Deep Learning Lab 7

311552004 李明倫

1. Introduction

In this lab, you need to implement a conditional Denoising Diffusion Probabilistic Models (DDPM) to generate synthetic images according to multi-label conditions. Given a specific condition, your model should generate the corresponding synthetic images (Fig. 1).

For example, given “red cube” and “blue cylinder”, your model should generate the synthetic images with red cube and blue cylinder and meanwhile, input your generated images to a pre-trained classifier for evaluation.

To achieve higher generation capacity, especially in computer vision, DDPM is proposed and has been widely applied on style transfer and image synthesis.

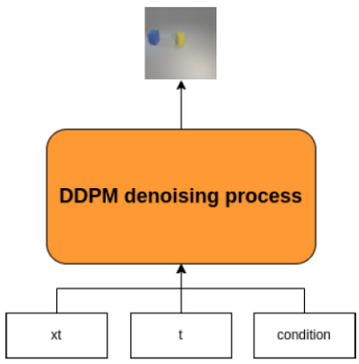


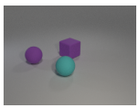
Figure 1: The illustration of conditional DDPM

Requirements

- Implement a conditional DDPM
  - Choose your conditional DDPM setting
  - Design your noise schedule and UNet architecture
  - Choose your loss functions
  - Implement the training function, testing function, and data loader
- Generate the synthetic images
  - Evaluate the accuracy of test.json and new_test.json

Dataset

We use i-CLEVR dataset to train our DDPM .



- objects.json

  A dictionary file that contains the number of objects and the indexes. There are totally 24 objects in i-CLEVR dataset with 3 shapes and 8 colors.

- train.json

  A dictionary file that contains 18009 training data where keys are filenames and values are objects.

  {"CLEVR_train_001032_0.png": ["yellow sphere"],
  "CLEVR_train_001032_1.png": ["yellow sphere", "gray cylinder"],
  "CLEVR_train_001032_2.png": ["yellow sphere", "gray cylinder", "purple cube"], ... }
  One image can include objects from 1 to 3

- test.json

  A list file contains 32 testing data where each element includes multiple objects

  [['gray cube'],
  ['red cube'],
  ['blue cube'],
  ['blue cube', 'green cube'], ...]

2. Implementation details

在 DDPM 的實作上，我採用 diffusers 的框架定義了一個 `MyConditionedUNet` 的 class。

UNet Architecture

```
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",
    ),
)
```

DDPM 的 denoising process 會需要做 noise prediction，這邊也採用大部分實作 DDPM 中會使用的 UNet 架構。原本 diffusers 的 UNet2DModel 只有提供單一 class label 的 condition，但這次作業需要的是 multi-label 的 condition，因此我把 UNet2DModel 的程式碼抓下改進行修改。我新增了一層 embedding layer 負責把輸入的 condition vector (onehot vector) 進行維度的轉換，會轉成跟 time embedding 同樣的大小。

```
self.class_embedding = nn.Linear(24, time_embed_dim)
```

接著在 `forward()` 裡面把輸入的 class_labels 透過剛剛定義的 `self.class_embedding` 轉換完之後，跟原本的 timestep embedding 做相加。由於 timestep embedding 會傳入 UNet 中的每一個 block，因此可以把 condition 的資訊也一起傳進去 model 中做學習。

```
def forward(self, x, t, class_labels):
...
  class_emb = self.class_embedding(class_labels).to(dtype=self.dtype)
  emb = emb + class_emb
```

