312553040_郭晉維_Lab5_Report

1. Introduction

電腦視覺中的 Image Inpainting 任務旨在自動填補圖像中的缺失或損壞區域，使其看起來自然且無痕跡。這項技術的核心在於能夠從圖像的剩餘部分推斷並生成適當的內容來填補空白。其應用廣泛，包括修復受損照片、移除不需要的物體或水印、圖像編輯、以及在電影和電視節目中的特效製作。Image Inpainting 的實現依賴於深度學習技術，特別是卷積神經網絡(CNNs)和生成對抗網絡(GANs)。這些模型通過大量的圖像數據進行訓練，學習圖像中物體和背景的紋理、結構和顏色分佈。CNNs 能夠提取圖像的特徵，而 GANs 則通過對抗學習生成高質量的填補內容，其中生成器生成圖像片段，判別器則判斷片段的真實性。在具體應用中，Image Inpainting 不僅要求生成的內容在視覺上與原始圖像保持一致，還需要考慮圖像的整體連續性和上下文關係。例如，修復一張損壞的臉部照片，不僅需要重建面部的紋理，還要確保五官的位置和比例正確。此外，技術的進步還使得這些模型能夠更快、更精確地完成填補任務，並在各種不同的圖像和場景中取得令人滿意的效果。

Vision Transformer (ViT) 是一種將 Transformer 架構應用於計算機視覺任務的模型。由 Dosovitskiy 等人在 2020 年提出，ViT 打破了傳統卷積神經網絡(CNN)在圖像識別中的主導地位。ViT 將圖像分割成固定大小的區塊(patches)，並將每個區塊展平成一維序列，然後將這些序列作為 Transformer 模型的輸入。通過 self-attention 機制，ViT 能夠捕捉圖像中不同區域之間的全局關係，從而實現高效且精確的圖像分類和識別。ViT 在多個視覺任務上取得了與最先進的 CNN 可比甚至更好的性能，展示了 Transformer 架構在視覺領域的巨大潛力。

MaskGIT 是一種用於圖像生成的自回歸模型，由 Google Research 團隊在 2021 年提出。這種模型結合了 Transformer 架構和生成對抗網絡(GAN)的優勢，專注於生成高質量和高分辨率的圖像。MaskGIT 的核心理念是將圖像生成問題轉化為一個逐步填充的過程，通過預測和填充隱藏的圖像塊(patches)，逐步構建完整圖像。這種方法能有效捕捉圖像的全局結構和細節，並且在生成過程中引入了 self-attention 機制，使得模型能夠理解和利用圖像內部的長距離依賴關係。MaskGIT 在多個圖像生成任務中展現了卓越的性能，包括圖像修復、圖像生成以及超分辨率圖像重建。

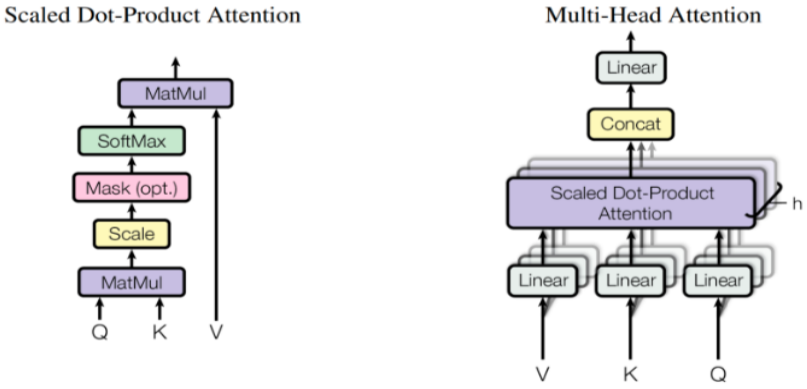
這次的作業主要實作了 MaskGIT 中的 Multi-Head Self-Attention、Stage 2 中的 Masked Visual Token modeling (MVTM) Training 以及 Different Mask Scheduling Inference，讓我對於 Transformer 以及 MaskGIT in Image Inpainting 有更深入的瞭解並學到許多實作時需要注意的細節，因此這次作業讓我獲益良多。

