Diffusion Model

Lab #6 report

B103040039

王宗義

Deep learning

Spring 2025

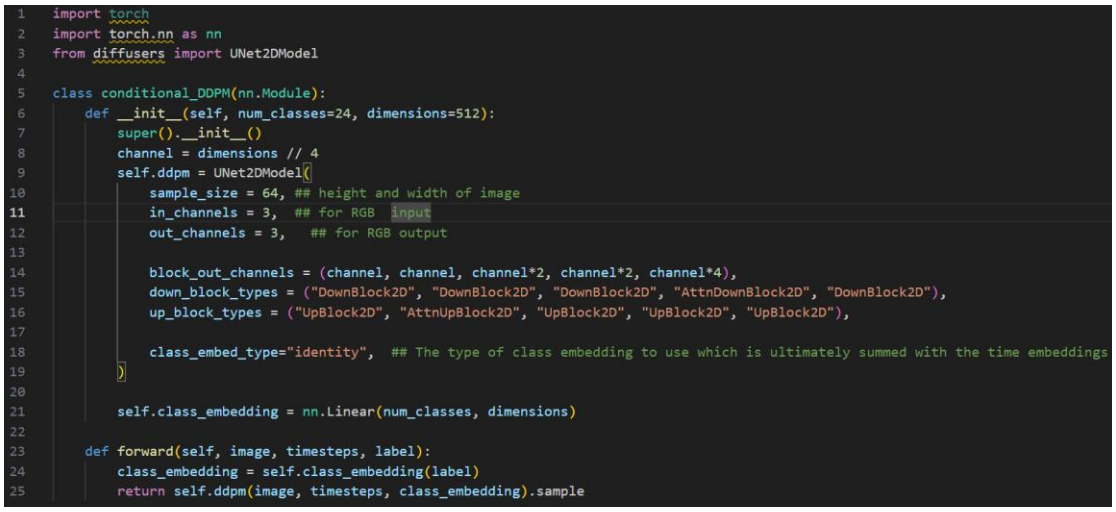
1. Introduction

The purpose of this report is to introduce the method of implementing diffusion model with a useful tool created by "hugging face". At the beginning of the article, detailed implementation will be provided, including the method to create custom dataset and design training and testing loop. In the subsequent paragraph, the analysis of experimental results will also be deliberated over.