- *model.py*

```
from dataclasses import dataclass
from typing import Optional, Tuple, Union
import torch
import torch.nn as nn
from diffusers import UNet2DModel

from diffusers.configuration_utils import ConfigMixin, register_to_config
from diffusers.utils import BaseOutput
from diffusers.models.embeddings import GaussianFourierProjection, TimestepEmbedding, Timesteps
from diffusers.models.modeling_utils import ModelMixin
from diffusers.models.unet_2d_blocks import UNetMidBlock2D, get_down_block, get_up_block


@dataclass
class UNet2DOutput(BaseOutput):
    """
    Args:
        sample (`torch.FloatTensor` of shape `(batch_size, num_channels, height, width)`):
            Hidden states output. Output of last layer of model.
    """

    sample: torch.FloatTensor

class MyConditionedUNet(ModelMixin, ConfigMixin):
    @register_to_config
    def __init__(
        self,
        sample_size: Optional[Union[int, Tuple[int, int]]] = None,
        in_channels: int = 3,
        out_channels: int = 3,
        center_input_sample: bool = False,
        time_embedding_type: str = "positional",
        freq_shift: int = 0,
        flip_sin_to_cos: bool = True,
        down_block_types: Tuple[str] = ("DownBlock2D", "AttnDownBlock2D", "AttnDownBlock2D", "AttnDownBlock2D"),
        up_block_types: Tuple[str] = ("AttnUpBlock2D", "AttnUpBlock2D", "AttnUpBlock2D", "UpBlock2D"),
        block_out_channels: Tuple[int] = (224, 448, 672, 896),
        layers_per_block: int = 2,
        mid_block_scale_factor: float = 1,
        downsample_padding: int = 1,
        act_fn: str = "silu",
        attention_head_dim: Optional[int] = 8,
        norm_num_groups: int = 32,
        norm_eps: float = 1e-5,
        resnet_time_scale_shift: str = "default",
        add_attention: bool = True,
    ):
        super(MyConditionedUNet, self).__init__()

        self.sample_size = sample_size
        time_embed_dim = block_out_channels[0] * 4

        # Check inputs
        if len(down_block_types) != len(up_block_types):
            raise ValueError(
```

```
                f"Must provide the same number of `down_block_types` as `up_block_types`. `down_block_types`: {down_block_types}. `up_block
            )

        if len(block_out_channels) != len(down_block_types):
            raise ValueError(
                f"Must provide the same number of `block_out_channels` as `down_block_types`. `block_out_channels`: {block_out_channels}. `
            )

        # input
        self.conv_in = nn.Conv2d(in_channels, block_out_channels[0], kernel_size=3, padding=(1, 1))

        # time
        if time_embedding_type == "fourier":
            self.time_proj = GaussianFourierProjection(embedding_size=block_out_channels[0], scale=16)
            timestep_input_dim = 2 * block_out_channels[0]
        elif time_embedding_type == "positional":
            self.time_proj = Timesteps(block_out_channels[0], flip_sin_to_cos, freq_shift)
            timestep_input_dim = block_out_channels[0]

        self.time_embedding = TimestepEmbedding(timestep_input_dim, time_embed_dim)

        # class embedding
        self.class_embedding = nn.Linear(24, time_embed_dim)        # time_embed_dim = 512

        # if class_embed_type is None and num_class_embeds is not None:
        #     self.class_embedding = nn.Embedding(num_class_embeds, time_embed_dim)
        # elif class_embed_type == "timestep":
        #     self.class_embedding = TimestepEmbedding(timestep_input_dim, time_embed_dim)
        # elif class_embed_type == "identity":
        #     self.class_embedding = nn.Identity(time_embed_dim, time_embed_dim)
        # else:
        #     self.class_embedding = None

        self.down_blocks = nn.ModuleList([])
        self.mid_block = None
        self.up_blocks = nn.ModuleList([])

        # down
        output_channel = block_out_channels[0]
        for i, down_block_type in enumerate(down_block_types):
            input_channel = output_channel
            output_channel = block_out_channels[i]
            is_final_block = i == len(block_out_channels) - 1

            down_block = get_down_block(
                down_block_type,
                num_layers=layers_per_block,
                in_channels=input_channel,
                out_channels=output_channel,
                temb_channels=time_embed_dim,
                add_downsample=not is_final_block,
                resnet_eps=norm_eps,
                resnet_act_fn=act_fn,
                resnet_groups=norm_num_groups,
                attn_num_head_channels=attention_head_dim,
                downsample_padding=downsample_padding,
                resnet_time_scale_shift=resnet_time_scale_shift,
            )
            self.down_blocks.append(down_block)

        # mid
        self.mid_block = UNetMidBlock2D(
            in_channels=block_out_channels[-1],
            temb_channels=time_embed_dim,
            resnet_eps=norm_eps,
            resnet_act_fn=act_fn,
            output_scale_factor=mid_block_scale_factor,
            resnet_time_scale_shift=resnet_time_scale_shift,
            attn_num_head_channels=attention_head_dim,
            resnet_groups=norm_num_groups,
            add_attention=add_attention,
        )

        # up
        reversed_block_out_channels = list(reversed(block_out_channels))
        output_channel = reversed_block_out_channels[0]
        for i, up_block_type in enumerate(up_block_types):
            prev_output_channel = output_channel
            output_channel = reversed_block_out_channels[i]