2. Implementation Details

A. Model (Multi-Head Self-Attention)

在 layer.py 檔案中實作 Multi-Head Self-Attention 的部分我參考了

https://juejin.cn/s/pytorch%20multiheadattention 使用 網站以及下圖一的流程圖，首先分別建立對 Q , K , V 進行線性轉換的 model 以及最後產生 output 的 fully connected layer。forward(self, x)的過程則是先將 x(batch_size, num_image_token, dim)分別轉換到 Q , K , V 空間中並把 Q , K , V 的 shape 都轉成(batch_size, num_image_token, num_of_heads = 16, d_head = 768/16)，接著先利用 Q , K , V 計算 Scaled Dot-Product Attention = $sotfmax(QK^T/\sqrt{d_k})\,V$，然後將 16 個 head 的結果 concat 起來變成 output(batch_size, num_image_token, dim)，最後再經過 fully connected layer 即可得到最後的 output。我實作 Multi-Head Self-Attention 的程式碼如下圖二所示。




```python
class MultiHeadAttention(nn.Module):
    def __init__(self, dim=768, num_heads=16, p=0.1):
        super(MultiHeadAttention, self).__init__()
        self.num_heads = num_heads
        self.d_head = dim // num_heads
        self.q_linear = nn.Linear(dim, dim)
        self.k_linear = nn.Linear(dim, dim)
        self.v_linear = nn.Linear(dim, dim)
        self.drop = nn.Dropout(p)
        self.out = nn.Linear(dim, dim)

    def forward(self, x):
        ''' Hint: input x tensor shape is (batch_size, num_image_tokens, dim),
            because the bidirectional transformer first will embed each token to dim dimension,
            and then pass to n_layers of encoders consist of Multi-Head Attention and MLP.
            # of head set 16, Total d_k , d_v set to 768
            d_k , d_v for one head will be 768//16.
        '''
        batch_size = x.size(0)
        # fully connected layer
        q = self.q_linear(x).view(batch_size, -1, self.num_heads, self.d_head)
        k = self.k_linear(x).view(batch_size, -1, self.num_heads, self.d_head)
        v = self.v_linear(x).view(batch_size, -1, self.num_heads, self.d_head)
        # transpose
        q = q.transpose(1,2)
        k = k.transpose(1,2)
        v = v.transpose(1,2)
        # scaled dot-product attention
        scores = torch.matmul(q, k.transpose(2, 3)) /  math.sqrt(self.d_head)
        scores = nn.functional.softmax(scores, dim=3)
        scores = self.drop(scores)
        output = torch.matmul(scores, v)
        # concat and fully connected layer
        output = output.transpose(1,2).contiguous().view(batch_size, -1, self.num_heads*self.d_head)
        output = self.out(output)
        return output
```

B. Stage2 training (MVTM, forward, loss)

在 VQGAN_Transformer.py 檔案中的 encode_to_z(self, x)函式部分，首先將 input 圖片 x 經過 self.vqgan.encode(x)後分別產生 codebook_mapping 的結果 zq (batch_size, latent_dim, h, w)跟 codebook_indices 的結果 indices(batch_size* latent_dim)，之後再把 indices 的 shape 轉成(batch_size, h *w)。我實作 encode_to_z( )函式的程式碼如下頁圖所示。

```python
##TODO2 step1-1: input x fed to vqgan encoder to get the latent and zq
    @torch.no_grad()
    def encode_to_z(self, x):
        zq, indices, _ = self.vqgan.encode(x)
        indices = indices.view(zq.shape[0], -1)
        return zq, indices
```

gamma_func( )函式部分，根據不同 mask scheduling 分為 cosine、linear、square 三種，在 iterative_decoding 的過程中根據當下的 t 得到 $r = t/T$，把 r 傳入 gamma_func 計算 γ，三種 mask scheduling 公式分別為 Cosine：$\gamma = \cos\frac{\pi r}{2}$、Linear：$\gamma = 1 - r$、Square：$\gamma = 1 - r^2$。

