Lab3_Report
110550161 張維程

1. Introduction (5%)

在CV領域中, Image Inpainting 的目標是自動修復圖像中受損或遺失的部分, 讓最終影像看起來完整且沒有明顯的修復橫績。這項technique的核心在於從input image 中未受影響的部分推斷出合理的補充內容, 進而填補原本缺乏的部分 。它不僅適用於修復破損照片, 還能用於去除多餘物體或水印, 以及進行各類圖像編輯, 並被廣泛應用於電影和電視的特效製作中。實現這一技術主要依賴DL方法, 尤其是卷積神經網絡(CNN)負責提取圖像中的關鍵特徵, 和生成對抗網絡(GAN)通過對抗式學習生成真實感十足的圖像片段。在Inpaint過程中, 不僅要重現缺失部分的紋理, 更必須確保補全內容與整體圖像在風格和上下文上無縫銜接, 例如在修復人像時, 需要同時兼顧面部細節和各特徵部位的相對比例。隨著技術的進步, 這些模型現已能夠更快速、精確地處理各種圖像修復任務, 呈現出令人滿意的效果。

另一方面, Vision Transformer (ViT) 是一種將 Transformer 架構引入視覺任務的模型, 由 Dosovitskiy 等人於 2020 年提出。ViT 不同於傳統的 CNN 模型, 它將圖像分解成若干固定尺寸的小區塊(patches), 然後將這些區塊展平成一維序列作為輸入, 利用 self-attention 機制捕捉圖像中各部分之間的全局關聯。這使得 ViT 在圖像分類和識別上能夠達到與甚至超越最先進 CNN 模型的性能, 充分展示了 Transformer 架構在視覺領域中的強大潛力。

此外, MaskGIT 是 Google Research 團隊於 2021 年推出的一種自回歸圖像生成模型。該模型結合了 Transformer 的長距離依賴捕捉能力和 GAN 的生成優勢, 專注於生產高質量與高分辨率的圖像。MaskGIT 的基本思路在於將圖像生成任務轉變為一個逐步填充的過程:首先遮掩部分圖像區塊, 然後逐步預測並填補這些區塊, 最終拼湊出完整圖像。這種方法不僅能夠抓住圖像的整體結構, 還能細緻地還原各種細節, 並充分利用 self-attention 技術來理解圖像內部的遠距離依賴關係。在實際應用中, MaskGIT 已在圖像修復、生成和超分辨率重建等任務中展現出卓越的性能。

本次作業主要聚焦於實現 MaskGIT 中的 Multi-Head Self-Attention 模塊、Stage 2 的 Masked Visual Token Modeling (MVTM) Training。通過這次實作, 我從原本只知道 Transformer是甚麼, 到實際操作過 Transformer和 MaskGIT 在 Image Inpainting 中的應用, 也掌握了許多在實際開發中必須注意的細節, 使我從中獲得了寶貴的學習經驗。

2. Implementation Details (45%)

A. The details of your model (Multi-Head Self-Attention)

這一部份實作在layer.py裡面的內容, 實作的內容大致就和spec上給的下圖fig1.相同, 先見對QKV做linear transformation 的 model , 還有最後output前的fc layer 。Forward 的過程如下：先把X投影到Q,K,V的空間之後再把大小轉成(batch size, num_image_token,num_of_heads＝ 16, (768/16)), 大小處理完之後先把Q、K、V 帶到 $softmax(QK^T/\sqrt{d_k})V$ 裡面去算 Scaled Dot-Product Attention。算完之後把 16 個 head 的結果

concat起來，就會有一個shape為 (batch_size, num_image_token, dim) 的輸出。最後，經過fc layer，即可獲得最終輸出。
程式碼在fig2.layers.py中可以看到