            input_channel = reversed_block_out_channels[min(i + 1, len(block_out_channels) - 1)]

            is_final_block = i == len(block_out_channels) - 1

            up_block = get_up_block(
                up_block_type,
                num_layers=layers_per_block + 1,
                in_channels=input_channel,
                out_channels=output_channel,
                prev_output_channel=prev_output_channel,
                temb_channels=time_embed_dim,
                add_upsample=not is_final_block,
                resnet_eps=norm_eps,
                resnet_act_fn=act_fn,
                resnet_groups=norm_num_groups,
                attn_num_head_channels=attention_head_dim,
                resnet_time_scale_shift=resnet_time_scale_shift,
            )
            self.up_blocks.append(up_block)
            prev_output_channel = output_channel

        # out
        num_groups_out = norm_num_groups if norm_num_groups is not None else min(block_out_channels[0] // 4, 32)
        self.conv_norm_out = nn.GroupNorm(num_channels=block_out_channels[0], num_groups=num_groups_out, eps=norm_eps)
        self.conv_act = nn.SiLU()
        self.conv_out = nn.Conv2d(block_out_channels[0], out_channels, kernel_size=3, padding=1)

    def forward(
        self,
```

```
        sample: torch.FloatTensor,
        timestep: Union[torch.Tensor, float, int],
        class_labels: Optional[torch.Tensor] = None,
        return_dict: bool = True,
    ) -> Union[UNet2DOutput, Tuple]:
        r"""
        Args:
            sample (`torch.FloatTensor`): (batch, channel, height, width) noisy inputs tensor
            timestep (`torch.FloatTensor` or `float` or `int): (batch) timesteps
            class_labels (`torch.FloatTensor`, *optional*, defaults to `None`):
                Optional class labels for conditioning. Their embeddings will be summed with the timestep embeddings.
            return_dict (`bool`, *optional*, defaults to `True`):
                Whether or not to return a [`~models.unet_2d.UNet2DOutput`] instead of a plain tuple.

        Returns:
            [`~models.unet_2d.UNet2DOutput`] or `tuple`: [`~models.unet_2d.UNet2DOutput`] if `return_dict` is True,
            otherwise a `tuple`. When returning a tuple, the first element is the sample tensor.
        """
        # 0. center input if necessary
        if self.config.center_input_sample:
            sample = 2 * sample - 1.0

        # 1. time
        timesteps = timestep
        if not torch.is_tensor(timesteps):
            timesteps = torch.tensor([timesteps], dtype=torch.long, device=sample.device)
        elif torch.is_tensor(timesteps) and len(timesteps.shape) == 0:
            timesteps = timesteps[None].to(sample.device)

        # broadcast to batch dimension in a way that's compatible with ONNX/Core ML
        timesteps = timesteps * torch.ones(sample.shape[0], dtype=timesteps.dtype, device=timesteps.device)

        t_emb = self.time_proj(timesteps)

        # print('timesteps: {}, shape: {}'.format(timesteps, timesteps.shape))
        # print('t_emb shape: {}'.format(t_emb.shape))

        # timesteps does not contain any weights and will always return f32 tensors
        # but time_embedding might actually be running in fp16. so we need to cast here.
        # there might be better ways to encapsulate this.
        t_emb = t_emb.to(dtype=self.dtype)
        emb = self.time_embedding(t_emb)

        if self.class_embedding is not None:
            if class_labels is None:
                raise ValueError("class_labels should be provided when doing class conditioning")

            # if self.config.class_embed_type == "timestep":
            #     class_labels = self.time_proj(class_labels)

            class_emb = self.class_embedding(class_labels).to(dtype=self.dtype)
            emb = emb + class_emb

        # 2. pre-process
        skip_sample = sample
        sample = self.conv_in(sample)

        # 3. down
        down_block_res_samples = (sample,)
        for downsample_block in self.down_blocks:
            if hasattr(downsample_block, "skip_conv"):
                sample, res_samples, skip_sample = downsample_block(
                    hidden_states=sample, temb=emb, skip_sample=skip_sample
                )
            else:
                sample, res_samples = downsample_block(hidden_states=sample, temb=emb)

            down_block_res_samples += res_samples

        # 4. mid
        sample = self.mid_block(sample, emb)

        # 5. up
        skip_sample = None
        for upsample_block in self.up_blocks:
            res_samples = down_block_res_samples[-len(upsample_block.resnets) :]
            down_block_res_samples = down_block_res_samples[: -len(upsample_block.resnets)]

            if hasattr(upsample_block, "skip_conv"):
                sample, skip_sample = upsample_block(sample, res_samples, emb, skip_sample)
            else:
                sample = upsample_block(sample, res_samples, emb)

        # 6. post-process
        sample = self.conv_norm_out(sample)
        sample = self.conv_act(sample)
        sample = self.conv_out(sample)

        if skip_sample is not None:
            sample += skip_sample

        if self.config.time_embedding_type == "fourier":
            timesteps = timesteps.reshape((sample.shape[0], *([1] * len(sample.shape[1:]))))
            sample = sample / timesteps

        if not return_dict:
            return (sample,)

        return UNet2DOutput(sample=sample)
```