我實作 gamma_func( )函式的程式碼如下圖所示。

```python
##TODO2 step1-2:
    def gamma_func(self, mode="cosine"):

        if mode == "cosine" :
            return lambda r : np.cos(r * np.pi / 2)
        elif mode == "linear" :
            return lambda r : 1 - r
        elif mode == "square" :
            return lambda r : 1 - r*r
```

forward( )函式部分我參考了 https://github.com/dome272/MaskGIT-pytorch/blob/main/transformer.py 網站，首先將 input 圖片 x 經過 self.encode_to_z(x)後產生 h*w 個 token 在 codebook 中最相近 vector，也就是當作 ground truth 的 z_indices(batch_size, h*w)。決定隨機 mask 數量 r，其中 0 < r < h*w，然後再從 h*w 個 token 中隨機選取 r 個 masked token 並記錄在 sample(batch_size, r)，然後建立 mask(batch_size, h*w)，一開始先將 mask 都設為 false，然後再把 sample 中的 index 設為 true。建立 masked_indices(batch_size, h*w)全為 1024 來代表 masked token 的 index，最後把所有 pixel 的 index 存進 a_indices(batch_size, h*w)，沒被 mask 的 token 維持原值(0~1024)，被 mask 的 token 設為 1024。透過 self.transformer(a_indices)產生 logits(batch_size, h*w, 1025)。我實作 forward( )函式的程式碼如下圖所示。

```python
##TODO2 step1-3:
# https://github.com/dome272/MaskGIT-pytorch/blob/main/transformer.py
    def forward(self, x):

        # z_indices = ground truth
        # logits = transformer predict the probability of tokens
        zq, z_indices = self.encode_to_z(x)

        r = math.floor(self.gamma(np.random.uniform()) * z_indices.shape[1])
        sample = torch.rand(z_indices.shape, device=z_indices.device).topk(r, dim=1).indices

        mask = torch.zeros(z_indices.shape, dtype=torch.bool, device=z_indices.device)
        mask.scatter_(dim=1, index=sample, value=True)

        masked_indices = self.mask_token_id * torch.ones_like(z_indices, device=z_indices.device)
        a_indices = mask * z_indices + (~mask) * masked_indices

        logits = self.transformer(a_indices)

        return logits, z_indices
```

train_one_epoch ( )跟 eval_one_epoch ( )函式部分我先將 input 圖片 img 經過 self.model(img)產生 logits (batch_size, h*w, 1025)跟 gt (batch_size, h*w)，然後把 logits 的 shape 變成(batch_size*h*w, 1025)還有把 gt 長度為 batch_size*h*w 的一維陣列，由於是一種離散的分類問題，所以使用 Cross Entropy 來計算 Loss。train_one_epoch ( )計算完 Loss 後再進行 backward 跟 optimizer，我用 Adam optimizer 跟 ExponentialLR Scheduler。兩個函式最後都回傳 epoch_loss。我實作 training_transformer.py 的程式碼如下面四張圖所示。

```python
def train_one_epoch(self, train_loader):
    self.model.train()
    epoch_loss = 0
    pbar = tqdm(train_loader, total=len(train_loader))
    pbar.set_description(f"Epoch {epoch}")
    for i, img in enumerate(pbar):
        self.optim.zero_grad()
        img = img.to(args.device)
        logits, gt = self.model(img)
        logits = logits.reshape(-1, logits.shape[-1])
        gt = gt.reshape(-1)
        loss = F.cross_entropy(logits, gt)
        epoch_loss += loss
        loss.backward()
        self.optim.step(self.scheduler)
        pbar.set_postfix_str(f'Loss: {loss:.3f}')
    epoch_loss /= len(train_loader)
    return epoch_loss
```

```python
def eval_one_epoch(self, val_loader):
    self.model.eval()
    with torch.no_grad():
        epoch_loss = 0
        pbar = tqdm(val_loader, total=len(val_loader))
        pbar.set_description(f"Validation ")
        for i, img in enumerate(pbar):
            img = img.to(args.device)
            logits, gt = self.model(img)
            logits = logits.reshape(-1, logits.size(-1))
            gt = gt.reshape(-1)
            loss = F.cross_entropy(logits, gt)
            epoch_loss += loss
            pbar.set_postfix_str(f'Loss: {loss:.3f}')

    epoch_loss /= len(val_loader)
    return epoch_loss
```

```python
def configure_optimizers(self):
    optimizer = torch.optim.Adam(self.model.transformer.parameters(), lr=args.learning_rate)
    scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer=optimizer, gamma = 0.8)
    return optimizer,scheduler
```

```python
#TODO2 step1-5:
    print("start training")
    train_loss = []
    valid_loss = []
    for epoch in range(args.start_from_epoch, args.epochs+1):
        epoch_train_loss = train_transformer.train_one_epoch(train_loader)
        epoch_valid_loss = train_transformer.eval_one_epoch(val_loader)

        train_loss.append(epoch_train_loss.item())
        valid_loss.append(epoch_valid_loss.item())

        print(f"Epoch {epoch} , Train loss : {epoch_train_loss.item()}, Valid loss : {epoch_valid_loss.item()}")
        torch.save(train_transformer.model.transformer.state_dict(), os.path.join(args.checkpoint_path, f"epoch_{epoch}.pt"))
```

