Lab7: DDPM

Wei-Yun Hsu
Institute of Multimedia Engineering
National Yang Ming Chiao Tung University

1. Introduction

In this lab, we will implement a conditional Denoising Diffusion Probabilistic Models (DDPM) to generate synthetic images.
There are the following requirements:

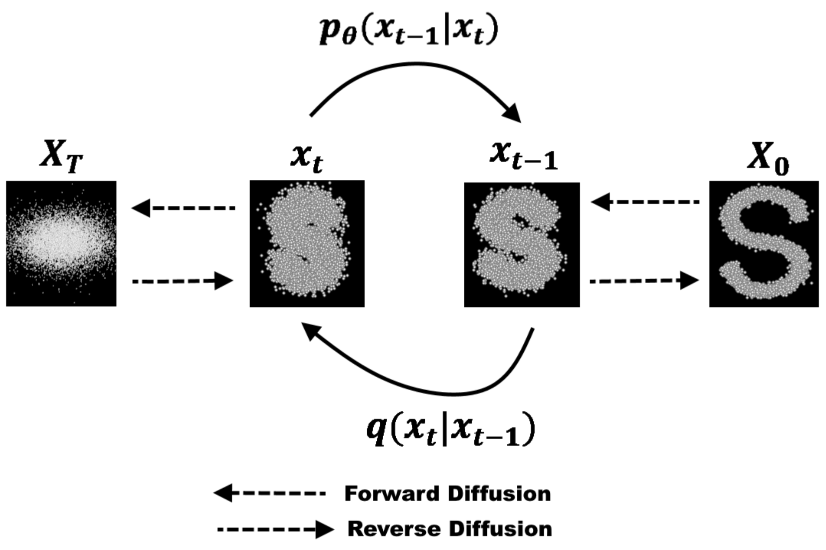
- Implement a conditional DDPM
- Design our noise schedule and UNet architecture
- Implement the dataloader, training function and testing function
- Generate the synthetic images

2. Implementation details

All the following section, we use directly the diffusers toolbox [1] that is user-friendly and flexible to implement our overall architecture and some specific methods.

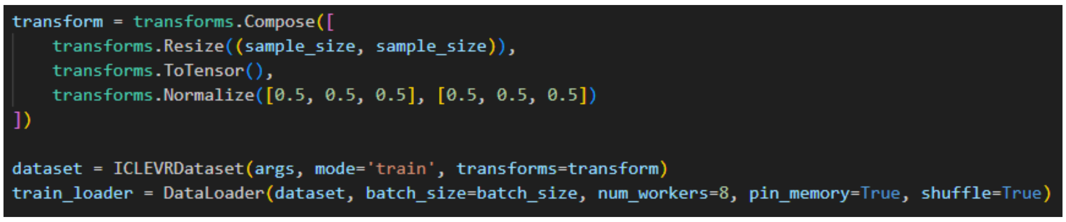
A. Describe how you implement your DDPM

For our task, we need to generate the images with the desired condition, so we can add additional conditioning information to a diffusion model, which is a so-called conditional DDPM. Compared with DDPM, the conditional model is almost identical but adds the encoding of the class label into the timestep by passing the label through an Embedding layer. Our overall DDPM pipeline contains a UNet model and a DDPM scheduler. The pipeline denoises an image by taking random noise the size of the desired output and passing it through the model several times. At each timestep, the model predicts the noise residual and the scheduler uses it to predict a less noisy image. Then, the pipeline repeats the above steps until it reach the expected results. The process of DDPM is shown below, as we can see that it can divided to two parts, forward diffusion and reverse diffusion.

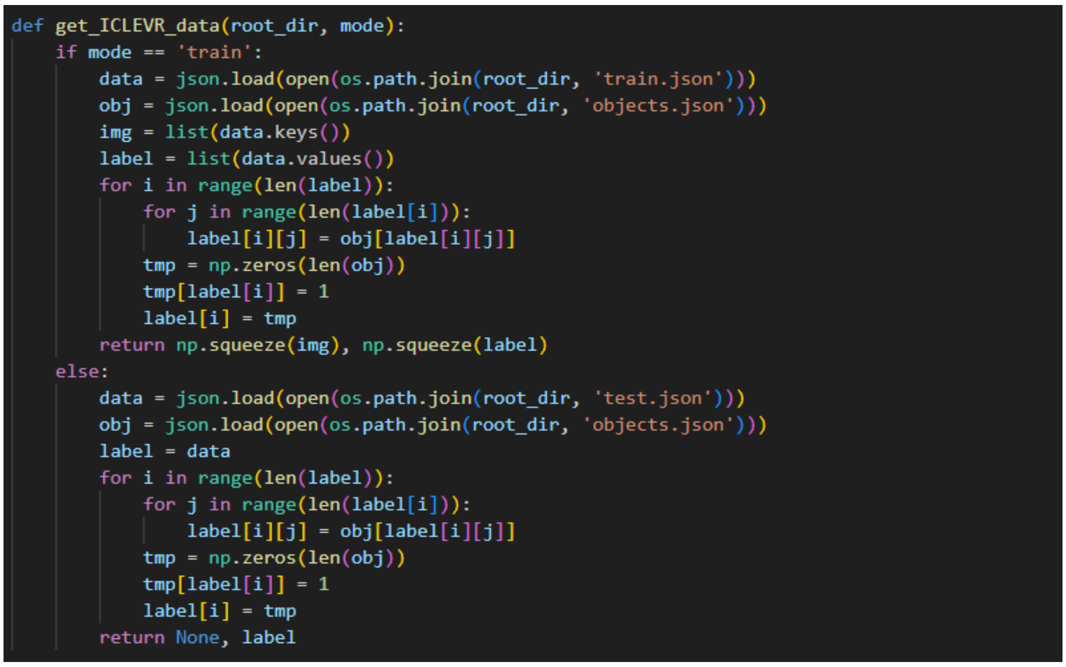


(image source: https://towardsdatascience.com/diffusion-models-made-easy-8414298ce4da)

