Lab6 - Generative Models

313551098

Report (60%)

Introduction (5%)

The objective of this project is to implement an image generation system based on a Conditional Diffusion Model (DDPM). Diffusion Models are generative models that progressively add noise and generate new images through a denoising process.

Specifically, we will implement a custom dataloader to handle the training and testing data. This class will pair images with their corresponding labels and generate one-hot encoded labels. We will also implement a Conditional DDPM model, which takes input labels into account during the image generation process to achieve conditional generation. Furthermore, we will train this model and evaluate its generative capability using test data.

In the implementation process, I referred to the basic tutorial on diffusion models provided by Hugging Face, and combined its principles to complete the custom model and data processing implementation.

Implementation details (25%)

- Describe how you implement your model, including your choice of

  DDPM, noise schedule.

  #### Model Architecture

  In this project, my model implementation is based on Hugging Face's `diffusers` and is inspired by the basic model architecture provided in their official tutorial. My model uses a standard U-Net architecture, `UNet2DModel`, which is commonly used for image generation and denoising tasks. The architectural parameters I selected for the model are as follows:

  - **Input image size**: 64x64 pixels, which is a typical size that allows effective model training.
  - **Input and output channels**: 3 channels, corresponding to RGB images.
  - **Number of layers per U-Net block**: Each block contains 2 ResNet layers, enhancing the model's expressive capacity.
  - **Number of channels per block**: We chose progressively increasing channels (128, 128, 256, 256, 512, 512) to capture richer features in the deeper layers of the model.

- **Downsampling and upsampling blocks**: We used standard 2D downsampling and upsampling blocks and incorporated self-attention mechanisms in the middle layers to improve the consistency and quality of the generated images.

In the conditional processing part, a fully connected layer embeds the class labels into vectors that match the model's dimensionality. These embeddings, along with the time step, are then passed into the U-Net model to enable conditional generation.

```python
from diffusers import UNet2DModel
import torch
import torch.nn as nn

# This code is inspired by the Hugging Face Diffusers tutorial:
# https://huggingface.co/docs/diffusers/tutorials/basic_training
class conditionalDDPM(nn.Module):
    def __init__(self, num_classes = 24, dim = 512):
        super().__init__()

        # Initialize a UNet2DModel, a generic U-Net model from the
diffusers library for denoising and image generation
        self.ddpm = UNet2DModel(
            sample_size = 64, # size of input images 64*64
            in_channels = 3, # RGB
            out_channels = 3,
            layers_per_block = 2, # Number of ResNet layers in each U-
Net block
            block_out_channels = [128, 128, 256, 256, 512, 512], # the
number of output channels for each UNet block

            # Standard 2D downsampling block
            down_block_types=[
                "DownBlock2D",
                "DownBlock2D",
                "DownBlock2D",
                "DownBlock2D",
                "AttnDownBlock2D",
                "DownBlock2D"
            ],

            # Standard 2D upsampling block
            up_block_types=[
                "UpBlock2D",
                "AttnUpBlock2D",
                "UpBlock2D",
                "UpBlock2D",
                "UpBlock2D",
                "UpBlock2D"
            ],

            # Use an identity function for class embedding (no
transformation)
            class_embed_type="identity",
        )

        # Define a fully connected layer to embed class labels into
vectors matching the model dimension
        self.class_embedding = nn.Linear(num_classes, dim)

    def forward(self, x, t, label):

        # Convert class labels into embedding vectors matching the
model dimensions
        class_embed = self.class_embedding(label)

        # Return the denoised image
        return self.ddpm(x, t, class_embed).sample
```

Noise Schedule

When training this conditional diffusion model, we chose to use `squaredcos_cap_v2` as the noise scheduling strategy. This strategy is based on a squared cosine function, which smoothly controls the addition and removal of noise during the diffusion process. Specifically, the `squaredcos_cap_v2` strategy gradually increases noise during the early time steps and then transitions smoothly in the later time steps. This helps the model learn the denoising process more stably, ultimately improving the quality of the generated images.

The noise scheduler is implemented through `DDPMScheduler`, which is responsible for generating random noise and adding it to the original images during training. This process simulates the actual diffusion process, and the model needs to learn how to remove this noise at each time step to eventually produce outputs that resemble the original images.