C. Inference for inpainting task (iterative decoding)

在 VQGAN_Transformer.py 檔案中的 inpaiting(self, z_indices, mask_bc, mask_num, ratio)函式部分，首先把 z_indices 中的 masked token 的 index 設為 1024，然後經過 self.transformer(z_indices)產生機率分佈 logits (batch_size, h*w, 1025)，然後再用 softmax 將 logits 變成對於 1025 種不同 label 的機率分

佈，然後 z_indices_predict_prob(batch_size, h*w)跟 z_indices_predict (batch_size, h*w)皆為 logits 中機率最大的 label，接著將 z_indices_predict_prob 中的 not masked token 機率設為 inf，加入 temperature annealing gumbel noise 增加 model 的隨機性，然後算出每個 masked token 對其預測的 confidence 並對其進行排序，挑選最低的 ratio*mask_num 個 token 在下一次 iteration 中繼續被 masked，其餘 masked token 的 index 設為其預測結果，並將 z_indices_predict 中的 not masked token 的 index 設為 ground truth，回傳 z_indices_predict 跟 mask_bc，完成此次 iteration。我實作 inpaiting( )函式的程式碼如下圖所示。

```python
##TODO3 step1-1: define one iteration decoding
    @torch.no_grad()
    def inpainting(self, z_indices, mask_bc, mask_num, ratio):

        z_indices[mask_bc] = 1024
        logits = self.transformer(z_indices)
        #Apply softmax to convert logits into a probability distribution across the last d
        logits = nn.functional.softmax(logits, -1)
        #FIND MAX probability for each token value
        z_indices_predict_prob, z_indices_predict = torch.max(logits, dim=-1)
        z_indices_predict_prob[~mask_bc] = float('inf')
        #predicted probabilities add temperature annealing gumbel noise as confidence
        g = -torch.log(-torch.log(torch.rand_like(z_indices_predict_prob))) # gumbel noise
        temperature = self.choice_temperature * (1 - ratio)
        confidence = z_indices_predict_prob + temperature * g
        #hint: If mask is False, the probability should be set to infinity, so that the to
        #sort the confidence for the rank
        #define how much the iteration remain predicted tokens by mask scheduling
        #At the end of the decoding process, add back the original token values that were
        _, sorted_indices = torch.sort(confidence)
        z_indices_predict[~mask_bc] = z_indices[~mask_bc]
        mask_bc[:, sorted_indices[:, math.floor(ratio*mask_num):]] = False
        return z_indices_predict, mask_bc
```

