Let’s Play DDPM

Introduction

In this lab, our goal is to implement a conditional Denoising Diffusion Probabilistic Model (DDPM) for generating synthetic images based on specific multi-label conditions. DDPM is a powerful approach that has gained popularity in computer vision, particularly in tasks such as style transfer and image synthesis. By leveraging DDPM, we aim to enhance the generation capabilities of our model.

Implementation

Model

DDPM often adopts UNet-based architectures to model the conditional distribution of an image given its noisy version. The UNet architecture is a key component in DDPM and consists of two main pathways: the downsampling path and the upsampling path.

The downsampling path operates by progressively reducing the size of the input image through a sequence of convolutional layers. This process enables the model to capture hierarchical features at different scales.

On the other hand, the upsampling path aims to reconstruct the output image to its original size by employing a series of transpose convolutional layers. These layers perform the reverse operation of convolutional layers, effectively expanding the feature maps and recovering the finer details of the image. The implementation is as follow:

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

    # self.model is an unconditional Unet with extra channels for class conditioning
    self.model = UNet2DModel(
      sample_size=64,  # the target image resolution
      in_channels=3+num_classes,  # the number of input channels, 3 for RGB images
      out_channels=3,  # the number of output channels
      layers_per_block=2,  # how many ResNet layers to use per UNet block
      block_out_channels=(128, 128, 256, 256, 512, 512),  # the number of output channels for each UNet block
      down_block_types=(
        "DownBlock2D",  # a regular ResNet downsampling block
        "DownBlock2D",
        "DownBlock2D",
        "DownBlock2D",  # a ResNet downsampling block with spatial self-attention
        "AttnDownBlock2D",  # a ResNet downsampling block with spatial self-attention
        "DownBlock2D",
      ),
      up_block_types=(
        "UpBlock2D",  # a regular ResNet upsampling block
        "AttnUpBlock2D",  # a ResNet upsampling block with spatial self-attention
        "UpBlock2D",  # a ResNet upsampling block with spatial self-attention
        "UpBlock2D",
        "UpBlock2D",
        "UpBlock2D",
      ),
    )
```

Noise schedule

In DDPM, a crucial element is the noise schedule, which plays a significant role in the generation process. The noise schedule determines the level of noise that is progressively added to the image during each step of the diffusion process.

In this lab, I examinated two types of noise schedulers, traditional linear scheduler and squared cosine scheduler. In a linear noise schedule, the noise level increases linearly over the diffusion steps. The squared cosine scheduler following the square cosine function that varies smoothly between minimum and maximum noise level.

```
noise_scheduler = DDPMScheduler(num_train_timesteps=1000, beta_schedule='args.scheduler')
```

Loss function

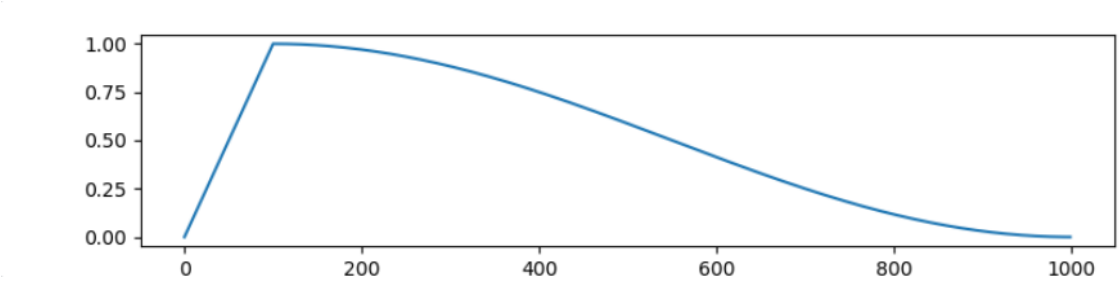
In DDPM the UNet architecture is used to predict the noise that is added to the image during the diffusion process. The predicted noise from the UNet is compared to the ground truth noise using the MSE loss. This approach allows the model to focus on accurately modeling and removing the noise from the input image.