Training

Parameter Settings: epoch = 300 / learning rate = 1e-4 / Noise Scheduler = squaredcos_cap_v2

1. Model Initialization and Optimizer Setup

   Before starting the training, the `conditionalDDPM` model was initialized. This model is based on the U-Net architecture and takes conditional labels to generate corresponding images. I used `DDPMScheduler` to control the addition and removal of noise during the diffusion process. To optimize the model, I chose the Adam optimizer and set the learning rate to 0.0001 to ensure stability during training.

2. Training Process (Denoising Process)

   During each training epoch, the model implements the denoising process through the following steps:

- **Noise Addition**: First, for each input image, a time step is randomly selected, and the noise scheduler is used to generate random noise based on this time step. This noise is added to the original image to simulate the noise disturbance in the diffusion process.
- **Forward Propagation**: The noisy image, along with the time step and conditional label, is input into the model. The model attempts to predict and remove the added noise based on the input conditional label and time step, thereby restoring the image to its original state. Here, the embedding vector of the conditional label plays a crucial role in guiding the model to generate an image that matches the given label at each step of the denoising process.
- **Loss Calculation**: The quality of denoising is evaluated through a loss function. We used Mean Squared Error (MSE Loss) to calculate the difference between the denoised image output by the model and the actual noise. The smaller this loss value, the better the model's denoising effect, and the closer the generated image is to the original noise-free image.
- **Model Update**: Based on the calculated loss value, backpropagation is performed, and the model's weights are updated. This process helps the model gradually learn how to denoise more accurately and generate high-quality images.

3. Training Results

   During the training process, the model parameters were continuously adjusted over multiple epochs, allowing the model to more precisely generate images that match the conditional labels during the step-by-step denoising process. As training progressed, the model gradually learned how to effectively denoise and generate realistic images at each time step. The model's state was saved at regular intervals of epochs to be used later for inference or further training.

```


def training_stage(args):
    train_loader = train_dataloader(args)

    model = conditionalDDPM().to(args.device)
    noise_scheduler = 
DDPMScheduler(num_train_timesteps=args.total_timesteps, 
beta_schedule=args.beta_schedule)
    optimizer = torch.optim.Adam(model.parameters(), lr=args.lr)

    experiment_name = f"lr_{args.lr}_{args.num_epoch}"
    writer = tensorboard.SummaryWriter(log_dir=f"runs/{experiment_name}")

    if args.ckpt_path != None:
        load_checkpoint(model, optimizer, args.checkpoint)

    for epoch in range(args.num_epoch):
        loss = training_one_step(epoch, model, optimizer, train_loader,
noise_scheduler, args.total_timesteps, args.device)
        writer.add_scalar('Loss/train', loss, epoch)

        if epoch % args.per_save == 0:
            save(model, optimizer, epoch, os.path.join(args.save_root,
f"epoch={epoch}.pth"))

        print(f"Epoch [{epoch}/{args.num_epoch}], Training Loss:
{loss:.4f}")

    save(model, optimizer, args.num_epoch, os.path.join(args.save_root,
f"epoch={args.num_epoch}.pth"))

def training_one_step(epoch, model, optimizer, train_loader,
noise_scheduler, total_timesteps, device):
    model.train()
    mse_loss = nn.MSELoss()
    train_losses = []

    progress_bar = tqdm(train_loader, desc=f'Epoch: {epoch}', leave=True)

    for i, (x, label) in enumerate(progress_bar):
        batch_size = x.shape[0]
        x, label = x.to(device), label.to(device)
        noise = torch.randn_like(x)

        timesteps = get_random_timesteps(batch_size, total_timesteps,
device)
        noisy_x = noise_scheduler.add_noise(x, noise, timesteps)
        output = model(noisy_x, timesteps, label)

        loss = mse_loss(output, noise)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        train_losses.append(loss.item())
        progress_bar.set_postfix({'Loss': np.mean(train_losses)})

    return np.mean(train_losses)

def train_dataloader(args):
    transform = transforms.Compose([
        transforms.Resize((64, 64)),
        transforms.ToTensor(),
        transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])
    dataset = iclevrDataset(root='iclevr', mode='train',
transform=transform)

    train_loader = DataLoader(dataset,
                              batch_size=args.batch_size,
                              num_workers=args.num_workers,
                                drop_last=True,
                                shuffle=False)
    return train_loader


def save(model, optimizer, current_epoch, path):
    torch.save({
        "state_dict": model.state_dict(),
        "optimizer": optimizer.state_dict(),
        "last_epoch": current_epoch
    }, path)
    print(f"save ckpt to {path}")

def load_checkpoint(model, optimizer, path):
    model.load_state_dict(torch.load(path)['state_dict'])
    optimizer.load_state_dict(torch.load(path)['optimizer'])

def get_random_timesteps(batch_size, total_timesteps, device):
    return torch.randint(0, total_timesteps,
(batch_size,)).long().to(device)

def main(args):
    os.makedirs(args.save_root, exist_ok=True)
    training_stage(args)
```

