

## Report

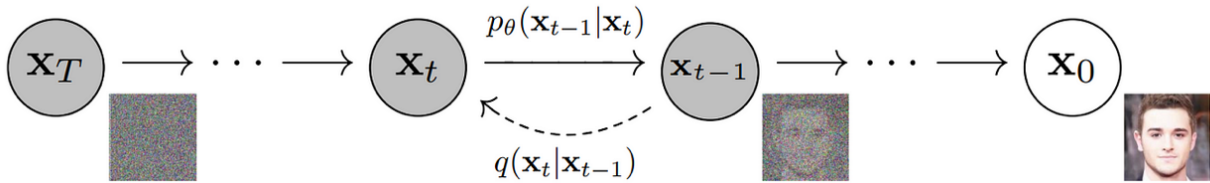
### Introduction

The DDPM model is a state-of-the-art generative model. It generates a picture by sampling from Gaussian noise and applying a denoising process step by step.

In the training stage, we sample some data from the dataloader. Next, we generate a sequence of  $t$  using a uniform distribution. After that, we sample Gaussian noise and compute the  $x_{t+1}$  data. This Gaussian noise is utilized as the ground truth for learning and evaluating whether the model has learned the denoising process.

In the sampling stage, we begin with a Gaussian noise and utilize our model to predict and remove the noise step by step. Eventually, we obtain a clear picture.

In this lab, we need to train a diffusion model and use iclevr dataset to generate the given shape written in test.json and new\_test.json.



#### Algorithm 1 Training

```

1: repeat
2:    $\mathbf{x}_0 \sim q(\mathbf{x}_0)$ 
3:    $t \sim \text{Uniform}(\{1, \dots, T\})$ 
4:    $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ 
5:   Take gradient descent step on
      $\nabla_\theta \|\epsilon - \epsilon_\theta(\sqrt{\alpha_t}\mathbf{x}_0 + \sqrt{1 - \alpha_t}\epsilon, t)\|^2$ 
6: until converged

```

#### Algorithm 2 Sampling

```

1:  $\mathbf{x}_T \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ 
2: for  $t = T, \dots, 1$  do
3:    $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$  if  $t > 1$ , else  $\mathbf{z} = \mathbf{0}$ 
4:    $\mathbf{x}_{t-1} = \frac{1}{\sqrt{\alpha_t}} \left( \mathbf{x}_t - \frac{1-\alpha_t}{\sqrt{1-\alpha_t}} \epsilon_\theta(\mathbf{x}_t, t) \right) + \sigma_t \mathbf{z}$ 
5: end for
6: return  $\mathbf{x}_0$ 

```

### Implementation details

1. Describe how you implement your model, including your choice of DDPM, UNet architectures, noise schedule, and loss functions.

- Unet

I use 'from diffusers import UNet2DModel' as my backbone. UNet has a series of downsampling and upsampling operations, and each symmetric block is connected with skip connections. The model is referenced from the link :

[https://colab.research.google.com/github/huggingface/notebooks/blob/main/diffusers\\_doc/en/pytorch/basic\\_training.ipynb?fbclid=IwAR00whkYG96g4upqpLAIYI25A-6RK-5tGO8tkVbjMgQZ5V-j69yi2u59lk#scrollTo=BWg\\_7W6ehw28](https://colab.research.google.com/github/huggingface/notebooks/blob/main/diffusers_doc/en/pytorch/basic_training.ipynb?fbclid=IwAR00whkYG96g4upqpLAIYI25A-6RK-5tGO8tkVbjMgQZ5V-j69yi2u59lk#scrollTo=BWg_7W6ehw28)

```
#create model
model = UNet2DModel(
    sample_size = 64,
    in_channels = 3,
    out_channels = 3,
    layers_per_block = 2,
    class_embed_type = None,
    num_class_embeds = 2325, #C (24, 3) + 1
    block_out_channels = (128, 128, 256, 256, 512, 512),
    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", # a regular ResNet upsampling block
        "AttnUpBlock2D", # a ResNet upsampling block with spatial self-attention
        "UpBlock2D",
        "UpBlock2D",
        "UpBlock2D",
        "UpBlock2D",
    ),
)
```

- Dataloader

To give the model what object we want to generate. It's crucial to embed the information to the input.

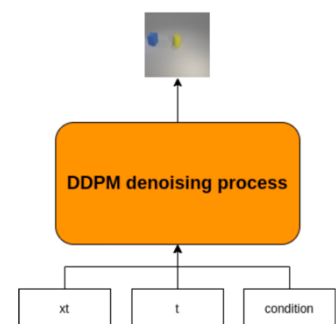
In dataloader, for train scheme's `__getitem__` function, it will return the normalize img and the encoded int label.