2. Implementation detail

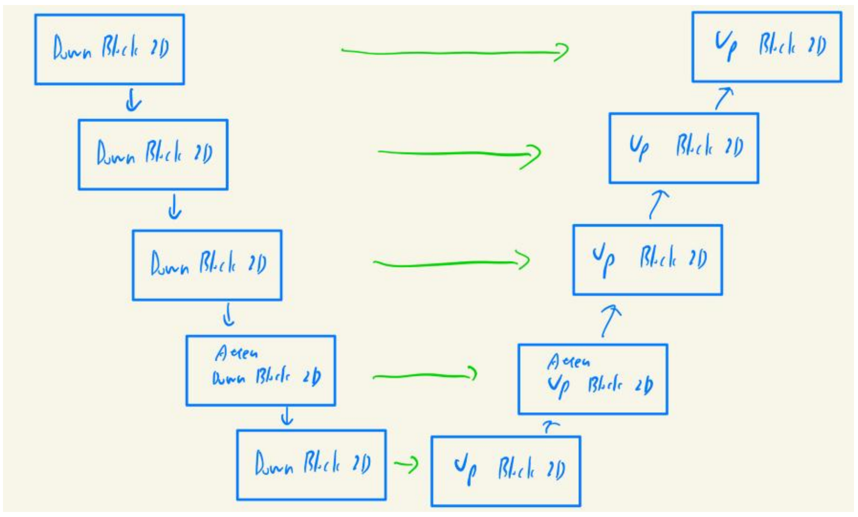
2.1 diffusion_model.py

Starting with "diffusion_model.py", a package called" diffusers" is utilized to create the structure of diffusion model (basically an Unet)[1], providing powerful pretrained model and high flexibility in training DDPM. After defining the number of classes and input dimensions of class_embedding (line 6 in graph 1), "Unet2DModel" is used to create the diffusion model. "block_out_channels" means the block output channels, while "down_block_types" and "up_block_types" mean the block type of down layers and up layers of the model respectively. In terms of class embedding, the type "ideentity" and a linear layer are used because it is the most common combination in designing diffusion model. Something worth mentioning is that the dimension of downsample of Unet need to be at least one forth of the input dimension of "class_embedding" function (line 8 in graph 1). Graph 2 illustrates the outline of the model.
In terms of forward function, the labels from dataset will first be computed by "class_embedding" function, and then be putted into the model.



```
1   import torch
2   import torch.nn as nn
3   from diffusers import UNet2DModel
4
5   class conditional_DDPM(nn.Module):
6       def __init__(self, num_classes=24, dimensions=512):
7           super().__init__()
8           channel = dimensions // 4
9           self.ddpm = UNet2DModel(
10              sample_size = 64, ## height and width of image
11              in_channels = 3,  ## for RGB  input
12              out_channels = 3,   ## for RGB output
13
14              block_out_channels = (channel, channel, channel*2, channel*2, channel*4),
15              down_block_types = ("DownBlock2D", "DownBlock2D", "DownBlock2D", "AttnDownBlock2D", "DownBlock2D"),
16              up_block_types = ("UpBlock2D", "AttnUpBlock2D", "UpBlock2D", "UpBlock2D", "UpBlock2D"),
17
18              class_embed_type="identity",  ## The type of class embedding to use which is ultimately summed with the time embeddings
19          )
20
21          self.class_embedding = nn.Linear(num_classes, dimensions)
22
23      def forward(self, image, timesteps, label):
24          class_embedding = self.class_embedding(label)
25          return self.ddpm(image, timesteps, class_embedding).sample
```

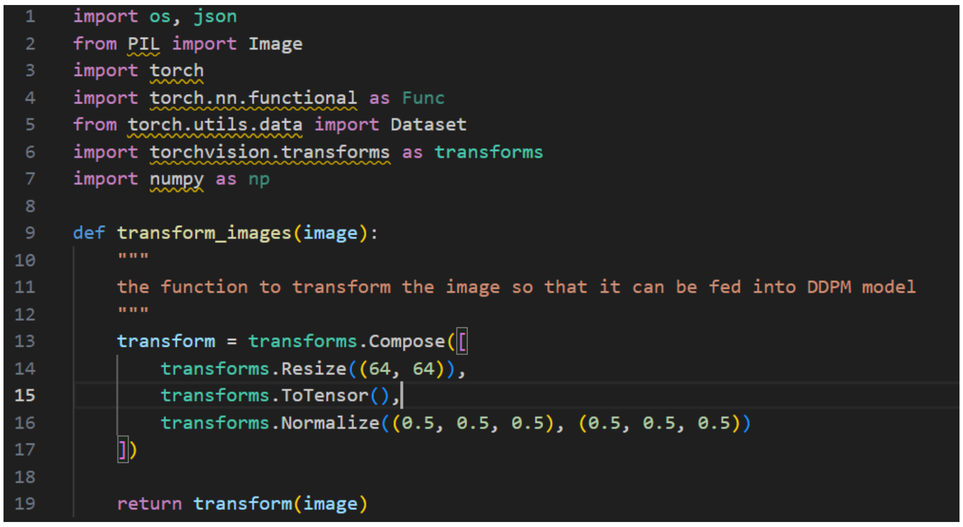
Graph 1. The implementation of "diffusion_model"