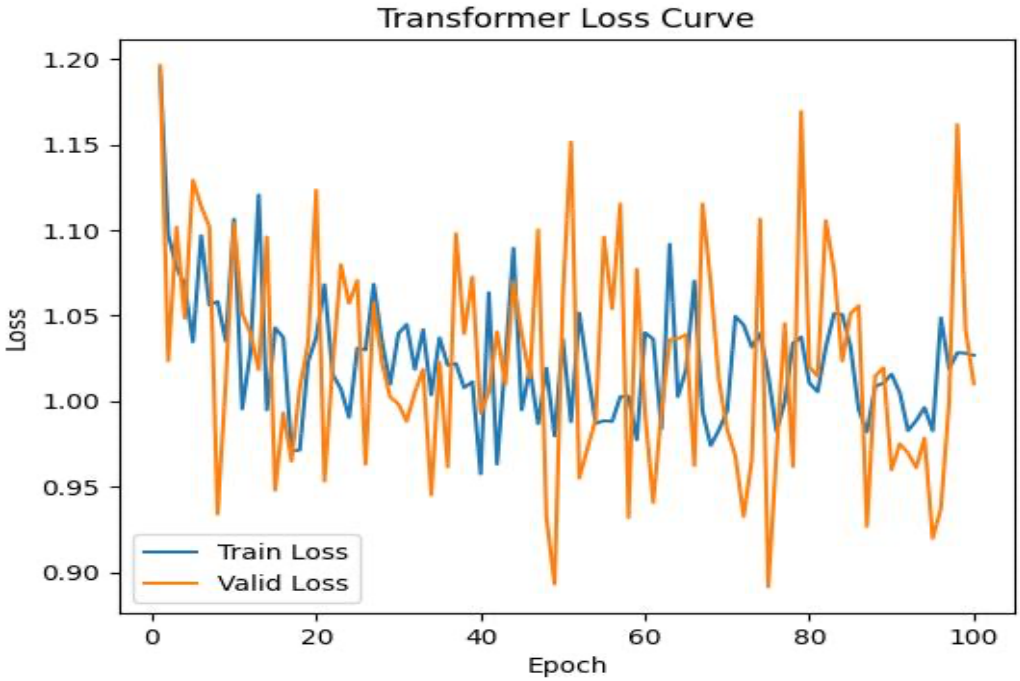
在 inpainting.py 中我將 input 圖片 img 經過 self.model.encode_to_z(img)產生 z_indices，還有在每次 iterration 中，將該次 step 透過 self.model.gamma(step / self.total_iter)產生這次 iteration 的 mask ratio，最後利用 self.model.inpainting(z_indices_predict, mask_bc, mask_num, ratio)產生 z_indices_predict 跟 mask_b。我實作 inpainting.py 的程式碼如下圖所示。

```python
self.model.eval()
with torch.no_grad():
    img = image.to(self.device)
    _, z_indices = self.model.encode_to_z(img) #z_indices: masked tokens (b,16*16)
    mask_num = mask_b.sum() #total number of mask token
    z_indices_predict=z_indices
    mask_bc=mask_b
    mask_b=mask_b.to(device=self.device)
    mask_bc=mask_bc.to(device=self.device)

    #iterative decoding for loop design
    #Hint: it's better to save original mask and the updated mask by scheduling separately
    for step in range(self.total_iter):
        if step == self.sweet_spot:
            break
        ratio = self.model.gamma(step/self.total_iter) #this should be updated

        z_indices_predict, mask_bc = self.model.inpainting(z_indices_predict, mask_bc, mask_num, ratio)
```

3. Experimental results

A. The best testing FID

我在 training_transformer.py 中使用的 hyperparameters 為 epoch = 100 以及 learning rate = 0.0001。Transformer Loss Curve 如下圖所示，可以看到 train loss 跟 valid loss 都會呈現震盪的狀態，我認為可能是因為沒有使用最恰當的 training strategy 才會讓 train loss 跟 valid loss 不會隨著 epoch 增加而有穩定下降的情況。



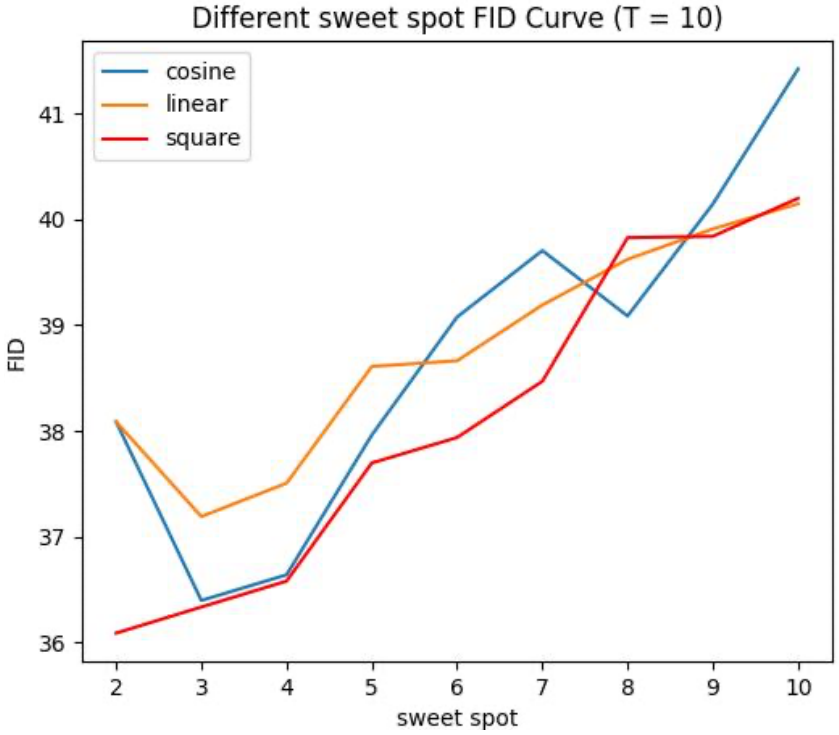
我在 MVTM 中使用 Cosine Mask Scheduling、Total Iteration = 10、Sweet Spot = 6 時所產生的 imga、mask、result 分別如下所示