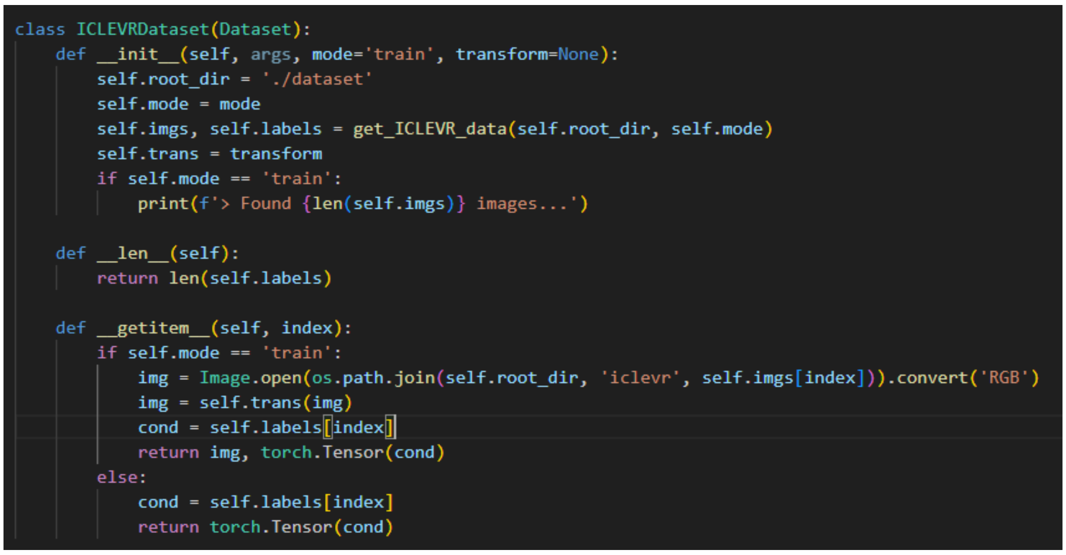
Therefore, our procedure of diffusion models is as follows.
First, we load data from our own dataloader and need to normalize the input images by transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)).



```python
transform = transforms.Compose([
    transforms.Resize((sample_size, sample_size)),
    transforms.ToTensor(),
    transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])
])

dataset = ICLEVRDataset(args, mode='train', transforms=transform)
train_loader = DataLoader(dataset, batch_size=batch_size, num_workers=8, pin_memory=True, shuffle=True)
```

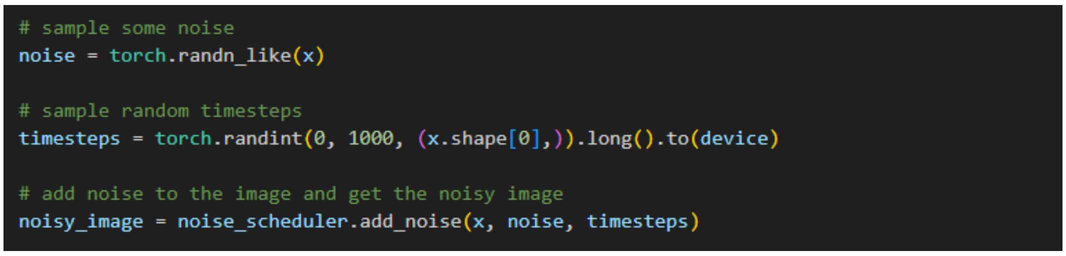


```python
def get_ICLEVR_data(root_dir, mode):
    if mode == 'train':
        data = json.load(open(os.path.join(root_dir, 'train.json')))
        obj = json.load(open(os.path.join(root_dir, 'objects.json')))
        img = list(data.keys())
        label = list(data.values())
        for i in range(len(label)):
            for j in range(len(label[i])):
                label[i][j] = obj[label[i][j]]
            tmp = np.zeros(len(obj))
            tmp[label[i]] = 1
            label[i] = tmp
        return np.squeeze(img), np.squeeze(label)
    else:
        data = json.load(open(os.path.join(root_dir, 'test.json')))
        obj = json.load(open(os.path.join(root_dir, 'objects.json')))
        label = data
        for i in range(len(label)):
            for j in range(len(label[i])):
                label[i][j] = obj[label[i][j]]
            tmp = np.zeros(len(obj))
            tmp[label[i]] = 1
            label[i] = tmp
        return None, label
```