```
clean_images, labels = batch
clean_images = clean_images.to(args.device)
labels = labels.to(args.device)

# Sample noise to add to the images
bs = clean_images.shape[0]
noise = torch.randn_like(clean_images)
timesteps = torch.randint(0, noise_scheduler.num_train_timesteps, (bs,)).to(clean_images.device).long()
# Add noise to the clean images according to the noise magnitude at each timestep
# (this is the forward diffusion process)
noisy_images = noise_scheduler.add_noise(clean_images, noise, timesteps)

pred = model(noisy_images, timesteps, labels)

loss = F.mse_loss(pred, noise)
```

Hyperparameters

The models of following experiments are train with:

- learning rate: 1e-4 with cosine scheduler and warmup step = 500



- epochs: 150
- batch_size: 72

Results & Discussion

In this section, I compare the different choices of noise scheduler or UNet structure, all the models are test on the for 20 times on both test.json and new_test.json to record their average and best accuracy.

Choice of UNet structure

To examinate the impact of UNet structure, I implemented two kinds of UNet, Base UNet and Shallow UNet.

Base UNet consists of 4 convolution block and followed by one attention block and one more convolution block on the top-down path, then reverse it on the way back.

```
class BaseConditionedUnet(nn.Module):
  def __init__(self, num_classes=24):
    ...
      block_out_channels=(128, 128, 256, 256, 512, 512),  # the number of output channels for each UNet block
      down_block_types=(
        "DownBlock2D",  # a regular ResNet downsampling block
        "DownBlock2D",
        "DownBlock2D",
        "DownBlock2D",  # a ResNet downsampling block with spatial self-attention
        "AttnDownBlock2D",  # a ResNet downsampling block with spatial self-attention
        "DownBlock2D",
      ),
      up_block_types=(
        "UpBlock2D",  # a regular ResNet upsampling block
        "AttnUpBlock2D",  # a ResNet upsampling block with spatial self-attention
        "UpBlock2D",  # a ResNet upsampling block with spatial self-attention
        "UpBlock2D",
        "UpBlock2D",
        "UpBlock2D",
      ),
    )
```

For Shallow UNet, I reduce the number of convolution blocks before attention block from 4 to 2, and left the other setting to be the same.

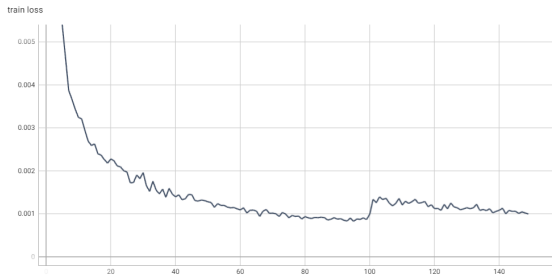
```
class ShallowConditionedUnet(nn.Module):
  def __init__(self):
    ...
      block_out_channels=(128, 128, 256, 256),
      down_block_types=(
        "DownBlock2D",        # a regular ResNet downsampling block
        "DownBlock2D",        # a regular ResNet downsampling block
        "AttnDownBlock2D",    # a ResNet downsampling block with spatial self-attention
        "DownBlock2D",        # a regular ResNet downsampling block
        # "AttnDownBlock2D",
      ),
      up_block_types=(
        # "AttnUpBlock2D",
        "UpBlock2D",          # a regular ResNet upsampling block
        "AttnUpBlock2D",      # a ResNet upsampling block with spatial self-attention
        "UpBlock2D",          # a regular ResNet upsampling block
        "UpBlock2D",          # a regular ResNet upsampling block
      ),
    )
```

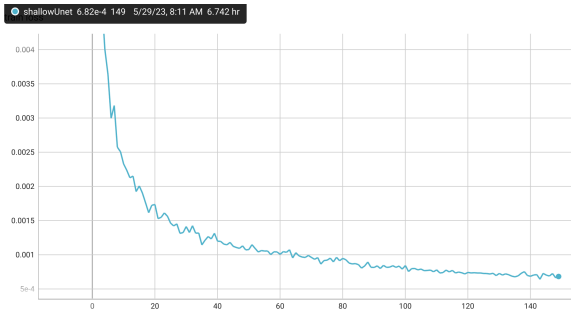
From the following comparison, we can observe that simply increase or decrease number of convolution blocks did not affect the resulting performance significantly.

- accuracy (best / avg)

	Base UNet	Shallow UNet
test.json	75% / 67%	72% / 65%
new_test.json	86% / 78%	85% / 77%

- learning curve



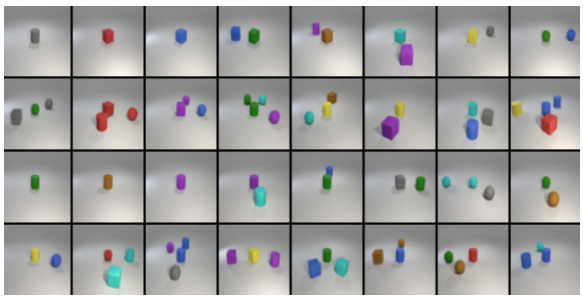


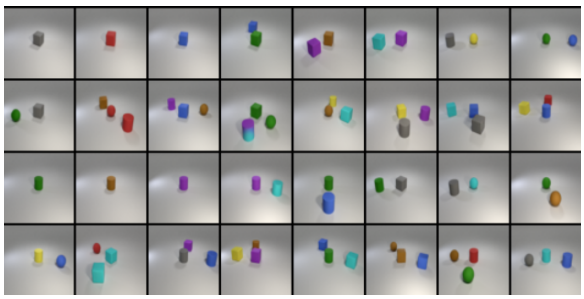
(resumed from ep 100)

- test.json

