

Introduction

In this lab, our task is to implement a conditional Denoising Diffusion Probabilistic Model (DDPM) to generate synthetic images based on provided textual descriptions, such as 'cyan cylinder' or 'brown sphere'.

Implementation details

Denoising Diffusion Probabilistic Models (DDPM)

I utilized `UNet2DModel` from the library `diffusers` in my implementation.

```
class ClassConditionedUnet(nn.Module):
    def __init__(self, num_classes=24):
        super().__init__()

        self.model = UNet2DModel(
            sample_size=64,
            in_channels=3 + num_classes,
            out_channels=3,
            time_embedding_type="positional",
            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",
            ),
        )
```

In the forward function, the one-hot label is first converted into a tensor with shape (24, 64, 64), where 24 is the number of objects, and (64, 64) is the input image size. After the resizing, the input image with noise is concatenated with the label tensor. Finally, the concatenated product is fed into the UNet.

```
def forward(self, x, t, class_labels):
    bs, ch, w, h = x.shape

    class_cond = class_labels.view(bs, class_labels.shape[1], 1, 1).expand(
        bs, class_labels.shape[1], w, h
    )

    net_input = torch.cat((x, class_cond), 1)

    return self.model(net_input, t).sample
```

Noise Schedule

I utilized `DDPMScheduler` from the library `diffusers` in my implementation.

The scheduler is initialized with:

- `num_train_timesteps`: 1000
- `beta_schedule`: "squaredcos_cap_v2"

The `beta_schedule` parameter in the `DDPMScheduler` defines how noise is added to the images over the timesteps during the diffusion process.

The "squaredcos_cap_v2" is a type of noise schedule based on a squared cosine function, which ensures that the noise level starts very low and gradually increases.

```
# Noise scheduler
noise_scheduler = DDPMScheduler(
    num_train_timesteps=config["denoise_steps"], beta_schedule="squaredcos_cap_v2"
)
```

In the training loop, noise is generated and added to the image according to a random timestamp with the noise scheduler.

```
# Add noise
noise = torch.randn_like(images)
timestamp = (
    torch.randint(0, config["denoise_steps"] - 1, (images.shape[0],))
    .long()
    .cuda()
)
noisy_images = noise_scheduler.add_noise(images, noise, timestamp)
```

In the inference phase, the residual is predicted by the UNet in the beginning. After that, the noise scheduler updates the output image according to the predicted residual.

```
# Predict residual
with torch.no_grad():
    residual = model(x, t, labels)

# Update sample
x = noise_scheduler.step(residual, t, x).prev_sample
```

Training

Prepare for the training phase.

```
# Dataset
dataset = CLEVRDataset(
    dataset_dir=config["dataset_dir"],
    train_json=config["train_json_path"],
    objects_json=config["objects_json_path"],
)
train_dataloader = DataLoader(
    dataset, batch_size=config["batch_size"], shuffle=True
)

# Model
model = ClassConditionedUnet(num_classes=config["num_classes"])
model = model.cuda()

# Optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=config["learning_rate"])

# Load checkpoint if available
start_epoch = load_checkpoint(model, optimizer, checkpoint_dir="checkpoints")

# Noise scheduler
noise_scheduler = DDPMsScheduler(
    num_train_timesteps=config["denoise_steps"], beta_schedule="squaredcos_cap_v2"
)

# Loss function
criterion = nn.MSELoss()
```

For each round in the training loop, the model predicts the noise that was added to the image at a specific time step. The MSE loss is calculated between the predicted noise and the actual noise. This loss is then backpropagated, and the model weights are updated accordingly.

```
for epoch in range(start_epoch, config["epochs"]):
    model.train()
    for images, labels in tqdm(train_dataloader):
        # Prepare data
        images = images.cuda()
        labels = labels.cuda()

        # Add noise
        noise = torch.randn_like(images)
        timestamp = (
            torch.randint(0, config["denoise_steps"] - 1, (images.shape[0],))
            .long()
            .cuda()
        )
        noisy_images = noise_scheduler.add_noise(images, noise, timestamp)

        # Forward pass
        predictions = model(noisy_images, timestamp, labels)
        loss = criterion(predictions, noise)

        # Backward pass
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # Save loss value
        losses.append(loss.item())
```