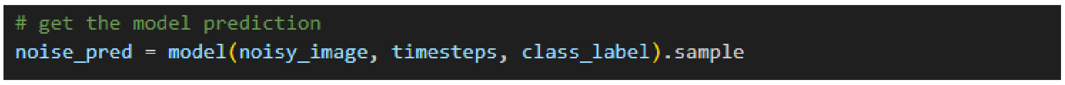
```python
class ICLEVRDataset(Dataset):
    def __init__(self, args, mode='train', transform=None):
        self.root_dir = './dataset'
        self.mode = mode
        self.imgs, self.labels = get_ICLEVR_data(self.root_dir, self.mode)
        self.trans = transform
        if self.mode == 'train':
            print(f'> Found {len(self.imgs)} images...')

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

    def __getitem__(self, index):
        if self.mode == 'train':
            img = Image.open(os.path.join(self.root_dir, 'iclevr', self.imgs[index])).convert('RGB')
            img = self.trans(img)
            cond = self.labels[index]
            return img, torch.Tensor(cond)
        else:
            cond = self.labels[index]
            return torch.Tensor(cond)
```

From here, this is the forward process, we add noise to the clean images at each timestep.



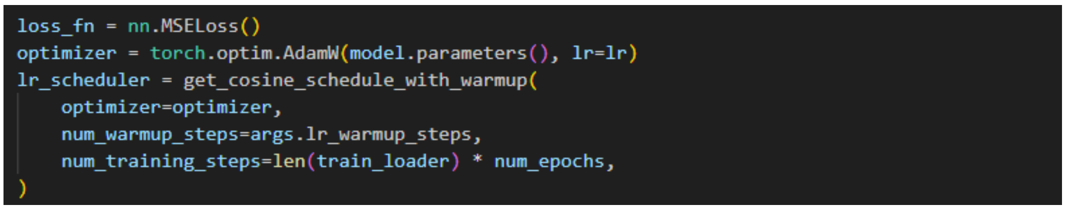
```python
# sample some noise
noise = torch.randn_like(x)

# sample random timesteps
timesteps = torch.randint(0, 1000, (x.shape[0],)).long().to(device)

# add noise to the image and get the noisy image
noisy_image = noise_scheduler.add_noise(x, noise, timesteps)
```

After getting the noisy image, we feed the noisy image to the model and get the model prediction.

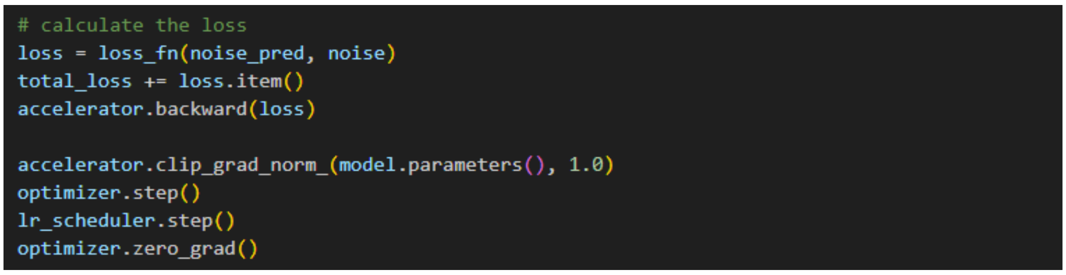


```python
# get the model prediction
noise_pred = model(noisy_image, timesteps, class_label).sample
```

Next, we evaluate how well the model denoises by MSE loss function and update the model by AdamW optimizer.

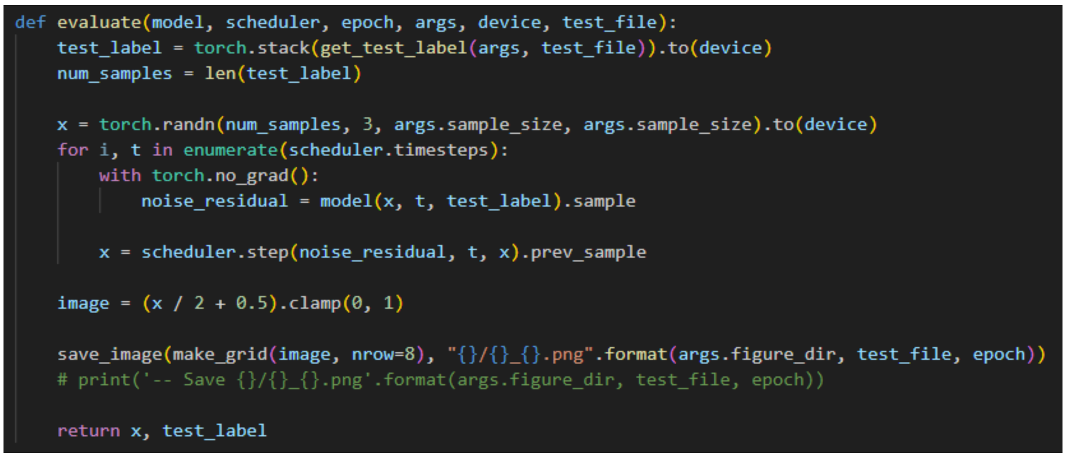


```python
loss_fn = nn.MSELoss()
optimizer = torch.optim.AdamW(model.parameters(), lr=lr)
lr_scheduler = get_cosine_schedule_with_warmup(
    optimizer=optimizer,
    num_warmup_steps=args.lr_warmup_steps,
    num_training_steps=len(train_loader) * num_epochs,
)
```