Noise Schedule

Scheduler 的實作上，也是採用 diffusers 提供的 `DDPMScheduler`，預設為 linear 的 beta schedule，並將 `num_train_timesteps` 設為 1000。

```
from diffusers import DDPMScheduler
...
noise_scheduler = DDPMScheduler(num_train_timesteps=1000)
```

Loss Function

這裡採用 `nn.MSELoss()` 去評估原始 noise 與 UNet 預測的 noise 的差異進行訓練。

```
loss_fn = nn.MSELoss()
...
loss = loss_fn(noise_pred, noise)
```

DDPM

這個部分為整個 DDPM 訓練的核心，每一次訓練的時候都是先 sample 一些 random noise 出來 `noise = torch.randn_like(x)`、sample 一些 random 的 timsteps `timesteps = torch.randint(0, 1000, (x.shape[0],))`，並一起傳進去給 `noise_scheduler.add_noise(x, noise, timesteps)`，他會根據 timesteps 的不同加上不同強度的 noise，並回傳加完 noise 的 image (*noisy_image*)。

接著就把這個 noisy_image、timesteps 還有 condition 傳進我們前面寫好的 ConditionalUNet，他就會幫我們去計算當前 image 的雜訊為何，並回傳他預測的雜訊 (*noise_pred*)。

最後透過 `MSELoss()` 去計算原本 sample 到的 noise 以及 model 預測的 noise_pred 之間的差異，我們會希望 model 預測的 noise 能夠越接近真實的 noise 越好。

```
for epoch in range(num_epochs):
    progress_bar = tqdm(total=len(train_loader), disable=not accelerator.is_local_main_process)
    progress_bar.set_description(f"Epoch {epoch+1}/{num_epochs}")

    total_loss = 0
    for i, (x, class_label) in enumerate(train_loader):
        x, class_label = x.to(device), class_label.to(device)

        # 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)

        with accelerator.accumulate(model):
            # get the model prediction
            noise_pred = model(noisy_image, timesteps, class_label).sample

            # 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()
```

Inference / Generation process

在 inference 的時候，一樣我們先從 normal distribution sample 一筆 random noise 開始，接著跑過 scheduler 的 num_inference_steps (1000, 999, ..., 0)，把當前的 sample 以及 conditional label 丟進去 UNet 中去計算雜訊。下一步透過 `scheduler.step()` 去移除計算出來的雜訊，得到一個更乾淨的 sample，一直持續這個過程直到 timestep 等於 0，最後就會得到我們想要的 image。

```
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))
```

Other implementation details

- ICLEVR_Dataset - 讀取 image 以及對應的 class label，並轉成 onehot vector 的形式回傳

```
class ICLEVR_Dataset(Dataset):
    def __init__(self, args, mode='train', transforms=None):
        self.root = args.dataset_dir
        self.mode = mode
        self.transforms = transforms
        self.images, self.labels = self.get_data()

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

    def __getitem__(self, idx):
```

```
        image = Image.open(self.images[idx]).convert("RGB")
        label = self.labels[idx]

        if self.transforms:
            image = self.transforms(image)

        return image, label

    def get_data(self):
        # 先讀 label 的對應檔, "gray cube" -> 0
        label_dict = json.load(open(os.path.join(self.root, "objects.json")))
        data_dict = json.load(open(os.path.join(self.root, self.mode + ".json")))

        images = list(data_dict.keys())
        labels = list(data_dict.values())

        newimages, newLabels = [], []
        for i in range(len(labels)):
            newimages.append(os.path.join(self.root, "iclevr/" + images[i]))

            onehot_label = np.zeros(24, dtype=np.float32)
            for j in range(len(labels[i])):
                onehot_label[label_dict[labels[i][j]]] = 1
            newLabels.append(onehot_label)

        return newimages, newLabels
```