Inference

In the inference stage, the process starts with an image of total random noise. The model predicts the residual iteratively, progressively denoising the image over multiple timesteps until a final synthesized image that aligns with the given conditional labels is generated.

```
# Inference loop
for i, labels in enumerate(dataloader):
    labels = labels.cuda()
    x = torch.randn((1, 3, config["image_size"], config["image_size"])).cuda()

    for t_idx, t in tqdm(
        enumerate(noise_scheduler.timesteps), total=len(noise_scheduler.timesteps)
    ):
        # Predict residual
        with torch.no_grad():
            residual = model(x, t, labels)

        # Update sample
        x = noise_scheduler.step(residual, t, x).prev_sample

        if t_idx in save_timesteps:
            denoise_images_list[i].append(x.clone().cpu().squeeze())

    final_results.append(x.clone().cpu().squeeze())

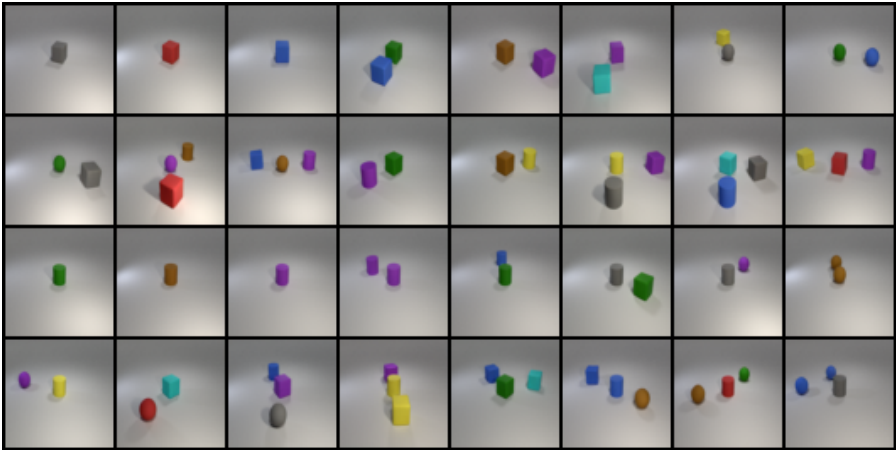

return denoise_images_list, final_results
```

Hyperparameters

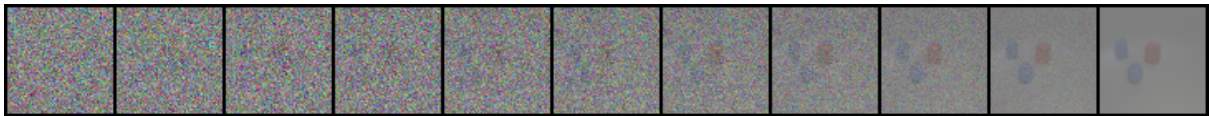
- learning_rate: 0.0001
- batch_size: 32
- epochs: 100
- denoise_steps: 1000
- seed: 910615

Results and Discussion

Synthetic Image Grids

test.json	
new_test.json	

Denoising Process Image



Discussion

The synthetic grids demonstrate the conditional DDPM's ability to generate realistic images based on conditions from the datasets.

The denoising process at the bottom shows the gradual refinement from noise to clear images, illustrating the model's effective image synthesis.

Experiment Results

Classification Accuracy of `test.json` and `new_test.json`

- `test.json` classification accuracy: 0.6805555555555556
- `new_test.json` classification accuracy: 0.8214285714285714

```
~/dlp/lab6  □ □ main *5 !2 .....  
> python3 inference.py  
100%|.....  
Classification accuracy: 0.8214285714285714, testing data: new_test.json  
~/dlp/lab6  □ □ main *5 !2 .....  
> mv results permanent/new_test  
~/dlp/lab6  □ □ main *5 !2 .....  
> python3 inference.py  
100%|.....  
Classification accuracy: 0.6805555555555556, testing data: test.json  
~/dlp/lab6  □ □ main *5 !2 .....  
> mv results permanent/test
```

Inference Commands

- To do model inference: `python3 inference.py`.
- To modify the testing dataset, modify the parameter `eval_json_path` in `config.yaml`. The value of the parameter `eval_json_path` at line 6 should be either `"test.json"` or `"new_test.json"`.
- The synthesized results will be in the directory `results`.