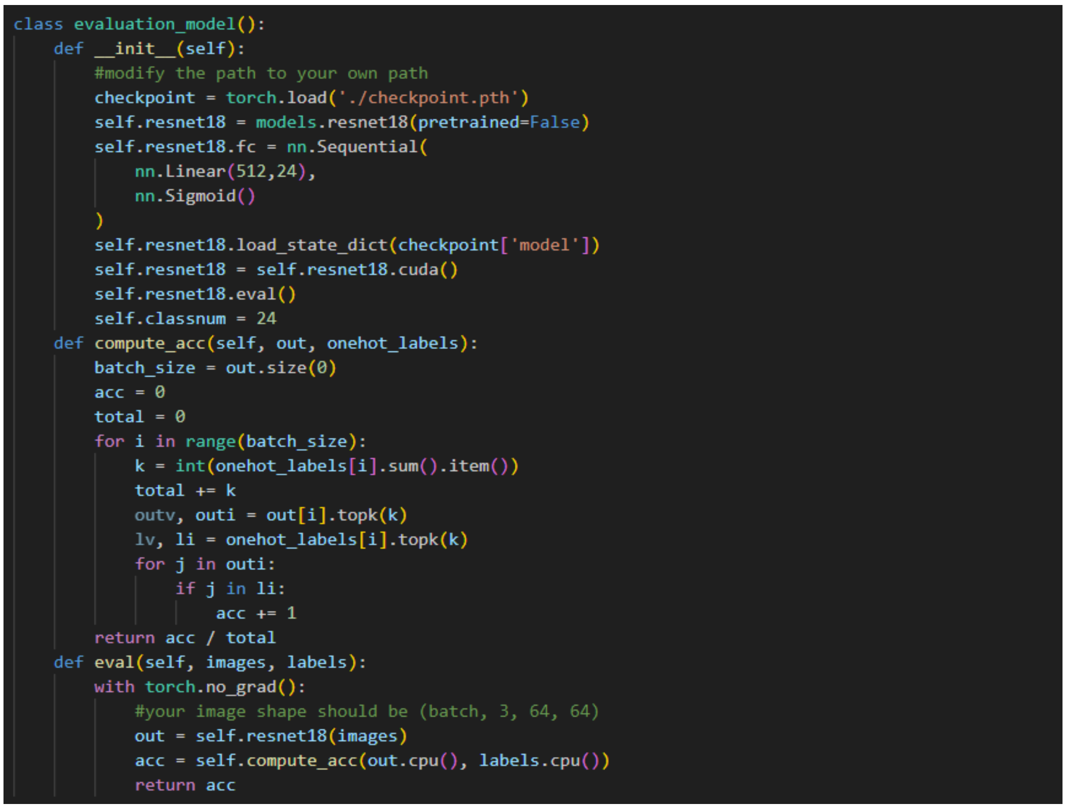
```python
# calculate the loss
loss = loss_fn(noise_pred, noise)
total_loss += loss.item()
accelerator.backward(loss)

accelerator.clip_grad_norm_(model.parameters(), 1.0)
optimizer.step()
lr_scheduler.step()
optimizer.zero_grad()
```

Then, since we have already applied normalization for the input images, in order to generate RGB images in the reverse diffusion process, we apply the de-normalization to finish it.

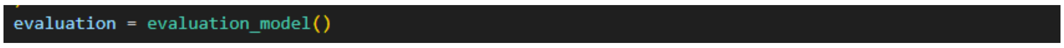


```python
def evaluate(model, scheduler, epoch, args, device, test_file):
    test_label = torch.stack(get_test_label(args, test_file)).to(device)
    num_samples = len(test_label)

    x = torch.randn(num_samples, 3, args.sample_size, args.sample_size).to(device)
    for i, t in enumerate(scheduler.timesteps):
        with torch.no_grad():
            noise_residual = model(x, t, test_label).sample

        x = scheduler.step(noise_residual, t, x).prev_sample

    image = (x / 2 + 0.5).clamp(0, 1)

    save_image(make_grid(image, nrow=8), "{}/{}_{}.png".format(args.figure_dir, test_file, epoch))
    # print('-- Save {}/{}_{}.png'.format(args.figure_dir, test_file, epoch))

    return x, test_label
```

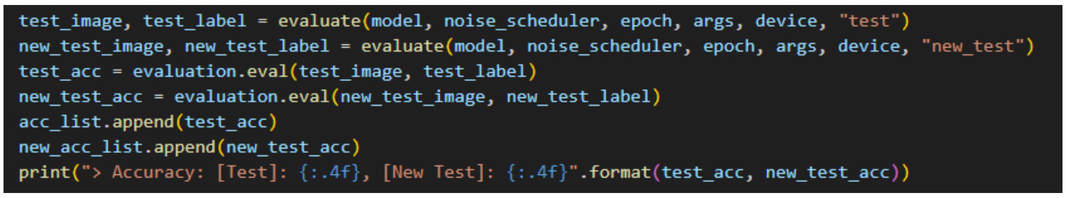
Finally, we use ResNet18 as our evaluator to compute the accuracy.



```python
class evaluation_model():
    def __init__(self):
        #modify the path to your own path
        checkpoint = torch.load('./checkpoint.pth')
        self.resnet18 = models.resnet18(pretrained=False)
        self.resnet18.fc = nn.Sequential(
            nn.Linear(512,24),
            nn.Sigmoid()
        )
        self.resnet18.load_state_dict(checkpoint['model'])
        self.resnet18 = self.resnet18.cuda()
        self.resnet18.eval()
        self.classnum = 24
    def compute_acc(self, out, onehot_labels):
        batch_size = out.size(0)
        acc = 0
        total = 0
        for i in range(batch_size):
            k = int(onehot_labels[i].sum().item())
            total += k
            outv, outi = out[i].topk(k)
            lv, li = onehot_labels[i].topk(k)
            for j in outi:
                if j in li:
                    acc += 1
        return acc / total
    def eval(self, images, labels):
        with torch.no_grad():
            #your image shape should be (batch, 3, 64, 64)
            out = self.resnet18(images)
            acc = self.compute_acc(out.cpu(), labels.cpu())
            return acc
```