For test scheme's `__getitem__` function, it will return the dummy img(not used) and the one-hot encoded label for evaluator use. The encode process will apply in sample function to transform to `time_embed`'s dimension.

- Class embedding

After declaring the model, I overwrite the `class_embed` to `nn.Linear`, which help us transform one hot encode to time embed dimension. When predicting noise, the model will consider sum up the `time_emb` and `class_emb` to provide the Information to predict noise.

```
model = UNet2DModel()
model.class_embedding = nn.Linear(24, 512)
```



- Load and save your model.

Save model is simple:

```
model.save_pretrained("./local-unet"+"epoch_"+str(epoch), variant="non_ema")
```

If you overwrite the model, there will be some issue when you try to load the model from your saved weight file.

Your nn.linear weight need to load separatly, and set

**low\_cpu\_mem\_usage=False,ignore\_mismatched\_sizes=True**

```
#load model
model = UNet2DModel.from_pretrained(pretrained_model_name_or_path="local-unetepoch_60",
    variant="non_ema", from_tf=True, low_cpu_mem_usage=False, ignore_mismatched_sizes=True)
print(model.class_embedding)
model.class_embedding = nn.Linear(24, 512)
state_dict = torch.load("local-unetepoch_60/diffusion_pytorch_model.non_ema.bin")
filtered_state_dict = {k[16:]: v for k, v in state_dict.items() if k != "class_embedding.weight" or k != "class_embedding.bias"}
model.class_embedding.load_state_dict(filtered_state_dict)
print(model.class_embedding)
model = model.to(args.device)
```

Some weights of the model checkpoint at local-unetepoch\_60 were not used when initializing UNet2DModel: ['class\_embedding.weight', 'class\_embedding.bias']  
 - This IS expected if you are initializing UNet2DModel from the checkpoint of a model trained on another task or with another architecture (e.g. initializing a BertForSequenceClassification model from a BertForPreTraining model).  
 - This IS NOT expected if you are initializing UNet2DModel from the checkpoint of a model that you expect to be exactly identical (initializing a BertForSequenceClassification model from a BertForSequenceClassification model).  
 None  
 Linear(in\_features=24, out\_features=512, bias=True)

- Train

- Prepare rand\_t, data and Gaussian noise, so we can compute noise img(xt) in step t.
- Prepare class\_label, which is encoded in dataloader, timestep, and xt img to model, the model will predict a noise.
- Compare the predict noise and ground truth noise to compute MSE loss.

- Sample

- Convert cond to embed.
- From max timestep, start from a Gaussian noise, iteratively input xt, timestep, class embed to denoise the image. Compute the xt-1.
- Use evaluator to evaluate the performance.

- noise schedule

I use the linear noise schedule, it is easy to use linspace to compute all the data in the beginning.

- loss functions

I use Mean Square Error as my loss function. The predict noise and the ground truth are the input to the loss function.

## 2. Specify the hyperparameters (learning rate, epochs, etc.)

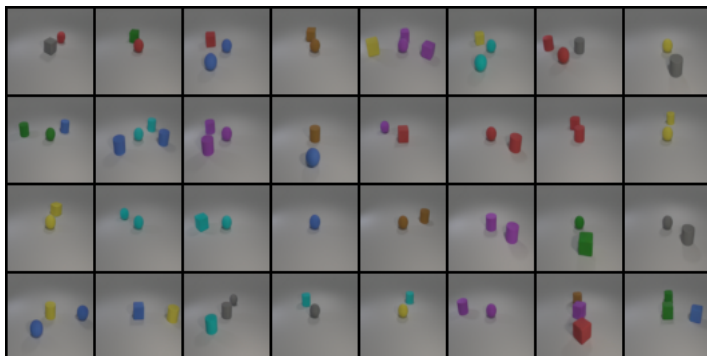
- learning rate :  $1e-4$
- epochs : 500
- optimizer : AdamW
- scheduler : get\_cosine\_schedule\_with\_warmup
- timestep = 1200

## Results and discussion

- Show your results based on the testing data. (including images)  
test.json



new\_test.json



- Discuss the results of different model architectures. For example, what is the effect with or without some specific embedding methods, or what kind of prediction type is more effective in this case.

I have tried another embed method.

