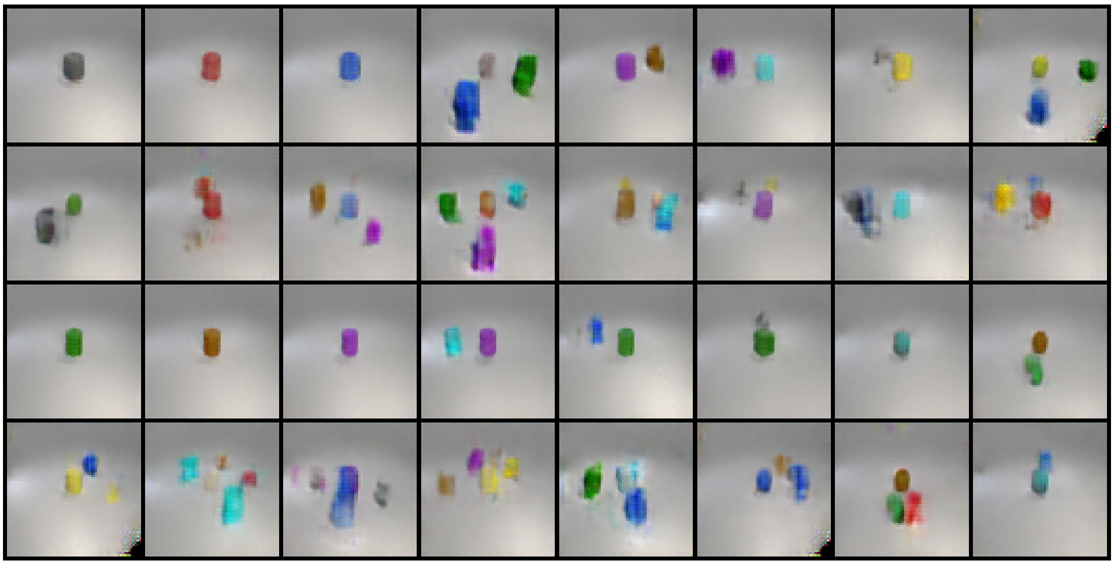

Figure 3.2: Results of test.json with guidance w 2.0.

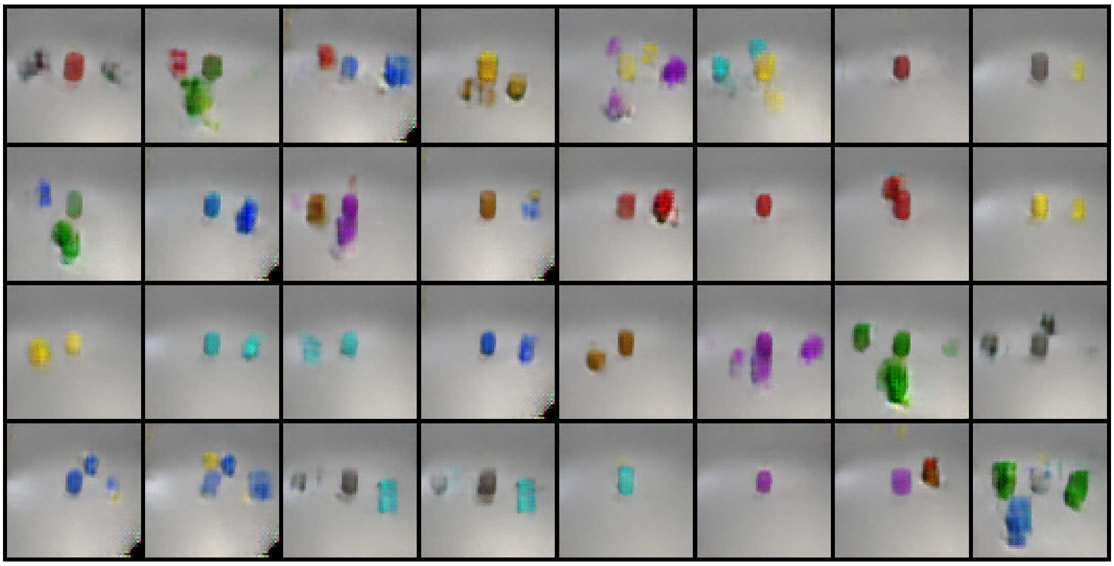

Figure 3.3: Results of new_test.json with guidance w 0.5.