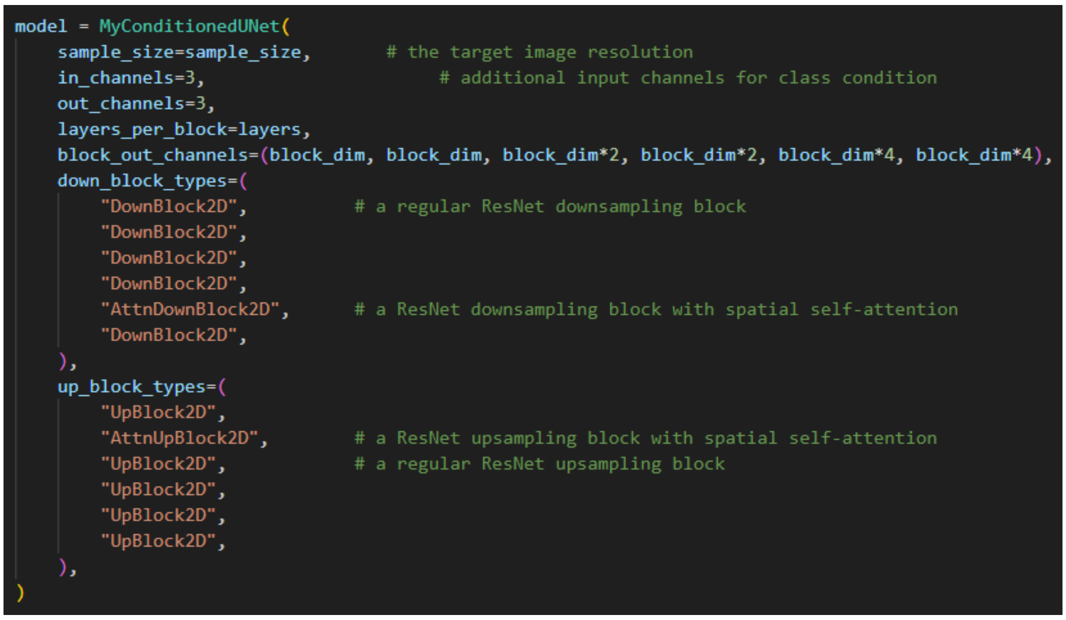
```python
evaluation = evaluation_model()
```



```python
test_image, test_label = evaluate(model, noise_scheduler, epoch, args, device, "test")
new_test_image, new_test_label = evaluate(model, noise_scheduler, epoch, args, device, "new_test")
test_acc = evaluation.eval(test_image, test_label)
new_test_acc = evaluation.eval(new_test_image, new_test_label)
acc_list.append(test_acc)
new_acc_list.append(new_test_acc)
print("> Accuracy: [Test]: {:.4f}, [New Test]: {:.4f}".format(test_acc, new_test_acc))
```

B. Describe how you implement your UNet architectures

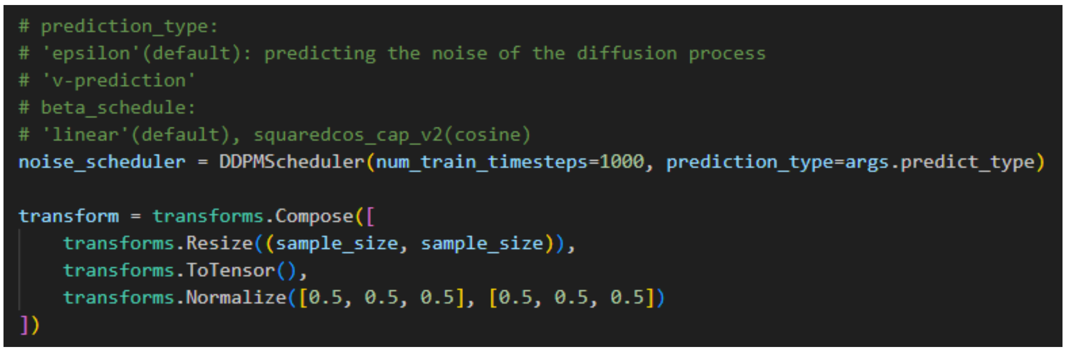
To fit our task, we also design our UNet architecture with conditional mode. We have the classic UNet architecture with downsampling and upsampling paths. The main difference with traditional UNet is that the up and down blocks support an extra timestep argument and add the encoding of the class label into the timestep by passing the label on their forward pass. In the up and down blocks, we have 4 block types and mainly use the regular ResNet downsampling block, ResNet downsampling and upsampling block with spatial self-attention and the regular ResNet upsampling block. The output channel of each block would be 128, 128, 256, 256, 512 and 512. More details can refer to `model.py`.



```
model = MyConditionedUNet(
    sample_size=sample_size,       # the target image resolution
    in_channels=3,                      # additional input channels for class condition
    out_channels=3,
    layers_per_block=layers,
    block_out_channels=(block_dim, block_dim, block_dim*2, block_dim*2, block_dim*4, block_dim*4),
    down_block_types=(
        "DownBlock2D",          # a regular ResNet downsampling block
        "DownBlock2D",
        "DownBlock2D",
        "DownBlock2D",
        "AttnDownBlock2D",      # a ResNet downsampling block with spatial self-attention
        "DownBlock2D",
    ),
    up_block_types=(
        "UpBlock2D",
        "AttnUpBlock2D",        # a ResNet upsampling block with spatial self-attention
        "UpBlock2D",            # a regular ResNet upsampling block
        "UpBlock2D",
        "UpBlock2D",
        "UpBlock2D",
    ),
)
```