Graph 2. Outline of model

2.2 dataset.py

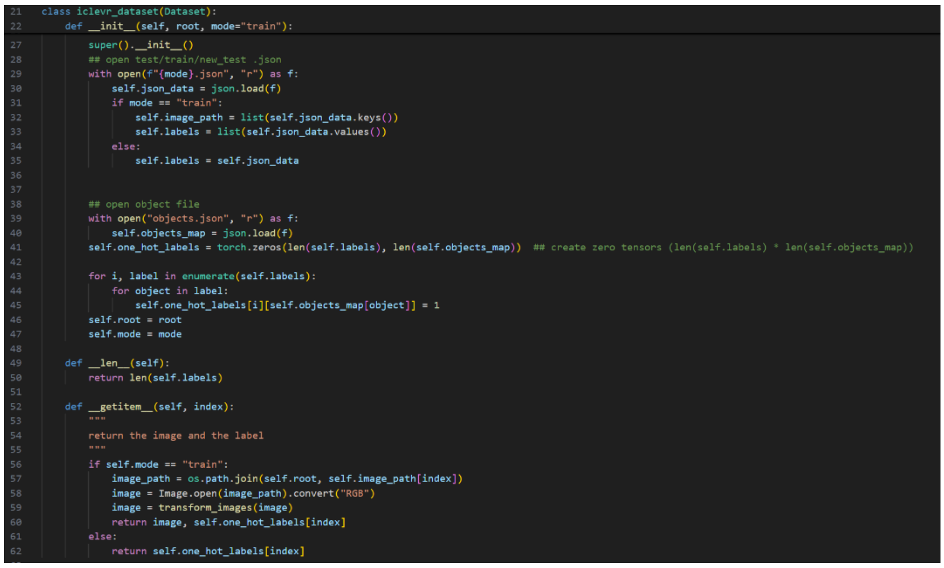
Move on to the method of creating custom dataset, we have to first define our preprocess function (line 9 in graph 3). At first, the image in raw dataset will be transformed into size of 64*64, and then be converted into tensor. Note that the tensor will also be normalized at the end of preprocessing.



```python
1   import os, json
2   from PIL import Image
3   import torch
4   import torch.nn.functional as Func
5   from torch.utils.data import Dataset
6   import torchvision.transforms as transforms
7   import numpy as np
8
9   def transform_images(image):
10      """
11      the function to transform the image so that it can be fed into DDPM model
12      """
13      transform = transforms.Compose([
14          transforms.Resize((64, 64)),
15          transforms.ToTensor(),
16          transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
17      ])
18
19      return transform(image)
```

Graph 3. The implementation of preprocess function

Subsequently, a class called “iclever_dataset” will be created. In training mode, default model of this dataset, the paths of image and labels will be collected, while other mode only collect labels. Thereafter, “object.json” will be opened to create one hot label of the corresponding data (line 39 ~ line 45 in graph 4). When the exterior program want to get the data, the pairs of image and one hot label will be returned in training mode, while only a one hot label will be returned in other mode.

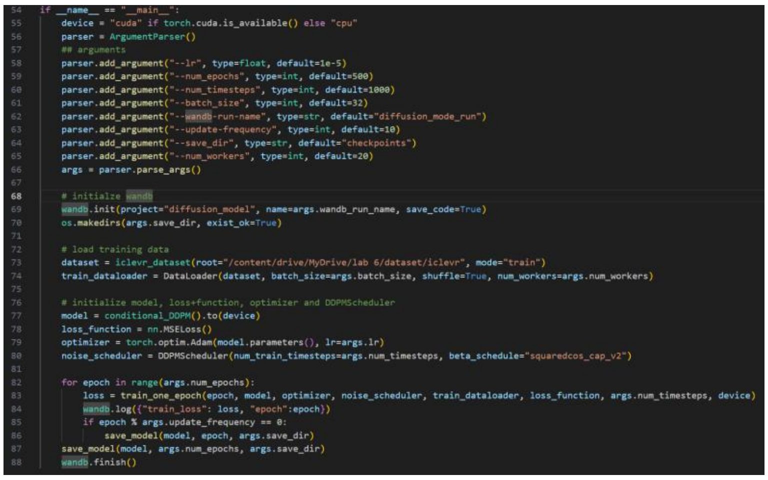


```
21  class iclevr_dataset(Dataset):
22      def __init__(self, root, mode="train"):
27          super().__init__()
28          ## open test/train/new_test .json
29          with open(f"{mode}.json", "r") as f:
30              self.json_data = json.load(f)
31              if mode == "train":
32                  self.image_path = list(self.json_data.keys())
33                  self.labels = list(self.json_data.values())
34              else:
35                  self.labels = self.json_data
36
37
38          ## open object file
39          with open("objects.json", "r") as f:
40              self.objects_map = json.load(f)
41          self.one_hot_labels = torch.zeros(len(self.labels), len(self.objects_map))  ## create zero tensors (len(self.labels) * len(self.objects_map))
42
43          for i, label in enumerate(self.labels):
44              for object in label:
45                  self.one_hot_labels[i][self.objects_map[object]] = 1
46          self.root = root
47          self.mode = mode
48
49      def __len__(self):
50          return len(self.labels)
51
52      def __getitem__(self, index):
53          """
54          return the image and the label
55          """
56          if self.mode == "train":
57              image_path = os.path.join(self.root, self.image_path[index])
58              image = Image.open(image_path).convert("RGB")
59              image = transform_images(image)
60              return image, self.one_hot_labels[index]
61          else:
62              return self.one_hot_labels[index]
```