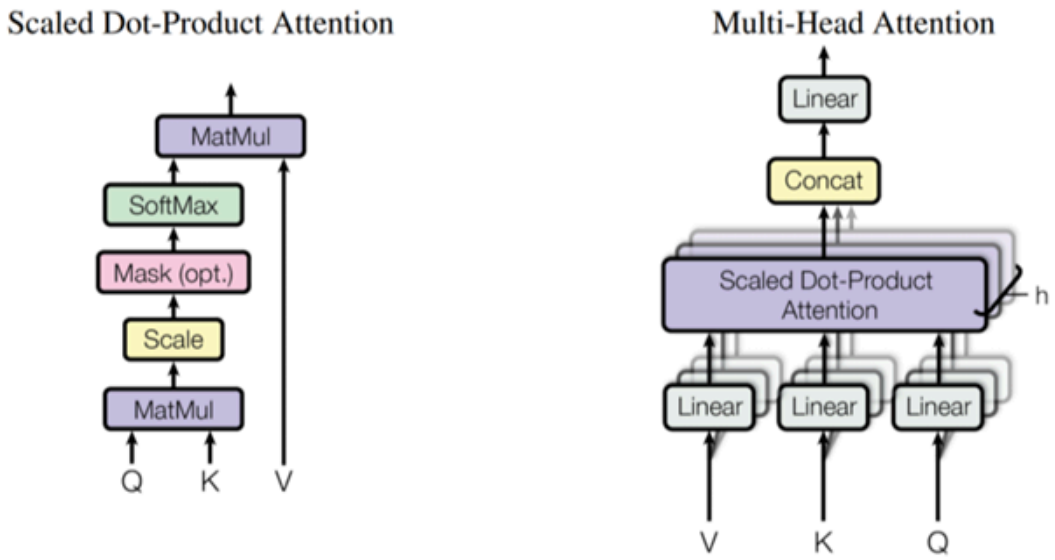


fig1.structure

```python
#TODO1
class MultiHeadAttention(nn.Module):
    def __init__(self, dim=768, num_heads=16, attn_drop=0.1):
        super(MultiHeadAttention, self).__init__()
        self.dim = dim
        self.num_heads = num_heads
        self.q_linear = nn.Linear(self.dim, self.dim)
        self.k_linear = nn.Linear(self.dim, self.dim)
        self.v_linear = nn.Linear(self.dim, self.dim)
        self.attn_drop = nn.Dropout(attn_drop)
        self.proj = nn.Linear(self.dim, self.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, num_image_tokens, dim = x.size()
        # fc layer
        q = self.q_linear(x).view(batch, -1, self.num_heads, self.dim // self.num_heads).transpose(1,2)
        k = self.k_linear(x).view(batch, -1, self.num_heads, self.dim // self.num_heads).transpose(1,2)
        v = self.v_linear(x).view(batch, -1, self.num_heads, self.dim // self.num_heads).transpose(1,2)
        # scaled dot-product attention

        scores = torch.matmul(q, k.transpose(2, 3)) /  math.sqrt(self.dim // self.num_heads)
        scores = nn.functional.softmax(scores, dim=3)
        scores = self.attn_drop(scores)
        output = torch.matmul(scores, v)
        # concat and fully connected layer
        output = output.transpose(1,2).contiguous().view(batch, -1, self.num_heads*self.dim // self.num_heads)
        output = self.proj(output)
        return output
        # raise Exception('TODO1!')
```

fig2.layers.py

B. The details of your stage2 training (MVTM, forward, loss)

這個部分這次lab主要要實做的事VQGAN_Transformer.py 。
encode_to_z(self,x)中 把input x 丟進 VQGAN的encoder並且把她轉成 latent representation zq 和對應的indices。最後再把indices 會被shape 成(view)為合適的格式, 然後與 zq 一起return。
程式碼如fig3.encode_to_z圖中所示。

```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
        # raise Exception('TODO2 step1-1!')
        # return None
```

fig3.encode_to_z

接著是gamma_func的部分, 總共有三種根據mask scheduling分別是cosine,linear, square ,function 會根據不同的mode 以及iterative decoding的過程中的t, 得到 r $=t/T$之後, 把r當作gamma_func 的input 計算gamma ,三個公式分別為

- $cosine: \gamma = cos(\pi r/2)$
- $linear: \gamma = 1 - r$
- $square: \gamma = 1 - r^2$

```python
##TODO2 step1-2:
    def gamma_func(self, mode="cosine"):
        """Generates a mask rate by scheduling mask functions R.

        Given a ratio in [0, 1), we generate a masking ratio from (0, 1].
        During training, the input ratio is uniformly sampled;
        during inference, the input ratio is based on the step number divided by the total iteration number: t/T.
        Based on experiements, we find that masking more in training helps.

        ratio:   The uniformly sampled ratio [0, 1) as input.
        Returns: The mask rate (float).

        """
        print("mask scheduling function: ", mode)
        if mode == "linear":
            return lambda r : 1 - r
            # raise Exception('TODO2 step1-2!')
            # return None
        elif mode == "cosine":
            return lambda r : np.cos(r * np.pi / 2)
            # raise Exception('TODO2 step1-2!')
            # return None
        elif mode == "square":
            return lambda r : 1 - r*r
            # raise Exception('TODO2 step1-2!')
            # return None
        else:
            raise NotImplementedError
```