C. Describe how you design your noise schedule

In the diffusers library, it provides a DDPMScheduler function, the scheduler generates the noisy images and feeds noisy images to the models, during inference we use the model predictions iteratively to remove the noise. Additionally, we found that there is a *prediction_type* parameter in this function, and its default value is “epsilon” which means that it will predict the noise of the diffusion process, and the other value is “v_prediction”, it seems to be demonstrated that its performance is better than “epsilon” for video generation [2]. Consequently, we decided to use these two modes to compare their effects for the process of DDPM. The implementation of defining our noise scheduler is as follows. The other interesting parameter is *beta_scheule*, it define the type of noise schedule that should be used for inference and training, we will try to set it to “linear” (default) and “squaredcos_cap_v2” (cosine) and compare their performance.



```
# prediction_type:
# 'epsilon'(default): predicting the noise of the diffusion process
# 'v-prediction'
# beta_schedule:
# 'linear'(default), squaredcos_cap_v2(cosine)
noise_scheduler = DDPMScheduler(num_train_timesteps=1000, prediction_type=args.predict_type)

transform = transforms.Compose([
    transforms.Resize((sample_size, sample_size)),
    transforms.ToTensor(),
    transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])
])
```

D. Experiment setup

The hyper-parameters for experiments as shown below.

- batch size: 16
- epoch: 150
- learning rate: 0.0001
- sample size: 64
- warmup step of learning rate: 500
- optimizer: AdamW
- loss function: MSE loss

3. Results and discussion

For our the best results, the hyper-parameter settings are shown the previous section. And we have also tried the strategy of learning rate decay during the training phase in order to avoiding the overfitting. Finally, ResNet18 was applied to compute the accuracy of our synthetic images, we expected that the same object will not appear twice in an image.

A. Show your results based on the testing data. (including images)

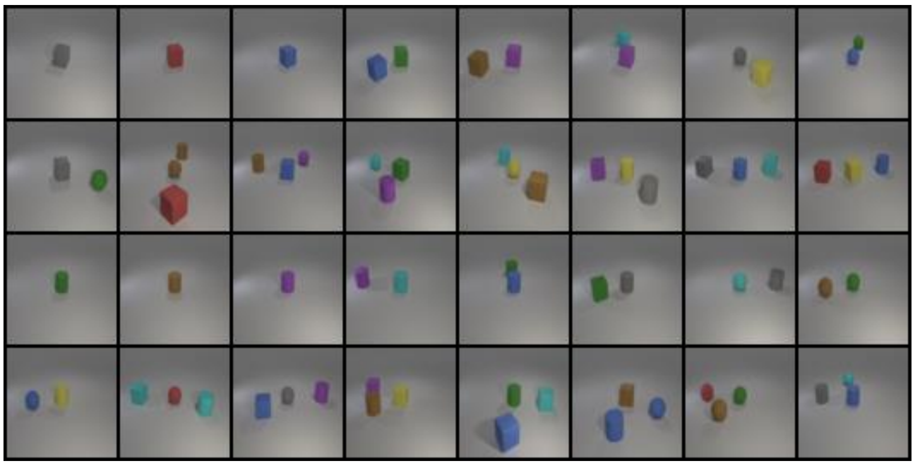


Figure 1: The synthetic images of accuracy of **0.9167** in *test.json* dataset

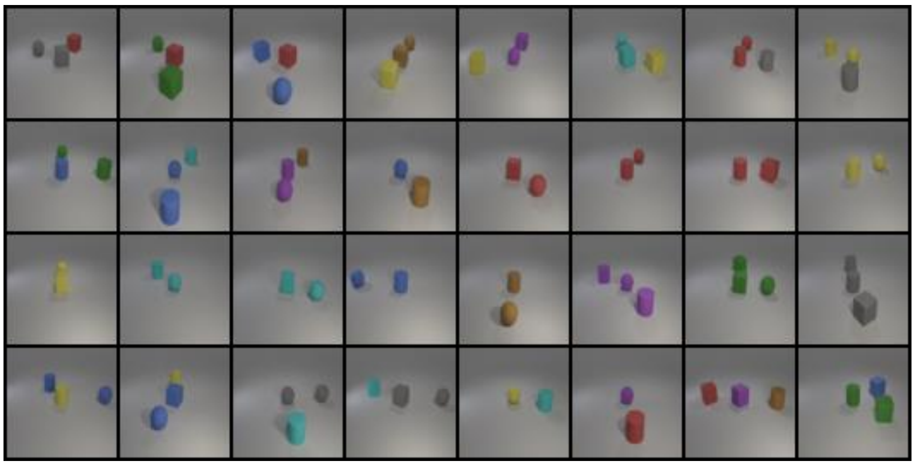


Figure 2: The synthetic images of accuracy of **0.9524** in *new_test.json* dataset