Graph 4. The implementation of custom dataset

2.3 train.py

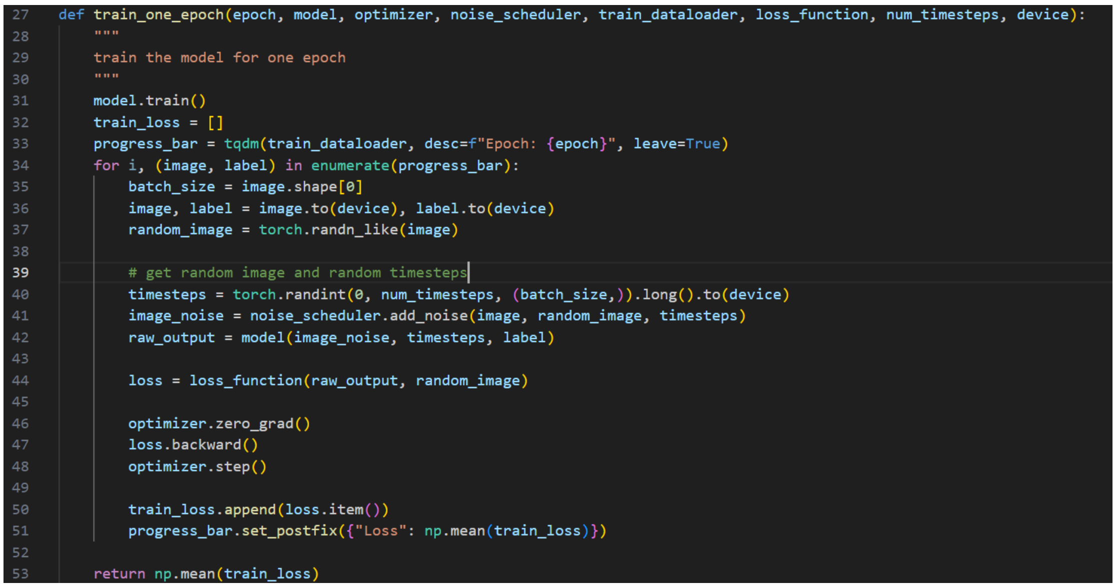
Graph 5 illustrates the implementation of main function in “train.py”. After setting the device, wandb board, and noise_scheduler (I chose DDPMScheduler due to its flexibility and high convenience[2]), the class in “dataset.py” will be generated to create a “dataloader” for training. Subsequently, the training process will begin. Note that after specific epochs the model will be saved (line 85 and line 87 of graph 3).



```
54  if __name__ == "__main__":
55      device = "cuda" if torch.cuda.is_available() else "cpu"
56      parser = ArgumentParser()
57      ## arguments
58      parser.add_argument("--lr", type=float, default=1e-5)
59      parser.add_argument("--num_epochs", type=int, default=500)
60      parser.add_argument("--num_timesteps", type=int, default=1000)
61      parser.add_argument("--batch_size", type=int, default=32)
62      parser.add_argument("--wandb-run-name", type=str, default="diffusion_mode_run")
63      parser.add_argument("--update-frequency", type=int, default=10)
64      parser.add_argument("--save_dir", type=str, default="checkpoints")
65      parser.add_argument("--num_workers", type=int, default=20)
66      args = parser.parse_args()
67
68      # initialze wandb
69      wandb.init(project="diffusion_model", name=args.wandb_run_name, save_code=True)
70      os.makedirs(args.save_dir, exist_ok=True)
71
72      # load training data
73      dataset = iclevr_dataset(root="/content/drive/MyDrive/lab 6/dataset/iclevr", mode="train")
74      train_dataloader = DataLoader(dataset, batch_size=args.batch_size, shuffle=True, num_workers=args.num_workers)
75
76      # initialize model, loss+function, optimizer and DDPMScheduler
77      model = conditional_DDPM().to(device)
78      loss_function = nn.MSELoss()
79      optimizer = torch.optim.Adam(model.parameters(), lr=args.lr)
80      noise_scheduler = DDPMScheduler(num_train_timesteps=args.num_timesteps, beta_schedule="squaredcos_cap_v2")
81
82      for epoch in range(args.num_epochs):
83          loss = train_one_epoch(epoch, model, optimizer, noise_scheduler, train_dataloader, loss_function, args.num_timesteps, device)
84          wandb.log({"train_loss": loss, "epoch":epoch})
85          if epoch % args.update_frequency == 0:
86              save_model(model, epoch, args.save_dir)
87      save_model(model, args.num_epochs, args.save_dir)
88      wandb.finish()
```

Graph 5. Main function of train.py

Graph 6 illustrates the training process of each epoch. A random image will first be created. After obtaining the random timesteps and the noise image, the model will generate a raw output according to the labels (line 42 in graph 6). Afterward, the noise will be added to the random image for training (line 41 in graph 6)[2]. Then the loss will be generated by calculating the disparity between the raw output and the random image. In the end of training, the model will experience backpropagation and optimize step (typical training process). Note that a useful paackage called tqdm is utilized to trace the training process (line 33 in graph 6)[3].