我在 MVTM 中使用 Sqaure Mask Scheduling、Total Iteration = 11、Sweet Spot = 3 的設定在 Test Data 中可以獲得最好的 FID = 36.019234578171904

```
(py3d) ibm@ibm:~/F/mark/DLP/lab5/faster-pytorch-fid$ python fid_score_gpu.py --predicted-path ../demo/
2/pic/ --device cuda:2
747
100%|                                                                                | 15/15 [00:02<00:00,  6.50it/s]
100%|                                                                                | 15/15 [00:01<00:00,  8.02it/s]
FID:  36.019234578171904
```

B. Different mask scheduling parameters setting

我在 MVTM 中分別使用 Cosine、Linear、Square 三種不同的 Mask Scheduling 在固定 Total Iteration = 10 的情況下，針對 Sweet Spot = 2 ~10 的設定下分別進行了實驗，從下圖可以看到 Square Mask Scheduling 的表現大部分都是最好的，Cosine 跟 Linear 則是會因為 sweet spot 取值的不同而互有高低。以最佳表現的話會是 Square 稍微優於 Cosine，Linear 則與兩者有一段距離。詳細的 FID Curve 及 Best FID 表格如下。



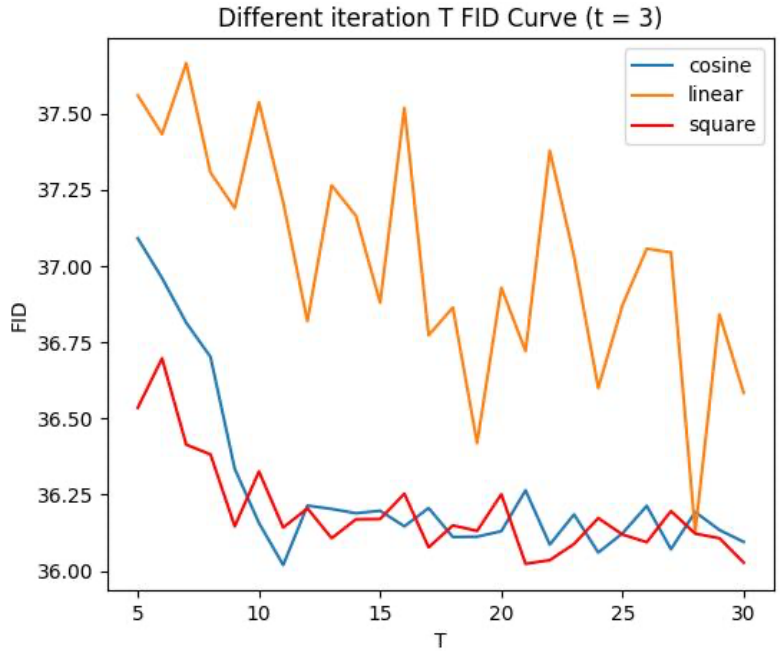
	Cosine	Linear	Square
Best FID	36.395	37.189	36.086
Sweet spot	3	3	2

Best FID in different Mask Scheduling Method

除了調整 Sweet Spot 外，我同樣使用 Cosine、Linear、Square 三種不同的 Mask Scheduling，嘗試固定 Sweet Spot = 3，設定 Total Iteration = 5 ~ 30，藉以觀察其變化。由下頁圖可以看到 Cosine 跟 Square 的 FID 都會比 Linear 好上許多，Cosine 跟 Square 的表現則是走勢很接近，以最佳表現的話會是 Cosineru 幾乎和 Square 一樣，Linear 則比兩者稍微差一些，但三者的表現都會比剛剛固定 Total Iteration = 10 的時候來的好。詳細的 Best FID 表格及 FID Curve 如下。