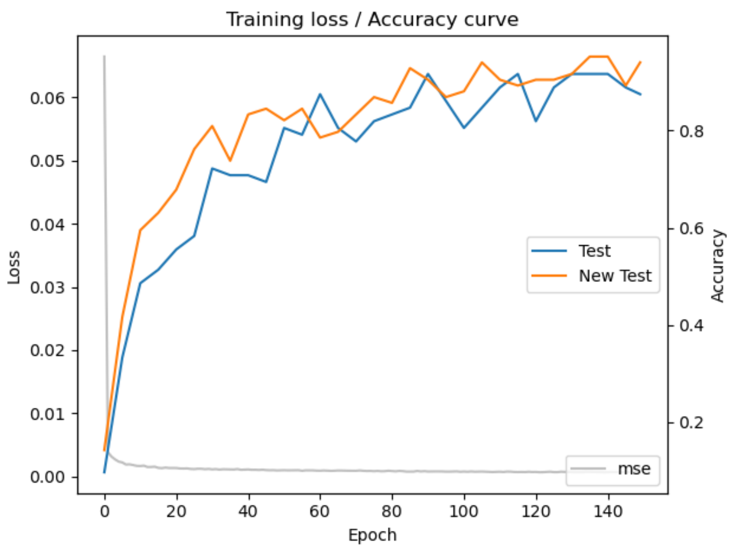


Figure 3: The training curve of our experiment

B. Discuss the results of different model architectures.

The experiment setup for this part is the same as the description in section 2-D.

B-1. Different prediction types for DDPM scheduler

This experiment is inspired by [2], because the author argued that it is effective for the text-conditioned video generation setting and is particularly useful for numerical stability throughout the diffusion process. On the other hand, when we choose the "epsilon" as our prediction type, it is obvious that the synthetic images have the background color shifting at the beginning of training phase. In the reference paper, they also discovered that the "v-prediction" avoids color shifting artifacts that affect high resolution diffusion models. For these reasons, we are curious about its performance on the image generation task.

	Acurracy (test)	Acurracy (new_test)	training loss
epsilon	**0.9167**	**0.9524**	0.0007
v_predicition	0.8472	0.8801	0.0007

Figure 4: The result comparison of different prediction types

As we can see that the result using "v_prediction" parameter isn't better than the one using "epsilon" parameter under the same training loss, so we argued that it may only fit the video generation task. The other thing we are curious about is the color shifting problem using "epsilon" parameter to train DDPM. Take *test.json* data as an example:

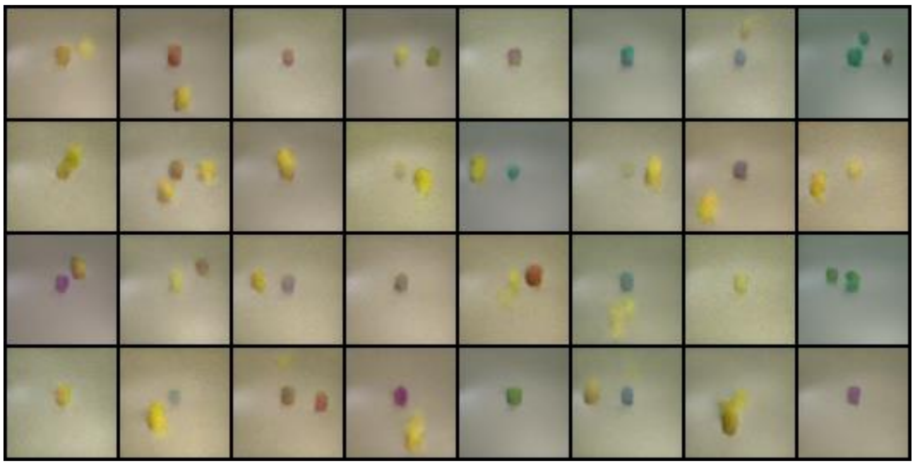


Figure 5: The result of "epsilon" parameter at the very beginning of training

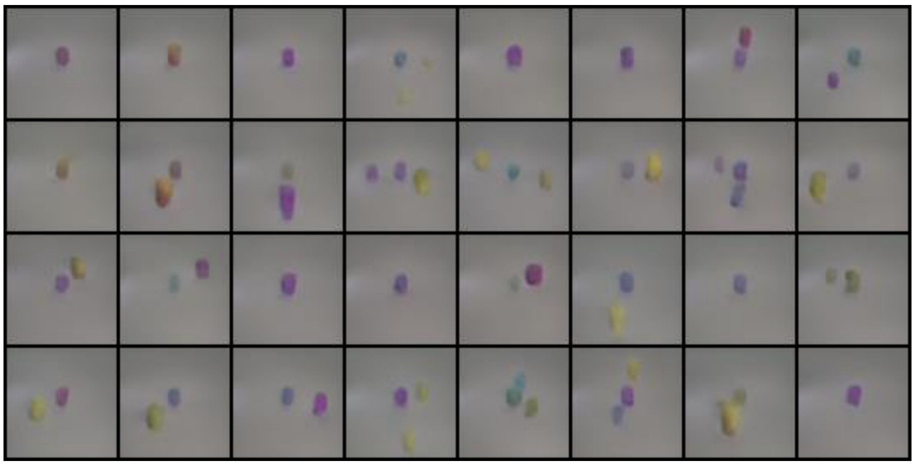


Figure 6: The result of "v_prediction" parameter at the very beginning of training

Indeed, we can demonstrate that the "v_prediction" parameter can mitigate the color shifting phenomenon, the experimental result also confirmed the conclusion in the paper [2].

B-2. Different beta schedules for DDPM scheduler

In the section 2-C, we found that the DDPM noise schedule has many parameters that can be adjusted. The forward diffusion process gradually adds noise to an image from the real distribution, in a number of timesteps. This happens according to a variance schedule, that is *beta_schedule* parameter. The original DDPM authors employed a linear schedule. However, it was shown in [3] that optimal results can be achieved when employing a cosine schedule. Based on the reference paper, we want to reproduce their conclusion. In the diffusers toolbox, the cosine schedule is defined as "squaredcos_cap_v2".

	Acurracy (test)	Acurracy (new_test)	training loss
linear	0.9167	0.9524	0.0007
cosine	**0.9722**	**0.9643**	0.0013

Figure 7: The result comparison of different beta schedules

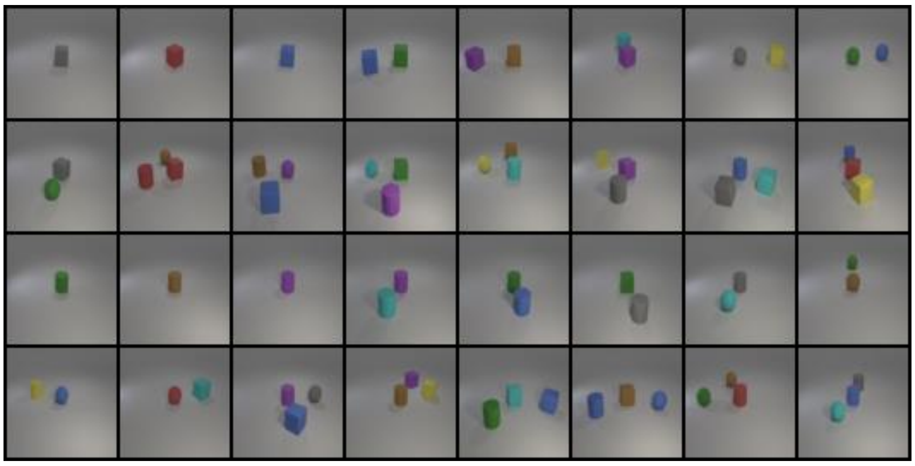


Figure 8: The synthetic images of accuracy of **0.9722** in *test.json* dataset

From the above results, we can confirm that the DDPM using cosine noise schedule outperforms the one with linear schedule and it also keeps the sample quality, this is a claim in the [3]. Additionally, we observe that cosine schedule tends to reach optimal performance more quickly than those trained with the linear schedule thorough the training curve (shown in Figure 9).

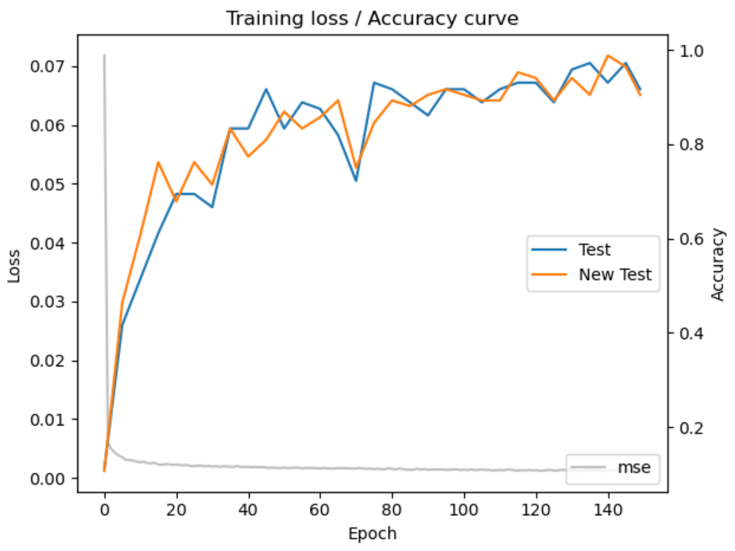


Figure 9: The training curve of conducting cosine schedule

4. Reference

[1] https://huggingface.co/docs/diffusers/v0.16.0/en/index

[2] Imagen Video: High Definition Video Generation with Diffusion Models: https://arxiv.org/abs/2210.02303

[3] Improved Denoising Diffusion Probabilistic Models: https://arxiv.org/abs/2102.09672