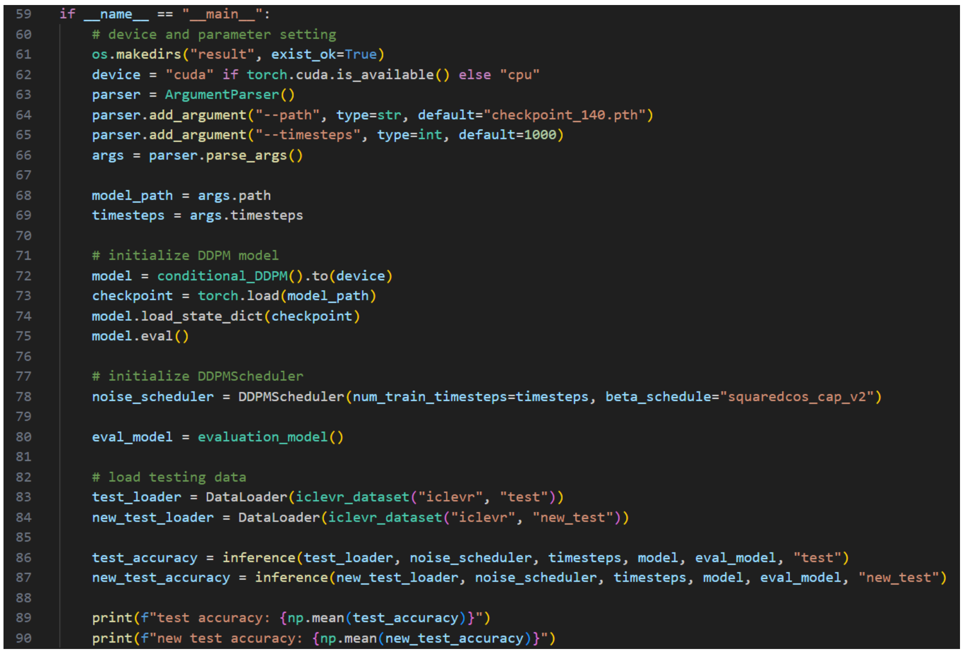


```
27  def train_one_epoch(epoch, model, optimizer, noise_scheduler, train_dataloader, loss_function, num_timesteps, device):
28      """
29      train the model for one epoch
30      """
31      model.train()
32      train_loss = []
33      progress_bar = tqdm(train_dataloader, desc=f"Epoch: {epoch}", leave=True)
34      for i, (image, label) in enumerate(progress_bar):
35          batch_size = image.shape[0]
36          image, label = image.to(device), label.to(device)
37          random_image = torch.randn_like(image)
38
39          # get random image and random timesteps
40          timesteps = torch.randint(0, num_timesteps, (batch_size,)).long().to(device)
41          image_noise = noise_scheduler.add_noise(image, random_image, timesteps)
42          raw_output = model(image_noise, timesteps, label)
43
44          loss = loss_function(raw_output, random_image)
45
46          optimizer.zero_grad()
47          loss.backward()
48          optimizer.step()
49
50          train_loss.append(loss.item())
51          progress_bar.set_postfix({"Loss": np.mean(train_loss)})
52
53      return np.mean(train_loss)
```