	Cosine	Linear	Square
Best FID	36.019	36.128	36.023
Total Iteration	11	28	21

Best FID in different Mask Scheduling Method (T = 10)

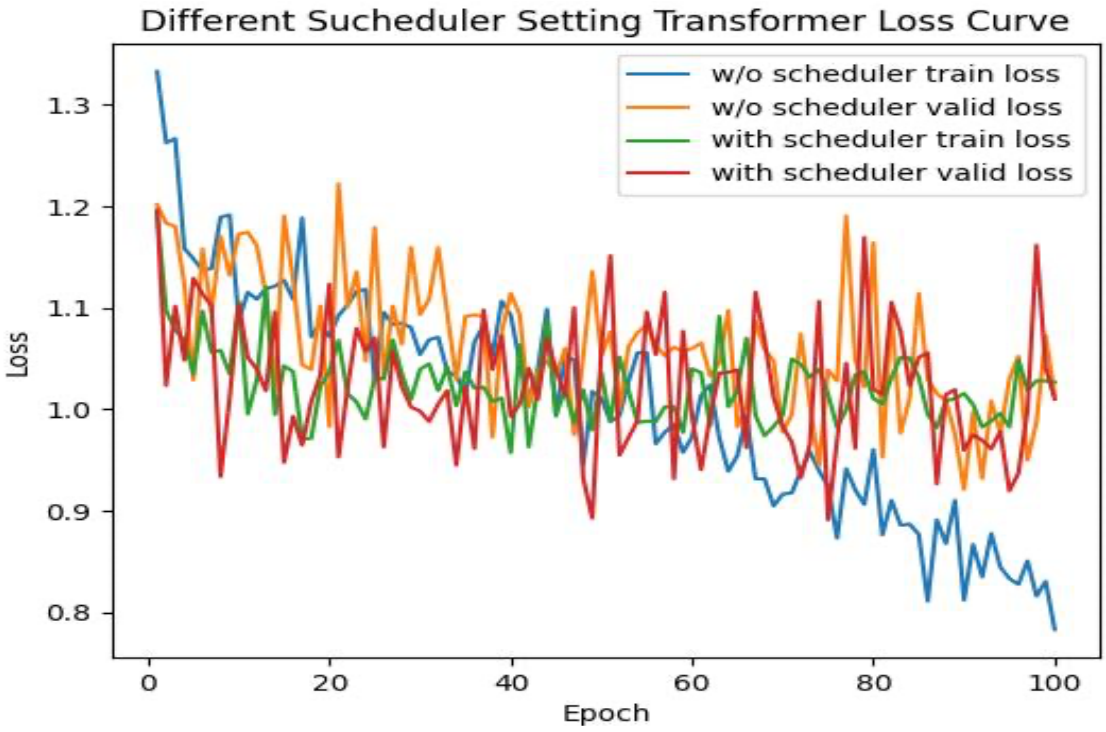


	Cosine	Linear	Square
Best FID	36.019	36.128	36.023
Total Iteration	11	28	21

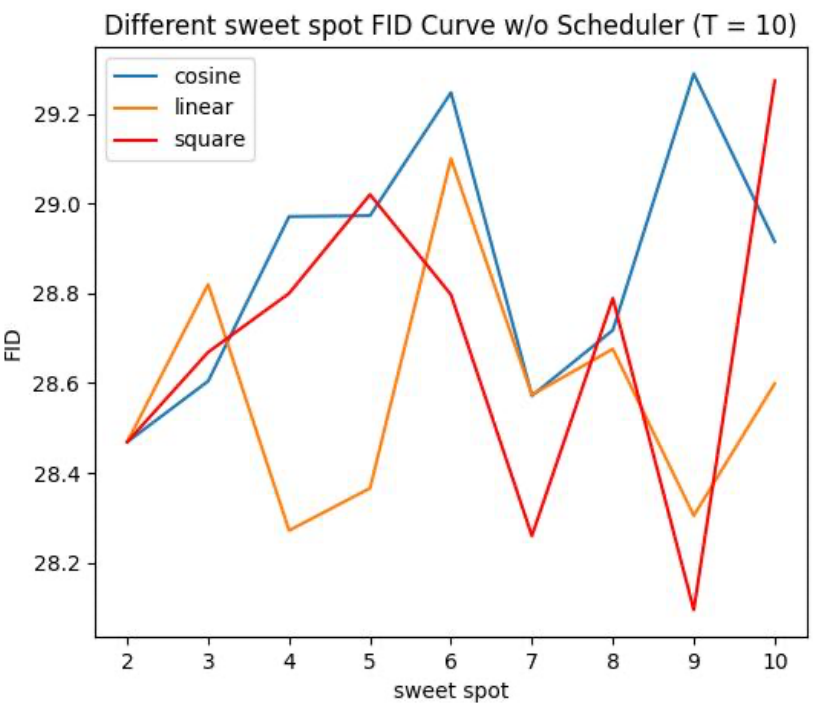
Best FID in different Mask Scheduling Method (t = 3)