- Accelerator - 透過 accelerator 幫助提升模型訓練的效率

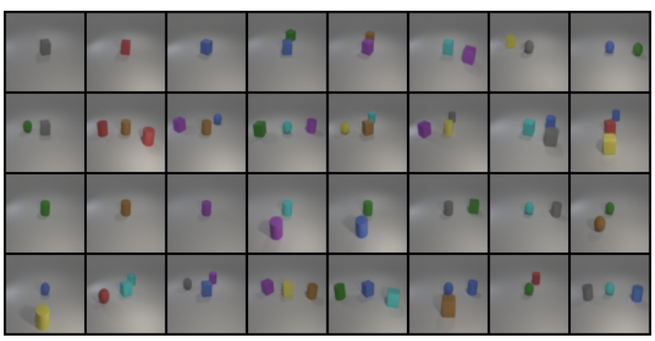
```
# Accelerator
accelerator = Accelerator(
    mixed_precision=args.mixed_precision,
    gradient_accumulation_steps=args.gradient_accumulation_steps,
    log_with="tensorboard",
    project_dir=os.path.join(args.log_dir, "logging"),
)

if accelerator.is_main_process:
    accelerator.init_trackers("train_example")

model, optimizer, train_loader, lr_scheduler = accelerator.prepare(
    model, optimizer, train_loader, lr_scheduler
)
```

Hyperparameters (learning rates, epochs, etc.)

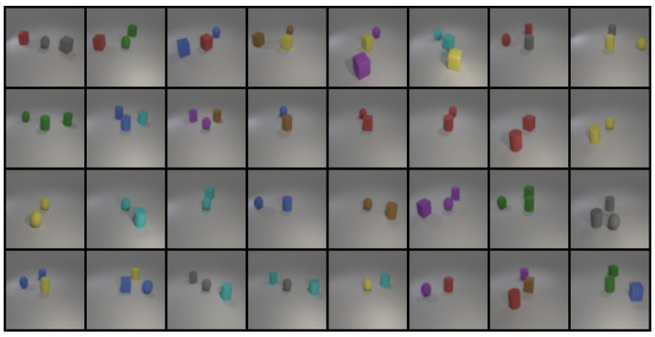
- batch_size: 64
- lr: 0.0001, with cosine schedule & warmup
- num_epochs: 150
- beta_schedule: Linear
- prediction_type: Epsilon
- UNet_Block_dim: [128, 128, 256, 256, 512, 512], [512, 512, 256, 256, 128, 128]
- Layers_per_block: 2

3. Results and discussion

- **Test -** 0.91667



- **New_Test -** 0.92857



- **Training Curve**

Discuss the results of different model architectures

Different condition embedding method

→ “Feeding it in as additional channels in the input to the UNet”

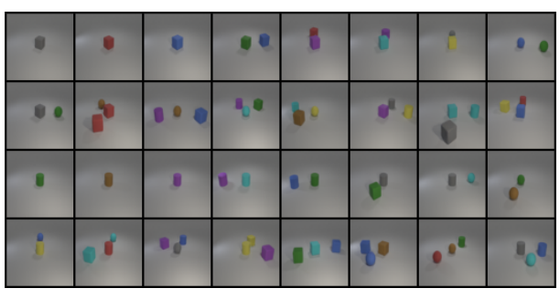
Class label is mapped to an embedding and then expanded to be the same width and height as the input image so that it can be fed in as additional channels.

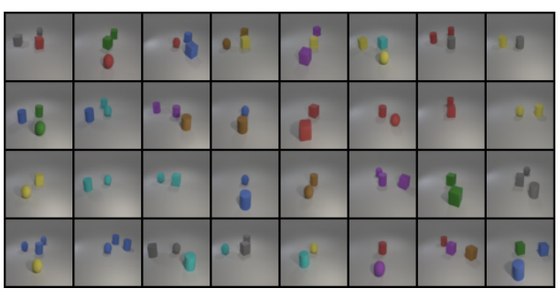
原本的輸入圖片有 RGB 三個 channel，這邊把 class label (dim=24 onehot vector) 變成 24 個 channel，疊到原本的輸入圖片後面。

```
if self.config.class_embed_type == "channel":
  bs, c, w, h = sample.shape
  class_emb = class_labels.view(bs, class_labels.shape[1], 1, 1).expand(bs, class_labels.shape[1], w, h)
  sample = torch.cat((sample, class_emb), 1)      # (bs, 3+24, 64, 64)
```

- Result - ***Test.json: 0.81944, New_Test.json: 0.79762***

從實驗結果來看，同樣訓練了 150 個 epoch 之後，model 依然可以產生品質不錯的圖片，但收斂的時間明顯比原本的做法來的慢了一些，結果沒有比原本的方法好，但也許再拉長一點訓練時間可以獲得更好的結果。