Graph 6. Training implementation of each epoch

2.4 test.py

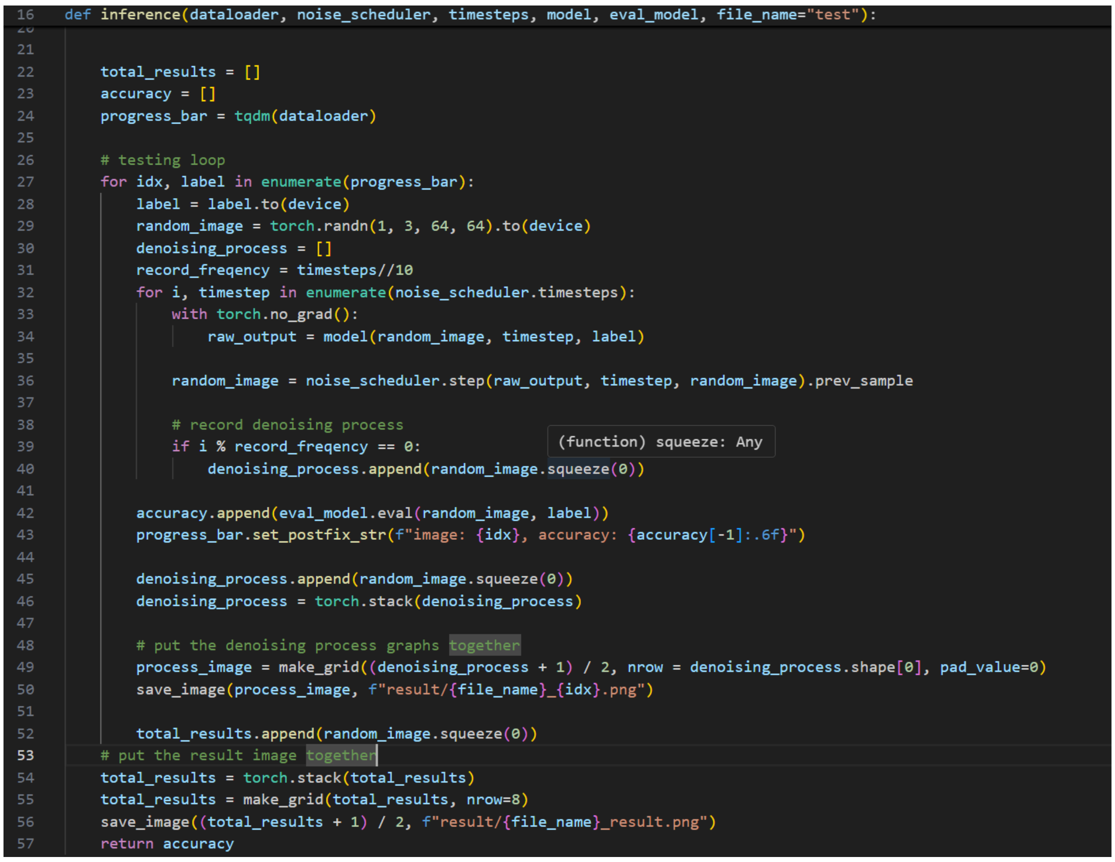
Graph 7 provides the layout of main function of “test.py”. After setting the required parameters, model, and DDPMScheduler, custom dataset of “test.json” as well as “new_test.json” will be loaded for testing. The function called “inference” will return the mean of accuracy of the certain dataset when testing process finish.



```python
if __name__ == "__main__":
    # device and parameter setting
    os.makedirs("result", exist_ok=True)
    device = "cuda" if torch.cuda.is_available() else "cpu"
    parser = ArgumentParser()
    parser.add_argument("--path", type=str, default="checkpoint_140.pth")
    parser.add_argument("--timesteps", type=int, default=1000)
    args = parser.parse_args()

    model_path = args.path
    timesteps = args.timesteps

    # initialize DDPM model
    model = conditional_DDPM().to(device)
    checkpoint = torch.load(model_path)
    model.load_state_dict(checkpoint)
    model.eval()

    # initialize DDPMScheduler
    noise_scheduler = DDPMScheduler(num_train_timesteps=timesteps, beta_schedule="squaredcos_cap_v2")

    eval_model = evaluation_model()

    # load testing data
    test_loader = DataLoader(iclevr_dataset("iclevr", "test"))
    new_test_loader = DataLoader(iclevr_dataset("iclevr", "new_test"))

    test_accuracy = inference(test_loader, noise_scheduler, timesteps, model, eval_model, "test")
    new_test_accuracy = inference(new_test_loader, noise_scheduler, timesteps, model, eval_model, "new_test")

    print(f"test accuracy: {np.mean(test_accuracy)}")
    print(f"new test accuracy: {np.mean(new_test_accuracy)}")
```

Graph 7. The main function of “test.py”