Testing

1. Model Loading and Initialization

   Before starting the testing, we first loaded the pre-trained model checkpoint. We used the `load_checkpoint` function to load the model file from the specified path and load it into the `conditionalDDPM` model. The model was set to evaluation mode (`eval`) to ensure that no gradient updates occur during the testing process, thereby maintaining the stability of the model parameters.

2. Noise Scheduling and Initialization

   Similar to the training process, we used the `squaredcos_cap_v2` noise scheduling strategy during testing. This strategy smoothly adjusts the noise intensity during the step-by-step denoising process, helping the model better restore the image. In testing, the noise scheduler is responsible for generating and removing noise according to the preset number of time steps (`total_timesteps`), and progressively guiding the denoising results closer to the final image at each time step.

3. Test Data Loading

I used the `test_dataloader` function to load the test data. This function loads the corresponding test datasets depending on the mode (such as `test` or `new_test`). Each test image is loaded into the model along with its corresponding conditional label, which is used to generate an image that matches the condition.

4. Denoising Process

   During the testing process, I generate random noise for each test image and perform step-by-step denoising based on the specified number of time steps. The process is as follows:

   - **Random Noise Generation**: For each test sample, we first generate a random noise image as the starting point, simulating the process of generating an image from pure noise in a real-world application.
   - **Step-by-Step Denoising**: The model predicts and removes portions of the noise at each time step `t` according to the given conditional label, ultimately producing a result that is close to a real image. During this process, the model leverages the knowledge learned during training to gradually reduce noise and generate an image that matches the conditional label.
   - **Result Saving**: During the denoising process, we save the intermediate results and visualize the images at each time step. The final generated image is saved to the specified path for subsequent analysis and display.

5. Accuracy Evaluation

   During the testing process, we used the `evaluation_model` provided by TA to evaluate the accuracy of the generated images. Each generated image is compared with its conditional label, and an accuracy score is calculated. These accuracy scores are accumulated and averaged at the end of the testing process, helping us quantify the model's performance.