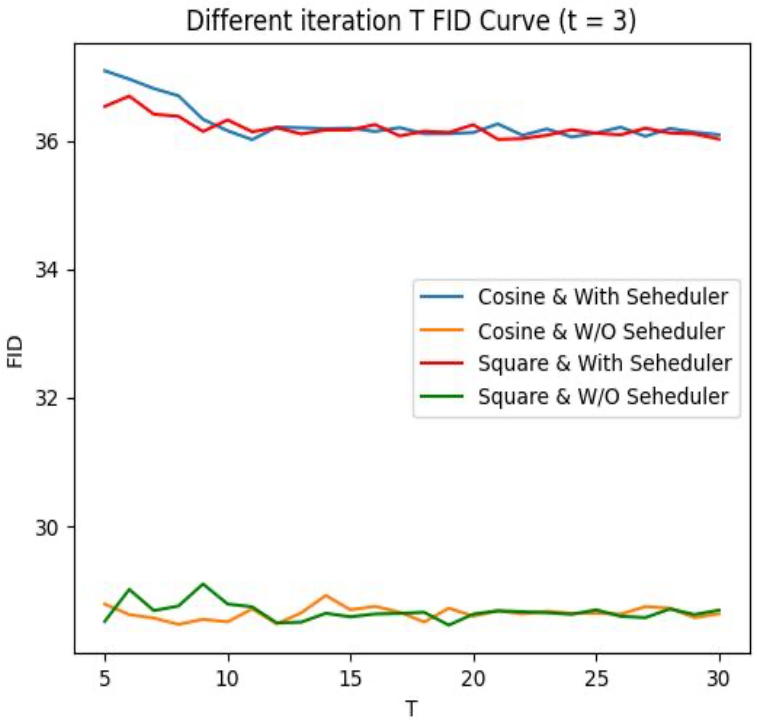
4. Discussion

我嘗試了不使用 ExponentialLR Scheduler 來進行訓練，讓 training 過程中的 learning rate 一直保持在 0.0001 而不會向下遞減，由下圖可以看到不使用 Scheduler 的 train loss 有逐步下降的趨勢，但 valid loss 卻還是震盪狀態，我認為可能是出現了 overfitting 的情況，而有使用 Scheduler 的 best valid loss 為 0.89132，沒有使用 Scheduler 的 best valid loss 則為 0.9104。



我本來認為沒有使用 Scheduler 的 FID Result 會跟有使用 Scheduler 差不多或者更糟，但實驗結果竟然是沒有使用 Scheduler 的 FID Result 不論在何種設定下 FID 都不會超過 30，並且遙遙領先有使用 Scheduler 的結果，令我不太知道該如何解釋這種情況，但我猜可能是因為 learning rate 已經是很小的 0.0001，因此使用 Scheduler 反而使 learning rate 過小導致無法學習，而沒有使用 Scheduler 能讓 model 較好的學習。下左圖是固定 Total Iteration = 10 的情況下，針對 Sweet Spot = 2 ~10 的實驗結果，下右圖是固定 Sweet Spot = 3，設定 Total Iteration = 5 ~ 30 的實驗結果，最下方兩個表格分別是是固定 Total Iteration 或 Sweet Spot 時 Without Scheduler 的 Best FID 表格。





	Cosine	Linear	Square
Best FID	28.469	28.272	28.095
Sweet Spot	2	4	9

Best FID in different Mask Scheduling Method w/o Scheduler (T = 10)

	Cosine	Square
Best FID	28.474	28.462
Total Iteration	8	19

Best FID in different Mask Scheduling Method w/o Scheduler (t = 3)