fig4.gamma_func

接下來是forward function 的部分，程式碼如fig5.forward圖中所示。
先把input x 過 self.encode_to_z(x) 轉成 h*w 個 token，並對應到 codebook 中最接近的向量，得到作為 gt 且形狀為 (batch_size, h*w)的 z_indices。然後隨機決定要 mask 的 token 數量 r，其中 0 < r < h*w，並從 h*w 個 token 中隨機選取 r 個作為 masked token，這些索引存入 sample（形狀為 (batch_size, r)）。然後，建立一個與 z_indices 相同形狀的 mask，初始時所有值皆為 False，接著根據 sample 中的索引將對應位置設為 True。
接下來，創建 masked_indices（形狀為 (batch_size, h*w)），其中所有值皆為 1024，用來表示 masked token 的索引。之後，根據 mask 更新 a_indices：未被 mask 的 token 保持原值（範圍 0～1024），被 mask 的 token 則設為 1024。最後，透過 self.transformer(a_indices) 生成 logits，其形狀為 (batch_size, h*w, 1025)，作為模型的最終output。

```python
#TODO2 step1-3:
    def forward(self, x):

        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)
        # masked_indices (batch_size, h*w)
        a_indices = mask * z_indices + (~mask) * masked_indices
        # a_indices (batch_size, h*w)
        logits = self.transformer(a_indices)
        # logits (batch_size, h*w, 1025)

        return logits, z_indices
```

fig5.forward

train_one_epoch()中，我有使用torch.cuda.amp.GradScaler() 啟用混合精度訓練，來加速計算並減少顯存使用。，一開始先初始化 total_loss ，call model.train()之後在每個 batch 的訓練過程中，先把input images 移到self.args.device，並把optimizer的梯度歸零 (self.optim.zero_grad())。然後，使用 torch.cuda.amp.autocast() 啟用自動混合精度，以 float16 進行部分計算，以提升效率。模型前向傳播時，產生 predictions 和 targets，接著將 predictions reshape，使其符合cross entropy loss的輸入格式，並計算 loss。接著用scaler.scale(loss).backward() 計算梯度，並透過 scaler.step(self.optim) 來更新模型參數。最後，調用 scaler.update() 來調整 GradScaler 的scale factor。過程中我使用的optimizer是Adam，沒有使用scheduler。程式碼如fig6.train_one_epoch圖中所示。

```python
def train_one_epoch(self, train_loader):
        self.model.train()
        total_loss = 0
        progress_bar = tqdm(train_loader, total=len(train_loader), desc=f"Epoch {epoch}")
        scaler = torch.cuda.amp.GradScaler()

        for batch_idx, images in enumerate(progress_bar):
            images = images.to(self.args.device)
            self.optim.zero_grad()


            with torch.cuda.amp.autocast():
                predictions, targets = self.model(images)
                predictions = predictions.view(-1, predictions.size(-1))
                targets = targets.view(-1)
                loss = F.cross_entropy(predictions, targets)

            scaler.scale(loss).backward()
            scaler.step(self.optim)
            scaler.update()

            total_loss += loss.item()
            progress_bar.set_postfix(loss=f"{loss.item():.3f}")

        average_loss = total_loss / len(train_loader)
        return average_loss
```

fig6.train_one_epoch

eval_one_epoch大致上和train_one_epoch相同, 只是不更新model weight。
程式碼如fig7.eval_one_epoch圖中所示。

```python
def eval_one_epoch(self, val_loader):
    self.model.eval()
    total_loss = 0
    progress_bar = tqdm(val_loader, total=len(val_loader), desc="Validation")

    with torch.no_grad():
        for batch_idx, images in enumerate(progress_bar):
            images = images.to(self.args.device)

            # Forward pass
            predictions, targets = self.model(images)
            predictions = predictions.view(-1, predictions.size(-1))
            targets = targets.view(-1)

            # Compute loss
            loss = F.cross_entropy(predictions, targets)
            total_loss += loss.item()

            # Update progress bar
            progress_bar.set_postfix(loss=f"{loss.item():.3f}")
```

fig7.eval_one_epoch

C. The details of your inference for inpainting task (iterative decoding)

這部分是在VQGAN_Transformer.py 檔案中的 inpaiting function

先把輸入的 z_indices 中代表被遮蔽 token 的位置設置為 1024, 再餵進self.transformer裡, 產生形狀為 (batch_size, h*w, 1025) 的 logits, 代表每個位置對應 1025 個可能label的預測分佈。之後透過 softmax 將 logits 轉換為各label的機率分佈, 然後挑出每個位置上機率最大的label及其機率值, 分別儲存在 z_indices_predict 和 z_indices_predict_prob 中。