6. Testing Results

   The testing process was conducted on both the `test` and `new_test` datasets, and the average accuracy for both datasets was reported separately.

```
def set_seed(seed):
    """Set random seed for reproducibility."""
    random.seed(seed)
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False


def inference(args, model, mode='test'):
    os.makedirs(os.path.join(args.save_root, mode), exist_ok=True)
    device = args.device
    timesteps = args.total_timesteps
    dataloader = test_dataloader(args, mode)
    noise_scheduler = DDPMScheduler(num_train_timesteps=timesteps,
beta_schedule=args.beta_schedule)
    evaluator = evaluation_model()

    results = []
    accuracy_list = []

    progress_bar = tqdm(dataloader)

    for idx, label in enumerate(progress_bar):
        label = label.to(device)
        img = torch.randn(1, 3, 64, 64).to(device)
        denoising_step = []

        for i, t in enumerate(noise_scheduler.timesteps):
            with torch.no_grad():
                residual = model(img, t, label)

            img = noise_scheduler.step(residual, t, img).prev_sample #
img denoising
            if i % (timesteps // 10) == 0:
                denoising_step.append(img.squeeze(0))

        accuracy = evaluator.eval(img, label)
        accuracy_list.append(accuracy)

        progress_bar.set_postfix_str(f'image: {idx}, accuracy:
{accuracy:.4f}')

        denoising_step.append(img.squeeze(0))
        denoising_step = torch.stack(denoising_step)

        row_image = make_grid((denoising_step + 1) / 2,
nrow=denoising_step.shape[0], pad_value=0)
        save_image(row_image, f'result/{mode}/{mode}_{idx}.png')
#pixel from [-1, 1] to [0, 1]

        results.append(img.squeeze(0))

    results = torch.stack(results)
    results = make_grid(results, nrow=8, pad_value=-1)
    save_image((results + 1) / 2, f'result/{mode}_result.png')

    return np.mean(accuracy_list)


def test_dataloader(args, mode="test"):
    dataset = iclevrDataset(root='iclevr', mode=mode)

    test_loader = DataLoader(dataset,
                             batch_size=args.batch_size,
                                  num_workers=args.num_workers,
                                  drop_last=True,
                                  shuffle=False)
    return test_loader

def load_checkpoint(args):
     if args.ckpt_path != None:
        model = conditionalDDPM().to(args.device)
        checkpoint = torch.load(args.ckpt_path, weights_only=True)
        model.load_state_dict(checkpoint['state_dict'], strict=True)
        model.eval()

        return model

def main(args):
    os.makedirs(args.save_root, exist_ok=True)
    set_seed(40)

    model = load_checkpoint(args)
    test_accuracy = inference(args, model, mode='test')
    print(f'test accuracy: {test_accuracy}')

    new_test_accuracy = inference(args, model, mode='new_test')
    print(f'new test accuracy: {new_test_accuracy}')
```

Dataloader

In this project, we implemented a dataset class called `iclevrDataset`, which is used to handle the data required for training and testing. This dataset class is responsible for loading image data and their corresponding labels and converting them into a format that can be used by the model.

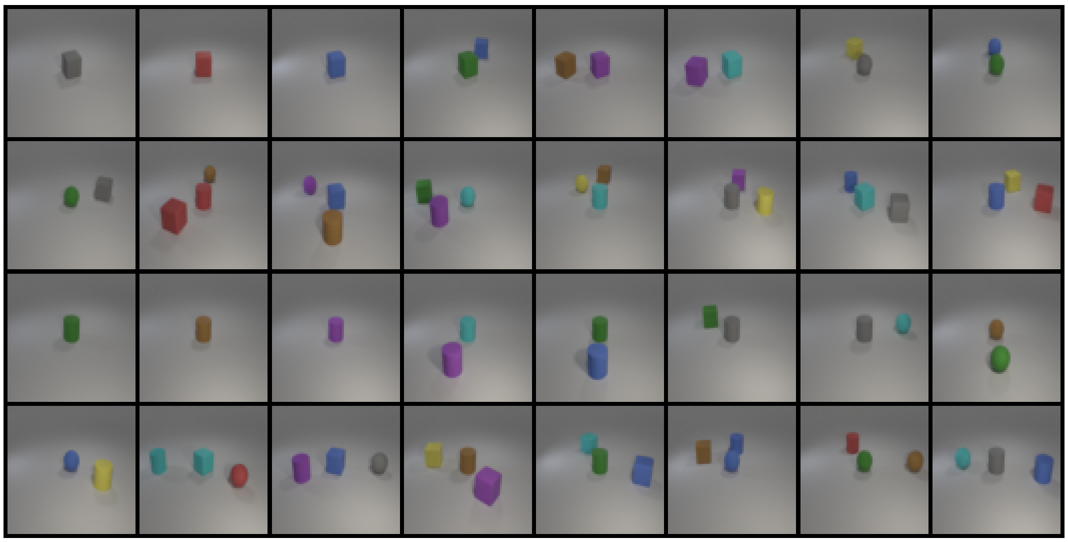
First, we load the data by initializing the `iclevrDataset` class. This class loads the appropriate JSON files based on the specified mode (`train`, `test`, or `new_test`). These files contain the image filenames and their corresponding labels. Through the `objects.json` file, we load a dictionary that contains all object labels and their corresponding indices. This dictionary is used in subsequent steps to convert the labels into one-hot encodings.