Graph 8 illustrates the implementation of function called “inference”. After setting the required variables, a random image will be generated at the beginning of testing process. For each timestep, the noise_scheduler will generate aa new denoised image according to the prediction outputted by the model (line 36 of graph 8). Note that the image will be recorded every 100 timesteps to visualize the denoising process. At the end of testing process, make_grid()[4] function is used to generate the image of denoising process. Something worth mentioning is that the variable, denoising process, should be add 1 and divided by 2 to make the number from [-1,1] to (0,1]. The reason to do that is because of the input format of function “save_image”[5], or it might report an error.



```python
def inference(dataloader, noise_scheduler, timesteps, model, eval_model, file_name="test"):

    total_results = []
    accuracy = []
    progress_bar = tqdm(dataloader)

    # testing loop
    for idx, label in enumerate(progress_bar):
        label = label.to(device)
        random_image = torch.randn(1, 3, 64, 64).to(device)
        denoising_process = []
        record_freqency = timesteps//10
        for i, timestep in enumerate(noise_scheduler.timesteps):
            with torch.no_grad():
                raw_output = model(random_image, timestep, label)

            random_image = noise_scheduler.step(raw_output, timestep, random_image).prev_sample

            # record denoising process
            if i % record_freqency == 0:
                denoising_process.append(random_image.squeeze(0))

        accuracy.append(eval_model.eval(random_image, label))
        progress_bar.set_postfix_str(f"image: {idx}, accuracy: {accuracy[-1]:.6f}")

        denoising_process.append(random_image.squeeze(0))
        denoising_process = torch.stack(denoising_process)

        # put the denoising process graphs together
        process_image = make_grid((denoising_process + 1) / 2, nrow = denoising_process.shape[0], pad_value=0)
        save_image(process_image, f"result/{file_name}_{idx}.png")

        total_results.append(random_image.squeeze(0))
    # put the result image together
    total_results = torch.stack(total_results)
    total_results = make_grid(total_results, nrow=8)
    save_image((total_results + 1) / 2, f"result/{file_name}_result.png")
    return accuracy
```

Graph 8. The implementation of testing loop

3. Result & Discussion

3.1 Results

The results of the experiment are showed in table 1. According to graph 9, score of "test" and "new_test" are 0.911 and 0.927 respectively. In addition, graph 10 depicts the training curve, we can observe that at first the loss will decrease drastically, and the figure remains stable at the end of training.

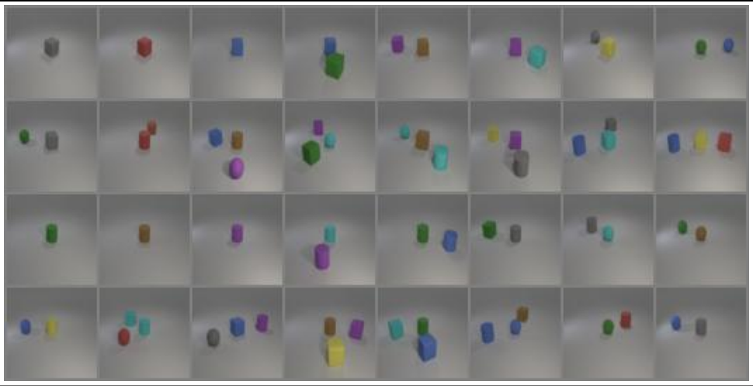
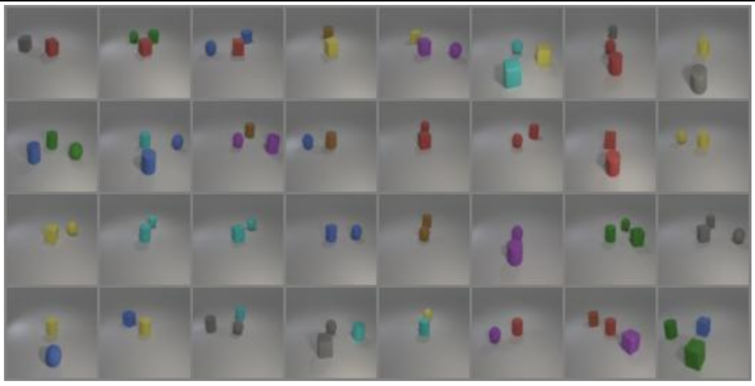
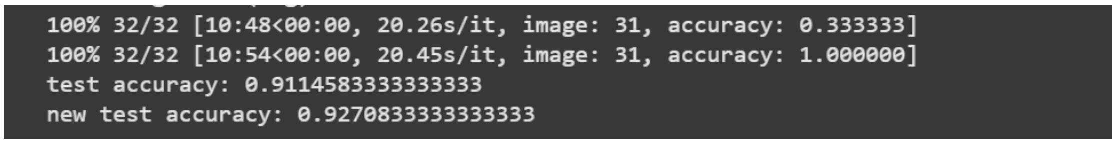
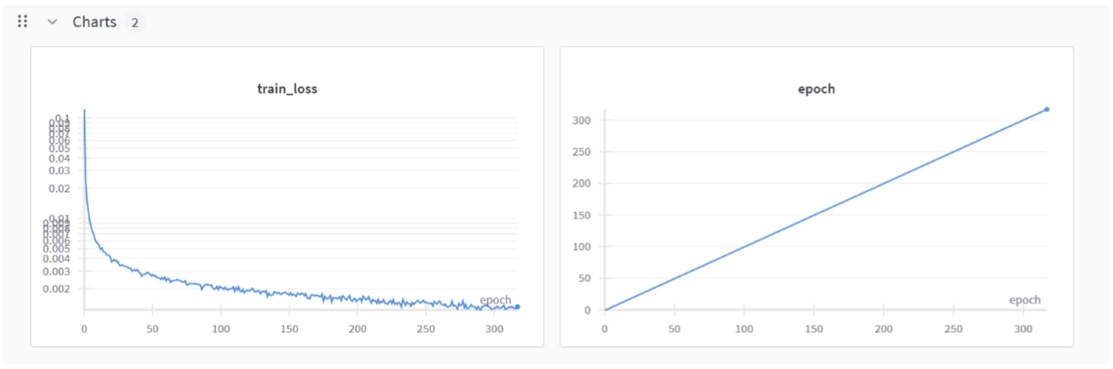
Test	
New_test	