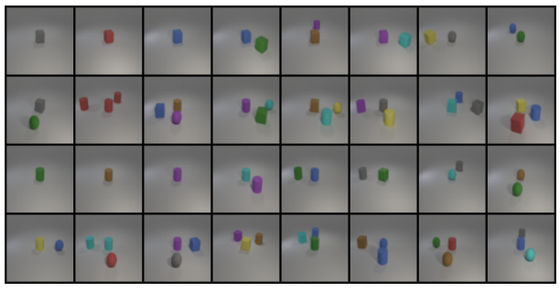


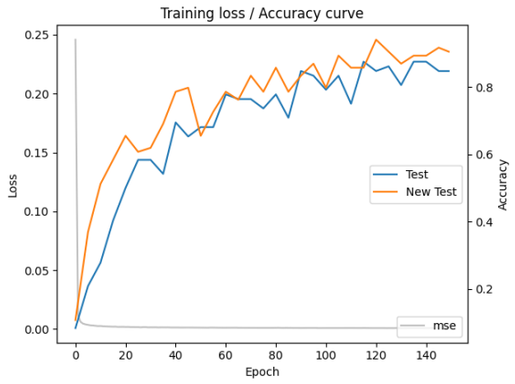
Different model architecture

→ “1 layer per block in UNet”

原本的 UNet 使用 2 layers per block，這邊簡化模型看看訓練的效果如何。結果可以看到即使只使用一層的 block，model 依然能夠得到不錯的結果，而且兩個 testing score 也都到還不錯的分數。

- Result - ***Test.json: 0.87500, New_Test.json: 0.89286***

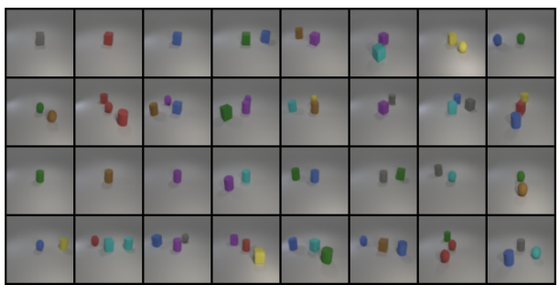


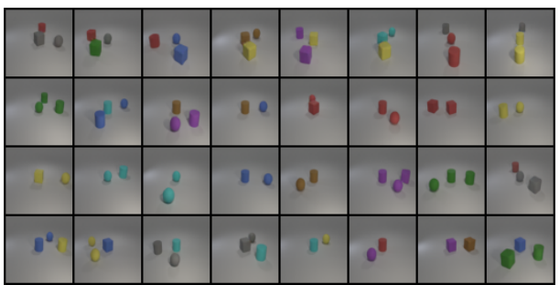


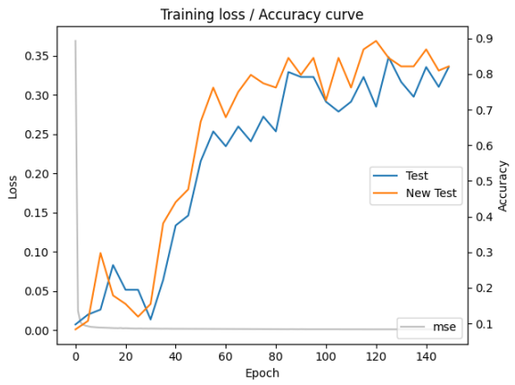
→ “Half channels in UNet”

原本的 UNet_Block_dim: [128, 128, 256, 256, 512, 512], [512, 512, 256, 256, 128, 128]，這邊把 channel 都降低為一半 [64, 64, 128, 128, 256, 256], [256, 256, 128, 128, 64, 64]，簡化模型看看訓練的效果如何。結果可以看到使用一半的 channel 數目，在前期的訓練會比較掙扎，但隨著訓練時間的拉長，最後 model 依然能夠得到不錯的結果，而且兩個 testing score 也都有到還不錯的分數。

- Result - ***Test.json: 0.84722, New_Test.json: 0.84524***







Different beta schedule

→ Cosine noise schedule

原本使用 Linear 的 noise schedule，這個部分換成 cosine 的版本試試看結果如何。結果可以看到比原本 Linear 的效果還要好，兩個 testing score 都到蠻高的分數。

- Result - ***Test.json: 0.93056, New_Test.json: 0.97619***