For the data in training mode, we load the images from the dataset and convert them to RGB format. Then, we apply a series of image transformations (`transforms`), including resizing the images, converting them to tensors, and performing normalization. These preprocessing steps help improve the effectiveness and stability of the model during training.

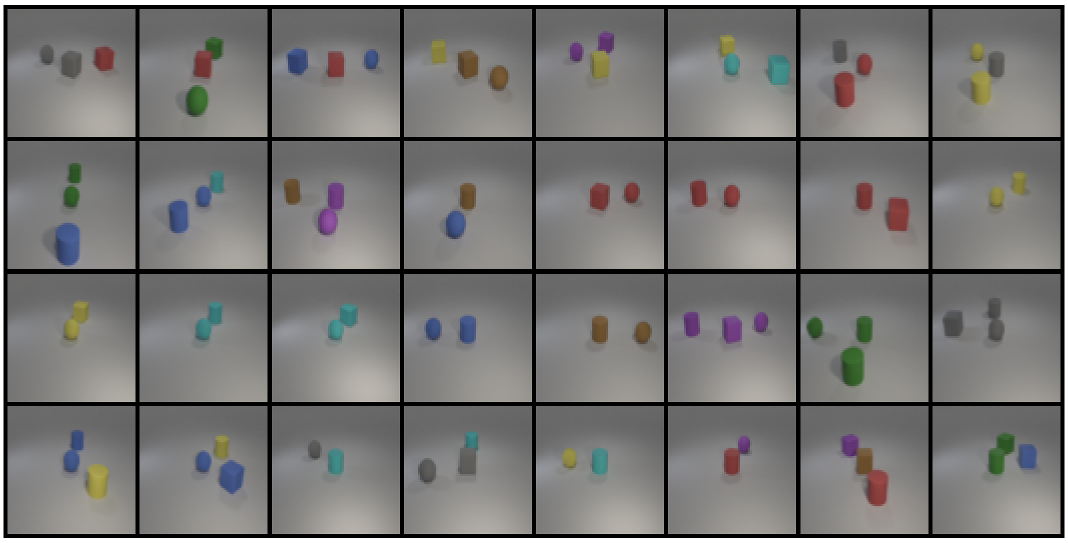
```
class iclevrDataset(Dataset):
    def __init__(self, root=None, mode="train", transform=None):
        super().__init__()
        assert mode in ["train", "test", "new_test"], "Invalid mode
specified!"

        self.root = root
        self.mode = mode
        self.transform = transform

        self.json_data = self._load_json(f'{self.mode}.json')
        self.objects_dict = self._load_json('objects.json')

        if self.mode == "train":
            self.img_paths, self.labels = list(self.json_data.keys()),
list(self.json_data.values())
        elif self.mode == "test" or self.mode == "new_test":
            self.labels = self.json_data

        self.labels_one_hot = torch.zeros(len(self.labels),
len(self.objects_dict)) #[samples, class]

        for i, label_list in enumerate(self.labels):
            # For each label in each label list, get the corresponding
index
            label_indices = []
            for label in label_list:
                index = self.objects_dict[label]
                label_indices.append(index)

            # Set the one-hot position corresponding to these indexes to 1
            self.labels_one_hot[i][label_indices] = 1

    def _load_json(self, filepath):
        """Helper function to load JSON files."""
        with open(filepath, 'r') as json_file:
            return json.load(json_file)

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

    def __getitem__(self, index):
        if self.mode == 'train':
            img_path = os.path.join(self.root, self.img_paths[index])
            img = Image.open(img_path).convert('RGB')
            img = self.transform(img)
            label_one_hot = self.labels_one_hot[index]
            return img, label_one_hot

        elif self.mode == "test" or self.mode == "new_test":
            label_one_hot = self.labels_one_hot[index]
            return label_one_hot
```

Results and discussion (30%)

- Show your synthetic image grids (total 16%: 8% * 2 testing data) and a denoising process image (4%)
    - `test.json`