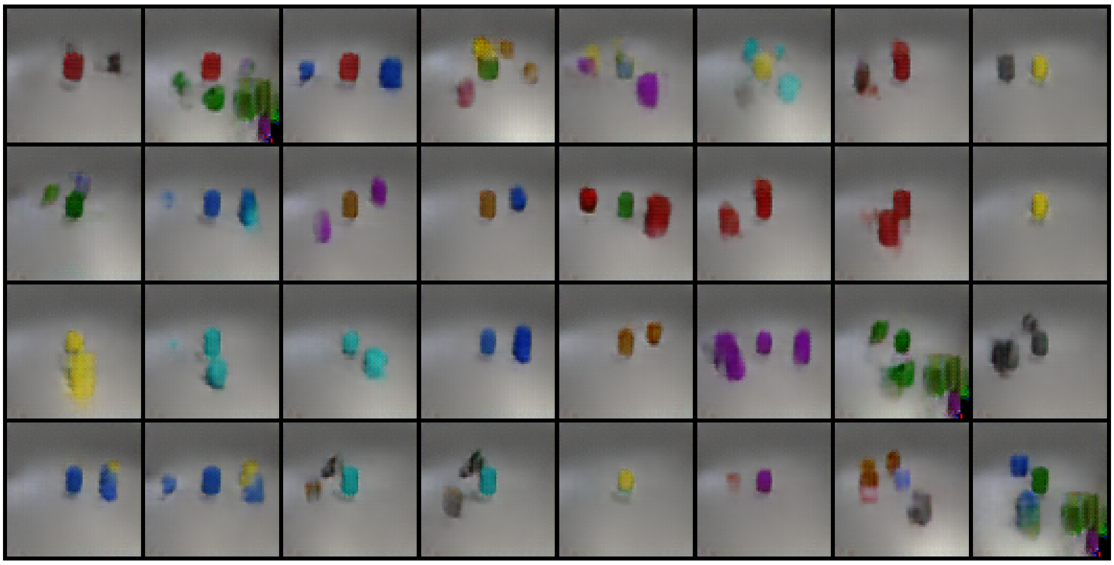

Figure 3.4: Results of new_test.json with guidance w 2.0.

Chapter 4

Discussion

As model architecture listed in Chapter 2, it's sufficient for model to learn and generate objects with specific multi-labels.
If activation of embedding layers, i.e., `GELU` activation layer between two fully connected layers, is removed, the score of generated images with testing labels would drop to about **0.7**. This might be caused by the non-linearity is removed.
Also, mask is predicted during image generation, and it's more intuitive and simple to optimize through MSE loss between input images and generated images.

Appendices

Appendix A

Code

```
class ContextUnet(nn.Module):
    def __init__(self, in_channels: int, n_feat=128, n_classes=24):
        super().__init__()

        self.in_channels = in_channels
        self.n_feat = n_feat
        self.n_classes = n_classes

        self.init_conv = ResidualConvBlock(in_channels, n_feat, is_res=True)

        self.down1 = UnetDown(n_feat, n_feat)
        self.down2 = UnetDown(n_feat, 2 * n_feat)

        self.to_vec = nn.Sequential(
            nn.AvgPool2d(7),
            nn.GELU())

        self.timeembed1 = EmbedFC(1, 2 * n_feat)
        self.timeembed2 = EmbedFC(1, 1 * n_feat)
        self.contextembed1 = EmbedFC(n_classes, 2 * n_feat)
        self.contextembed2 = EmbedFC(n_classes, 1 * n_feat)

        self.up0 = nn.Sequential(
            nn.ConvTranspose2d(2 * n_feat, 2 * n_feat, 8, 8),
            nn.GroupNorm(8, 2 * n_feat),
            nn.ReLU(),
        )

        self.up1 = UnetUp(4 * n_feat, n_feat)
        self.up2 = UnetUp(2 * n_feat, n_feat)
        self.out = nn.Sequential(
```

```
            nn.Conv2d(2 * n_feat, n_feat, 3, 1, 1),
            nn.GroupNorm(8, n_feat),
            nn.ReLU(),
            nn.Conv2d(n_feat, self.in_channels, 3, 1, 1),
        )

    def forward(self,
            x: torch.FloatTensor, c: torch.FloatTensor, t: torch.FloatTensor,
            context_mask: torch.FloatTensor) -> torch.FloatTensor:
        # x is (noisy) image, c is context label, t is timestep,
        # context_mask says which samples to block the context on

        x = self.init_conv(x)
        down1 = self.down1(x)
        down2 = self.down2(down1)
        hiddenvec = self.to_vec(down2)

        context_mask = (-1 * (1 - context_mask)) # flip 0 <-> 1
        c = c * context_mask

        # embed context, time step
        cemb1 = self.contextembed1(c).view(-1, self.n_feat * 2, 1, 1)
        temb1 = self.timeembed1(t).view(-1, self.n_feat * 2, 1, 1)
        cemb2 = self.contextembed2(c).view(-1, self.n_feat, 1, 1)
        temb2 = self.timeembed2(t).view(-1, self.n_feat, 1, 1)

        up1 = self.up0(hiddenvec)
        up2 = self.up1(cemb1 * up1 + temb1, down2)
        up3 = self.up2(cemb2 * up2 + temb2, down1)
        out = self.out(torch.cat((up3, x), 1))
        return out
```