Table 1. Testing result



```
100% 32/32 [10:48<00:00, 20.26s/it, image: 31, accuracy: 0.333333]
100% 32/32 [10:54<00:00, 20.45s/it, image: 31, accuracy: 1.000000]
test accuracy: 0.9114583333333333
new test accuracy: 0.9270833333333333
```

Graph 9. Accuracy



Graph 10. Training curve

Table 2 illustrates the output of label set, ["red sphere", "cyan cylinder", "cyan cube"], with the denoising process.

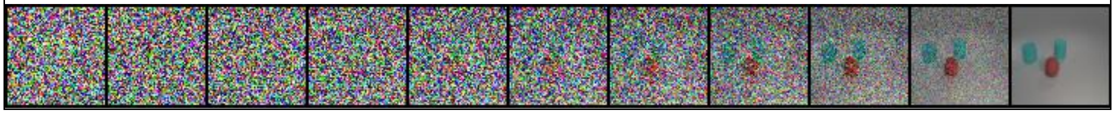
red sphere, cyan cylinder, cyan cube


Table 2. The denoising process of the label set

3.2 extra experiments and discussion

In order to investigate how the structure of models affect the result, an experiment was designed to discover the difference. The model of the experiment above is 5 layers model including 1 layer of attention layer. Table 3 illustrates the difference, including the size of .pth file. Note that the numbers of each epoch are set to 50 in training.

	Score in test	Score in new_test	file size
3 layers model	0.71875	0.74479	65.9 MB
3 layers model including 1 attention layer	0.57813	0.73958	71 MB
4 layers model	0.64063	0.70833	103 MB
4 layers model including 1 attention layer	0.76042	0.84375	108 MB
5 layers model	0.67708	0.76563	267 MB
5 layers model including 1 attention layer	0.69791	0.76042	281 MB

Table 3. the difference of performance with regard to 6 types of model

According to table 3, 4 layers model including 1 attention layer might have the best performance after training for 50 epochs. However, 3 layers model is the most efficient one because its performance is good enough with a smaller number of weights in comparison with 4 layers model.

4. References

[1]: https://huggingface.co/docs/diffusers/api/models/unet2d
[2]: https://huggingface.co/docs/diffusers/api/schedulers/ddpm
[3]: https://tqdm.github.io/docs/tqdm/
[4]:
https://pytorch.org/vision/main/generated/torchvision.utils.make_grid.html
[5]:
https://pytorch.org/vision/main/generated/torchvision.utils.save_image.html
[6]: https://colab.research.google.com/github/huggingface/diffusion-models-class/blob/main/unit1/01_introduction_to_diffusers.ipynb
[7]: https://colab.research.google.com/github/huggingface/diffusion-models-class/blob/main/unit1/02_diffusion_models_from_scratch.ipynb
[8]: https://huggingface.co/docs/diffusers/v0.3.0/en/api/schedulers