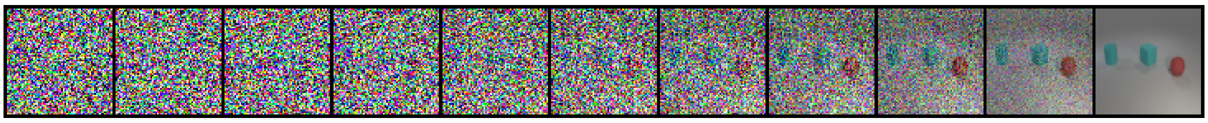


    - `new_test.json`



    - denoising process image

      with the label set ["red sphere", "cyan cylinder", "cyan cube"]



- Discussion of your extra implementations or experiments (10%)

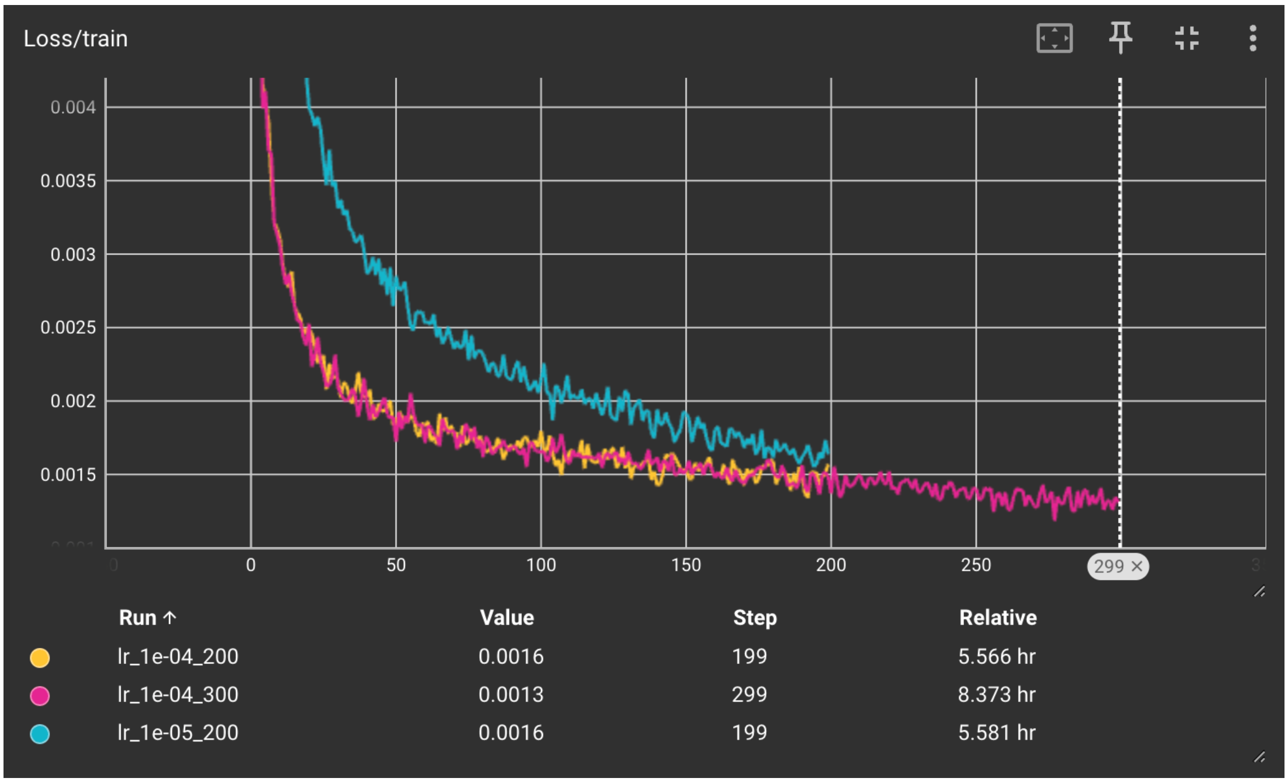
  During the training, I tried three different parameter settings, which are as follows:

  learning rate = 1e-4 / epoch = 200

  learning rate = 1e-4 / epoch = 300

learning rate = 1e-5 / epoch = 200

As seen in the following image, after training for 300 epochs with a learning rate set to 1e-4, the results achieved were the best.



