Deep Learning Lab6

110550085 房天越

1. Introduction

In this homework, we aim to implement a Denoising Diffusion Probabilistic Model to do Image Generation through multi-label condition. Then, use a pretrained model with ResNet18 on this task to test its accuracy, our goal is to make this score as high as possible.

2. Implementation Details

2.1 Model Architecture

We apply UNet2DModel from HuggingFace as the noise predictor.

The block channels are set as [128, 128. 256, 256, 512, 512].

The Down and Up blocks is a mix of standard and attention-based modules.

For the conditional embedding, we use one-hot labels to linear projection into a 512-dim time embedding via nn.Linear(num_classes, time_embed_dim), which we feed into UNet as class embeddings.

```python
# UNet2DModel
self.unet = UNet2DModel(
    sample_size=sample_size,
    in_channels=3,
    out_channels=3,
    layers_per_block=2,
    block_out_channels=[128,128,256,256,512,512],
    down_block_types=[
        "DownBlock2D",
        "DownBlock2D",
        "DownBlock2D",
        "DownBlock2D",
        "AttnDownBlock2D",
        "DownBlock2D",
    ],
    up_block_types=[
        "UpBlock2D",
        "AttnUpBlock2D",
        "UpBlock2D",
        "UpBlock2D",
        "UpBlock2D",
        "UpBlock2D",
    ],
    class_embed_type="identity",
)
```

2.2 Noise Scheduler

We employ DDPMScheduler with 1000 training timesteps and the squaredcos_cap_v2 schedule, which is an improved cosine schedule for smoother noise variance decay.

```python
def __init__(
    self,
    num_classes: int = 24,
    time_embed_dim: int = 512,          # 缩
    sample_size: int = 64,
    beta_schedule: str = "squaredcos_cap_v2",
    guidance_scale: float = 2.0,
):
    super().__init__()
    self.guidance_scale = guidance_scale

    # Scheduler for noise
    self.scheduler = DDPMScheduler(
        num_train_timesteps=1000,
        beta_schedule=beta_schedule,
        prediction_type="epsilon",
    )
```

2.3 Classifier-Free Guidance

During training, we drop the class embedding to simulate unconditional generation with probability = 0.2.

At sampling time, we compute:

$$\epsilon = \epsilon_{\text{uncond}} + s \cdot \left(\epsilon_{\text{cond}} - \epsilon_{\text{uncond}}\right)$$

, where s is the guidance scale we can adjust.

```python
def sample(self, y, device, num_inference_steps=1000):
    b = y.shape[0]
    x = torch.randn(b, 3, 64, 64, device=device)
    scheduler = self.scheduler
    scheduler.set_timesteps(num_inference_steps, device=device)

    y_zero = torch.zeros_like(y)
    for t in scheduler.timesteps:
        eps_uncond = self.unet(x, t, self.class_embedding(y_zero)).sample
        eps_cond   = self.unet(x, t, self.class_embedding(y)).sample
        eps = eps_uncond + self.guidance_scale * (eps_cond - eps_uncond)
        step = scheduler.step(eps, t, x)
        x = step.prev_sample

    return x

def add_noise(self, x, noise, timesteps):
    return self.scheduler.add_noise(x, noise, timesteps)
```

2.4 Training Configuration

Data Loader: A custom IClevrDataset reads JSON for image mappings, resize to 64*64, converts to tensor, and normalizes. The batch size is set to 32.

```python
class IClevrDataset(Dataset):
    def __init__(self, json_path, image_dir, transform=None):
        with open(json_path) as f:
            self.conditions = json.load(f)
        self.image_dir = image_dir
        self.transform = transform or transforms.Compose([
            transforms.Resize((64,64)),
            #transforms.RandomHorizontalFlip(p=0.5),
            #transforms.ColorJitter(0.1,0.1,0.1,0.1),
            transforms.ToTensor(),
            transforms.Normalize((0.5,)*3, (0.5,)*3)
        ])
        self.keys = list(self.conditions.keys())

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

    def __getitem__(self, idx):
        fname = self.keys[idx]
        labels = self.conditions[fname]
        path = os.path.join(self.image_dir, fname)
        img = Image.open(path).convert('RGB')
        x = self.transform(img)
        y = torch.zeros(len(OBJECTS_MAP))
        for obj in labels:
            y[OBJECTS_MAP[obj]] = 1
        return x, y
```

Optimizer and Scheduler: Adam with 1e-5 and and CosineAnnealingLR over the epochs.

Mixed Precision: We use torch.cuda.amp to accelerate training and save memory.

```python
optimizer = optim.Adam(model.parameters(), lr=args.lr)
lr_scheduler = optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=args.epochs)
scaler = GradScaler()  # AMP
```

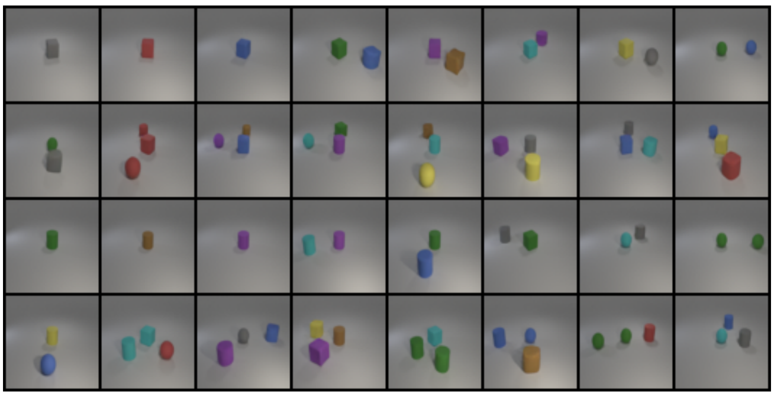
Checkpoint and Logging: We save model weights after each epoch and log loss/learning-rate scalars with TensorBoard.

```python
lr_scheduler.step()
writer.add_scalar('train/loss', np.mean(losses), epoch)
writer.add_scalar('train/lr', lr_scheduler.get_last_lr()[0], epoch)

ckpt_path = os.path.join(args.out_dir, f'ddpm_epoch{epoch}.pth')
torch.save({'model': model.state_dict()}, ckpt_path)
```

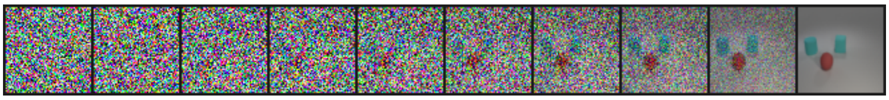
3. Results and Discussion

3.1 test.json



3.2 new_test.json

3.3 Denoising Image of ["red sphere", "cyan cylinder", "cyan cube"]



3.4 Discussion

3.4.1 Guidance Scale

During the training process, we compute:

$$\epsilon = \epsilon_{\text{uncond}} + s \cdot \left(\epsilon_{\text{cond}} - \epsilon_{\text{uncond}}\right)$$

I tried to use s = 1 (Fully conditional), s = 1.5, and s = 2, and run for 100 epochs. No significant difference in accuracy is observed.

3.4.2 Noise Scheduler

I tried to use linear, cosine, and squaredcos_cap_v2 schedulers, the performance is squaredcos_cap_v2 > cosine > linear.

The accuracy of linear is about 0.6 after 200 epochs.

The accuracy of cosine is about 0.7 after 200 epochs.

The accuracy of squaredcos_cap_v2 is around 0.88 (tested many times) after 200 epochs.

4. Experimental Results

```
test set (./test.json) accuracy: 84.72%
new_test set (./new_test.json) accuracy: 90.48%
```