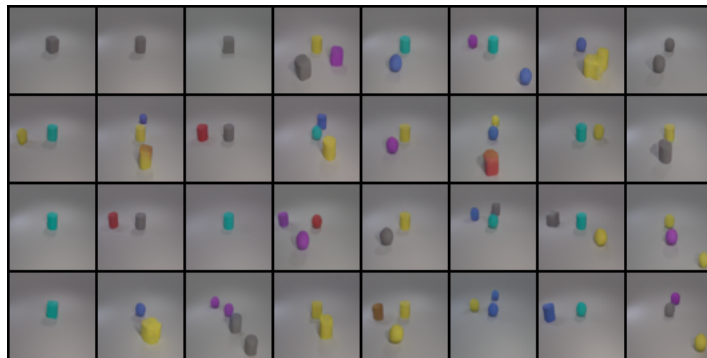
There are some restrictions for the dataset :

- One image can include objects from 1 to 3
- The same object will not appear twice in an image.
- There are 24 objects in iclevr dataset.



So, there are  $C_3^{24} + C_2^{24} + C_1^{24}$  of possibility for dataset. I encode each condition to an integer, which can be pass to model.(passed by class\_labels) Model will transform it to time\_embed dimension, and will be sum with the time\_embed code to provide more information to model.

**It turns out to be a failure.** Although the loss keep decreasing to 1e-3, but the generate image didn't follow the instruction in the json file. It just generate the image whatever it want. I think it is because using an encoded number can not quickly learn the relation between each task, and one-hot encoding can provide more information than an encoded number.

test.json



## Experimental results

test.json	new_test.json
	
<div><div>main.py</div><div>test_37.txt ×</div><div>test_37.png</div></div> <div><div>test_37.txt</div><div>1 Task 0 : 1.0</div><div>2 Task 1 : 1.0</div><div>3 Task 2 : 1.0</div><div>4 Task 3 : 0.5</div><div>5 Task 4 : 1.0</div><div>6 Task 5 : 0.5</div><div>7 Task 6 : 1.0</div><div>8 Task 7 : 1.0</div><div>9 Task 8 : 1.0</div><div>10 Task 9 : 0.6666666666666666</div><div>11 Task 10 : 0.6666666666666666</div><div>12 Task 11 : 1.0</div><div>13 Task 12 : 0.6666666666666666</div><div>14 Task 13 : 1.0</div><div>15 Task 14 : 0.6666666666666666</div><div>16 Task 15 : 0.6666666666666666</div><div>17 Task 16 : 1.0</div><div>18 Task 17 : 1.0</div><div>19 Task 18 : 1.0</div><div>20 Task 19 : 1.0</div><div>21 Task 20 : 0.5</div><div>22 Task 21 : 0.5</div><div>23 Task 22 : 1.0</div><div>24 Task 23 : 1.0</div><div>25 Task 24 : 0.5</div><div>26 Task 25 : 1.0</div><div>27 Task 26 : 0.6666666666666666</div><div>28 Task 27 : 0.6666666666666666</div><div>29 Task 28 : 0.6666666666666666</div><div>30 Task 29 : 1.0</div><div>31 Task 30 : 0.6666666666666666</div><div>32 Task 31 : 1.0</div><div>33 Accuracy : 0.8055555555555556</div><div>34</div></div>	<div><div>main.py</div><div>new_test_30.txt ×</div><div>new_test_30.png</div></div> <div><div>dataset &gt; Result &gt; new_test_30.txt</div><div>1 Task 0 : 0.6666666666666666</div><div>2 Task 1 : 0.3333333333333333</div><div>3 Task 2 : 0.6666666666666666</div><div>4 Task 3 : 0.6666666666666666</div><div>5 Task 4 : 1.0</div><div>6 Task 5 : 0.6666666666666666</div><div>7 Task 6 : 1.0</div><div>8 Task 7 : 1.0</div><div>9 Task 8 : 1.0</div><div>10 Task 9 : 0.6666666666666666</div><div>11 Task 10 : 0.6666666666666666</div><div>12 Task 11 : 1.0</div><div>13 Task 12 : 0.5</div><div>14 Task 13 : 1.0</div><div>15 Task 14 : 1.0</div><div>16 Task 15 : 1.0</div><div>17 Task 16 : 1.0</div><div>18 Task 17 : 0.5</div><div>19 Task 18 : 1.0</div><div>20 Task 19 : 1.0</div><div>21 Task 20 : 1.0</div><div>22 Task 21 : 0.6666666666666666</div><div>23 Task 22 : 1.0</div><div>24 Task 23 : 1.0</div><div>25 Task 24 : 0.6666666666666666</div><div>26 Task 25 : 0.6666666666666666</div><div>27 Task 26 : 1.0</div><div>28 Task 27 : 0.6666666666666666</div><div>29 Task 28 : 1.0</div><div>30 Task 29 : 0.5</div><div>31 Task 30 : 1.0</div><div>32 Task 31 : 1.0</div><div>33 Accuracy : 0.8214285714285714</div><div>34</div></div>