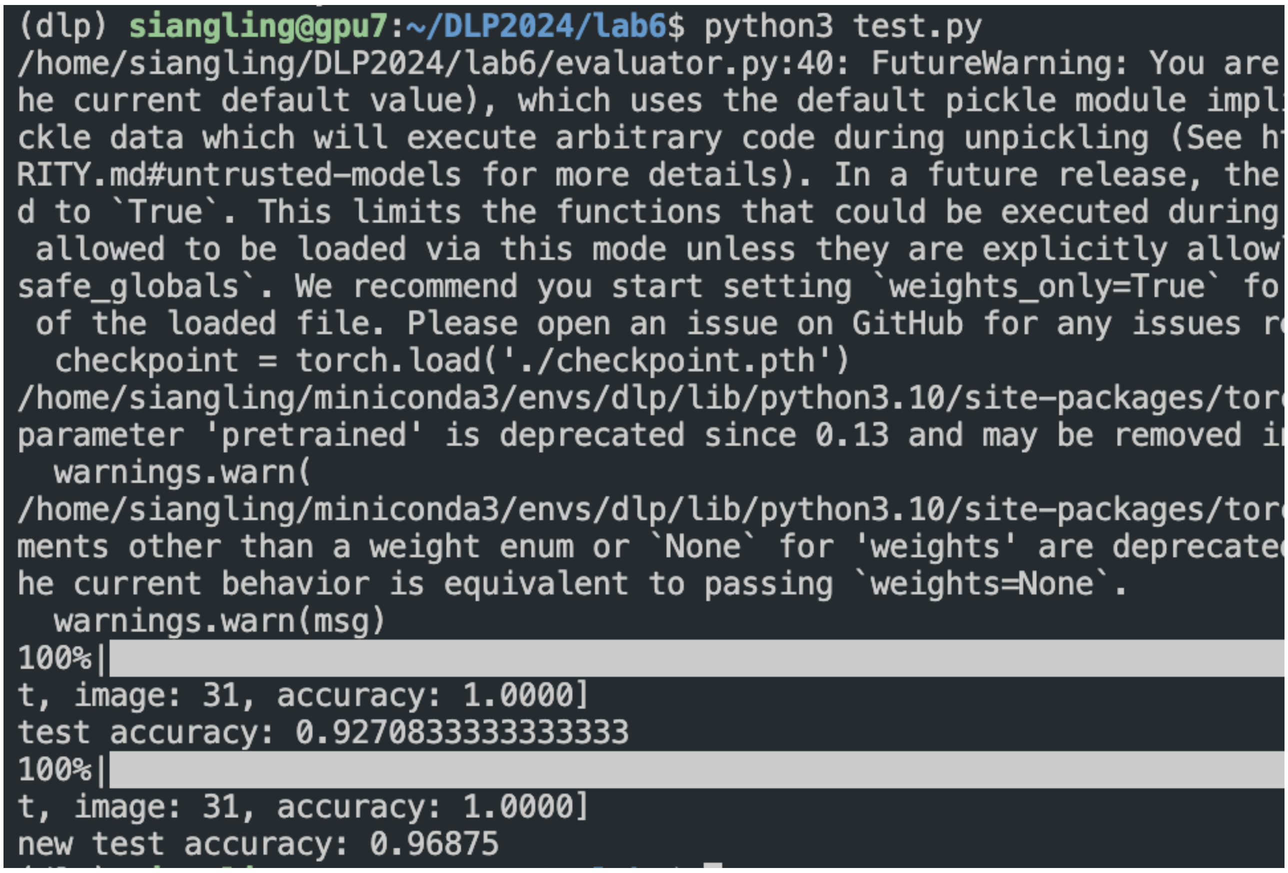
	Run ↑	Value	Step	Relative
●	lr_1e-04_200	0.0016	199	5.566 hr
●	lr_1e-04_300	0.0013	299	8.373 hr
●	lr_1e-05_200	0.0016	199	5.581 hr

Experimental results (40%) (based on results shown in your report)