Listing A.1: Full Python code of **ContextUnet** of DDPM.

```
class ResidualConvBlock(nn.Module):
    def __init__(self,
            in_channels: int, out_channels: int, is_res: bool=False) -> None:
        super().__init__()

        self.same_channels = (in_channels==out_channels)
        self.is_res = is_res
        self.conv1 = nn.Sequential(
            nn.Conv2d(in_channels, out_channels, 3, 1, 1),
            nn.BatchNorm2d(out_channels),
            nn.GELU(),
        )
```

```
        self.conv2 = nn.Sequential(
            nn.Conv2d(out_channels, out_channels, 3, 1, 1),
            nn.BatchNorm2d(out_channels),
            nn.GELU(),
        )

    def forward(self, x: torch.FloatTensor) -> torch.FloatTensor:
        if self.is_res:
            x1 = self.conv1(x)
            x2 = self.conv2(x1)

            if self.same_channels:
                out = x + x2
            else:
                out = x1 + x2

            return out / 1.414
        else:
            x1 = self.conv1(x)
            x2 = self.conv2(x1)

            return x2

class UnetDown(nn.Module):
    def __init__(self, in_channels: int, out_channels: int):
        super().__init__()

        self.layers = nn.Sequential(
            ResidualConvBlock(in_channels, out_channels),
            nn.MaxPool2d(2)
        )

    def forward(self, x: torch.FloatTensor) -> torch.FloatTensor:
        return self.layers(x)

class UnetUp(nn.Module):
    def __init__(self, in_channels: int, out_channels: int):
        super().__init__()

        self.layers = nn.Sequential(
            nn.ConvTranspose2d(in_channels, out_channels, 2, 2),
            ResidualConvBlock(out_channels, out_channels),
            ResidualConvBlock(out_channels, out_channels),
        )

```

```python
    def forward(self, x: torch.FloatTensor, skip: torch.FloatTensor) -> torch.
    FloatTensor:
        x = torch.cat((x, skip), 1)
        x = self.layers(x)

        return x

class EmbedFC(nn.Module):
    def __init__(self, input_dim: int, emb_dim: int):
        super().__init__()

        self.input_dim = input_dim
        self.layers = nn.Sequential(
            nn.Linear(input_dim, emb_dim),
            nn.GELU(),
            nn.Linear(emb_dim, emb_dim))

    def forward(self, x: torch.FloatTensor) -> torch.FloatTensor:
        x = x.view(-1, self.input_dim)

        return self.layers(x)
```

Listing A.2: Python code of components of **ContextUnet**.

```python
def get_train_loader(
        batch_size: int, dir_dataset: str,
        num_workers=8, shuffle=True) -> tuple[DataLoader, int]:
    dataset = _ClevrTrainingDataset(dir_dataset)

    return DataLoader(
        dataset, batch_size=batch_size, num_workers=num_workers, shuffle=
    shuffle), \
    dataset.num_classes, dataset.data_shape

def get_test_labels(dir_dataset: str) -> list:
    with open(Path(dir_dataset, 'objects.json'), 'r') as file:
        classes = json.load(file)

    with open(Path(dir_dataset, 'test.json'), 'r') as file:
        conds_list = json.load(file)

    labels = torch.zeros(len(conds_list), len(classes))
    for i, conds in enumerate(conds_list):
        for cond in conds:
            labels[i, int(classes[cond])] = 1.

```

```
    return labels

class _ClevrTrainingDataset(Dataset):
    def __init__(self, dir_dataset: str, dir_root='iclevr'):
        self.dir_dataset = dir_dataset
        self.dir_root = dir_root

        with open(Path(dir_dataset, 'objects.json'), 'r') as file:
            self.classes = json.load(file)
        self.num_classes = len(self.classes)

        self.transforms = transforms.Compose([
            transforms.Resize((64, 64)),
            transforms.ToTensor(),
            transforms.Normalize((0.5, 0.5, 0.5),(0.5, 0.5, 0.5))])

        self.max_objects = 0
        self.img_names = []
        self.img_conds = []

        with open(Path(dir_dataset, 'train.json'), 'r') as file:
            dict = json.load(file)
            for name, conds in dict.items():
                self.img_names.append(name)
                self.max_objects = max(self.max_objects, len(conds))
                self.img_conds.append([self.classes[cond] for cond in conds])

        img, _ = self.__getitem__(0)
        self.data_shape = img.shape

    def __len__(self):
        return len(self.img_names)

    def __getitem__(self, index):
        img = Image.open(
            Path(self.dir_dataset, self.dir_root, self.img_names[index])).
    convert('RGB')
        img = self.transforms(img)
        cond = self.int2one_hot(self.img_conds[index])

        return img, cond

    def int2one_hot(self,int_list):
        one_hot = torch.zeros(self.num_classes)
        for i in int_list:
```

```
            one_hot[i] = 1.

        return one_hot
```

Listing A.3: Python code of data pre-process and loader.