```
##################################################
load from log/baseUnet/best.pth to test.
##################################################
100%|
Average accuracy:  0.6673611111111111
Best acc:  0.75
```

```
##################################################
load from log/shallowUnet/best.pth to test.
##################################################
100%|
Average accuracy:  0.6479166666666666
Best acc:  0.7222222222222222
```

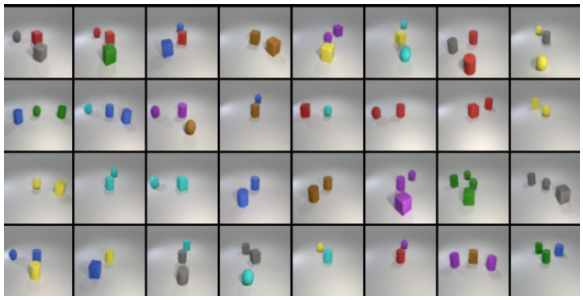




- new_test.json

```
##################################################
load from log/baseUnet/best.pth to test.
##################################################
100%|
Average accuracy:  0.7833333333333334
Best acc:  0.8571428571428571
```

```
##################################################
load from log/shallowUnet/best.pth to test.
##################################################
100%|
Average accuracy:  0.7702380952380952
Best acc:  0.8452380952380952
```



Choice of noise scheduler

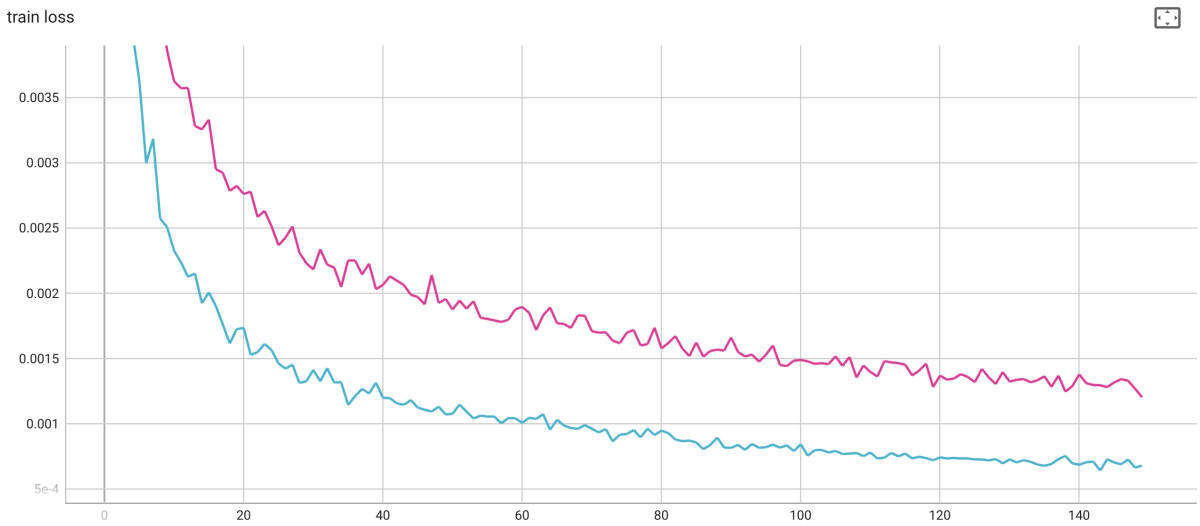
In this part, I compared the impact of different noise scheduler under same UNet structure, here I used the shallow one. According to the comparison below, squaredcos_cap_v2 scheduler perform better than linear one significantly ot test.json, and slightly better at new_test.json, even with a higher training loss.