- Show your accuracy screenshots

  "test.json" : 0.9270833333333333

"new_test.json" : 0.96875



```
(dlp) siangling@gpu7:~/DLP2024/lab6$ python3 test.py
/home/siangling/DLP2024/lab6/evaluator.py:40: FutureWarning: You are
he current default value), which uses the default pickle module impl
ckle data which will execute arbitrary code during unpickling (See h
RITY.md#untrusted-models for more details). In a future release, the
d to `True`. This limits the functions that could be executed during
 allowed to be loaded via this mode unless they are explicitly allow
safe_globals`. We recommend you start setting `weights_only=True` fo
 of the loaded file. Please open an issue on GitHub for any issues r
  checkpoint = torch.load('./checkpoint.pth')
/home/siangling/miniconda3/envs/dlp/lib/python3.10/site-packages/tor
parameter 'pretrained' is deprecated since 0.13 and may be removed i
  warnings.warn(
/home/siangling/miniconda3/envs/dlp/lib/python3.10/site-packages/tor
ments other than a weight enum or `None` for 'weights' are deprecate
he current behavior is equivalent to passing `weights=None`.
  warnings.warn(msg)
100%|
t, image: 31, accuracy: 1.0000]
test accuracy: 0.9270833333333333
100%|
t, image: 31, accuracy: 1.0000]
new test accuracy: 0.96875
```

- The command for inference process for both testing data

  1. prepare data Following is the directory structure :

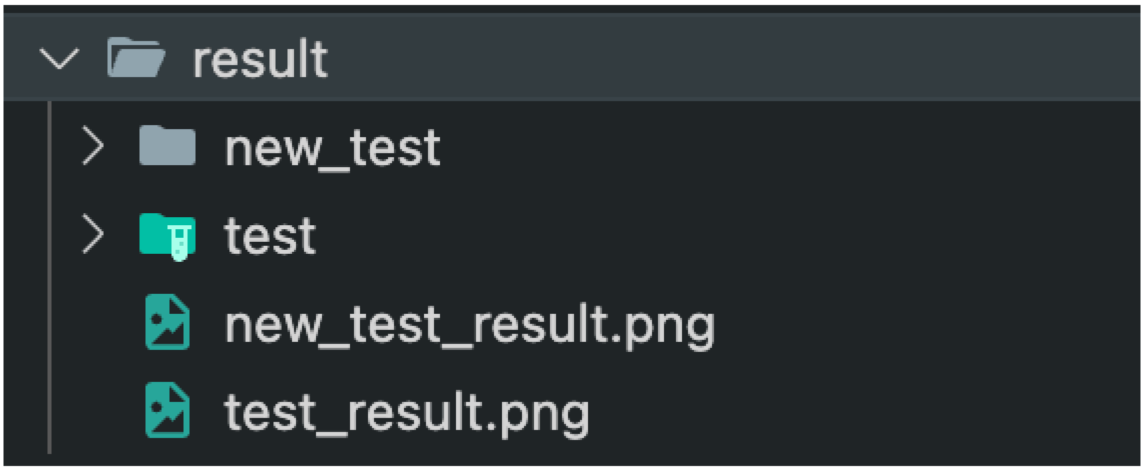
```
DL_LAB6_313551098_□□□/
|── lab6/
|    ├── checkpoints/
|    |    └── (put ddpm checkpoint "DL_lab6_313551098_□□□.pth")
|    ├── model/
|    |    └── DDPM.py
|    |
|    ├── (put evaluator checkpoint "checkpoint.pth")
|    ├── evaluator.py
|    |
|    ├── dataloader.py
|    ├── train.py
|    ├── test.py
|    |
|    ├── train.json
|    ├── objects.json
|    ├── test.json
|    ├── new_test.json
|
└── Report.pdf
```

  2. command

```
cd lab6
```

```
python3 test.py --ckpt_path checkpoints/DL_lab6_313551098_□□□.pth
```

The above command will run both "test.json" and "new_test.json" simultaneously, and the resulting folders will look like the image below:



```
result
  new_test
  test
  new_test_result.png
  test_result.png
```

Parameters that may need to be modified:

- --ckpt_path :

  ```
  The path of model

  default="checkpoints/DL_lab6_313551098_□□□.pth"
  ```

- --save_root :

  ```
  The path to save your data

  default="result"
  ```

- --device :

  ```
  default="cuda:0"
  ```