This outcome aligns with the characteristics of the squared cosine noise scheduler. The squared cosine function gradually increases and then decreases the noise level during the diffusion process. This pattern allows the model to explore a wide range of noise levels and capture both coarse and fine-grained details in the generated images. The non-linear progression of noise levels in the squared cosine scheduler may help the model avoid convergence to suboptimal solutions and enhance the overall quality of the generated images.

- accuracy (best / avg)

	linear	squaredcos_cap_v2
test.json	72% / 65%	88% / 74%
new_test.json	85% / 77%	87% / 81%

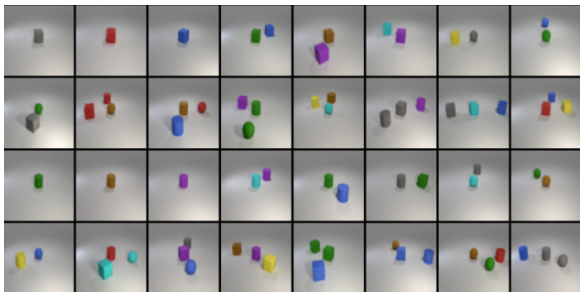
- learning curve (red: squaredcos_cap_v2, blue: linear)

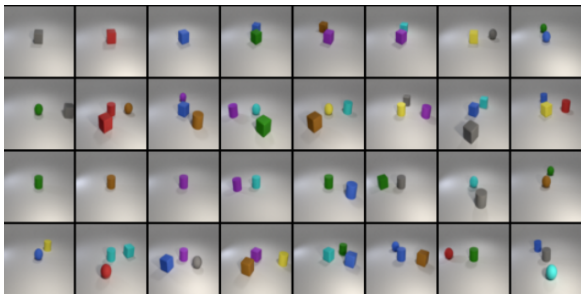


- test.json

```
####################################################
load from log/shallowUnet/best.pth to test.
####################################################
100%|
Average accuracy:  0.6479166666666666
Best acc:  0.7222222222222222
```

```
####################################################
load from log/squaredcos_cap_v2/shallowUnet/best.pth to test.
####################################################
100%|
Average accuracy:  0.7361111111111112
Best acc:  0.875
```





- new_test.json

```
####################################################
load from log/shallowUnet/best.pth to test.
####################################################
100%|
Average accuracy:  0.7702380952380952
Best acc:  0.8452380952380952
```

```
####################################################
load from log/squaredcos_cap_v2/shallowUnet/best.pth to test.
####################################################
100%|
Average accuracy:  0.8053571428571427
Best acc:  0.8690476190476191
```