至於unmasked token, 將其預測機率設定為inf, 確保這些 token 的值保持不變。接著, 加入溫度退火與 Gumbel noise以提升模型的隨機性, 進而計算出每個被遮蔽 token 的信心分數。依照這些信心分數排序後, 挑選出最低的 ratio * mask_num 個 token, 讓它們在下一次迭代中依然保持遮蔽狀態;而剩餘被遮蔽 token 則更新為預測的標籤。同時, 將未遮蔽 token 的位置保持為原始的 ground truth。最後, 此函式回傳update後的 z_indices_predict 與 mask_bc。程式碼如fig8.inpainting圖中所示。

```
##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)
        # logits (batch_size, h*w, 1025)

        #Apply softmax to convert logits into a probability distribution across the last dimension.
        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 (batch_size, h*w)
        # z_indices_predict (batch_size, h*w)
        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 tokens are not affected by the transformer's prediction
        #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 not masked to the predicted tokens
        _, 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
```

fig8.inpainting

在inpainting.py 中的inpainting function, 和原本給的code相異之處只有第3,17行之處而已, 其他就是原本給的code裡的內容, 所以只擷取了相異的部分我。

第3行就是把 input img 2u.餵給self.model.encode_to_z(img)產生z_indices。

第17行就是在iterate過程中, 將該iteration的step透過 self.model.gamma(step /self.total_iter)產生mask ratio之後, 再把output丟進self.model.inpainting(z_indices_predict,mask_bc,mask_num, ratio)產生 z_indices_predict 跟 mask_b。實作 inpainting.py 中的inpainting function 中的相異處之程式碼如下圖所示。

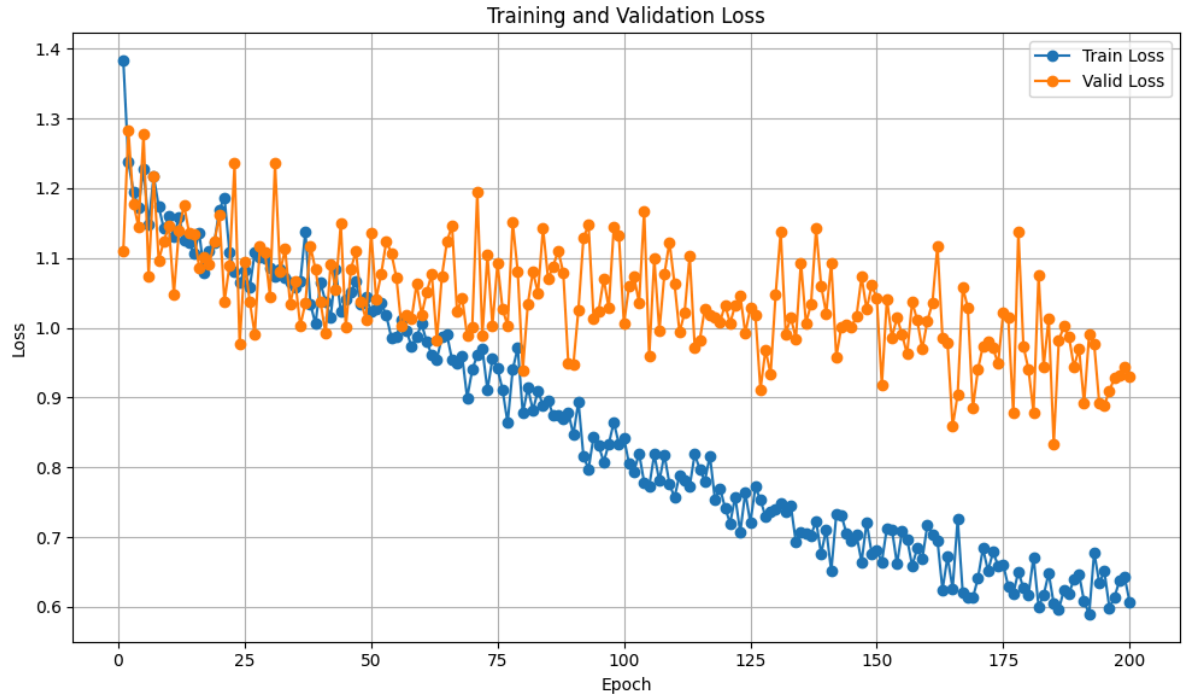
```
self.model.eval()
        with torch.no_grad():
            _, z_indices = self.model.encode_to_z(image.to(device=self.device)) #z_indices: masked tokens (b,16*16)
            mask_num = mask_b.sum() #total number of mask token
            z_indices_predict=z_indices
            mask_bc=mask_b
 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

            # raise Exception('TODO3 step1-1!')
            ratio = 0
            #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
```

fig9.inpainting.py

3. Discussion(bonus: 10%) A. Anything you want to share

再實作過程中原本我有用 ExponentialLR Scheduler +ADAM 去train, 但是使用ExponentialLR Scheduler之後FID反而比不使用還要高, 只使用ADAM可以落在28.xx左右, 但是使用ExponentialLR Scheduler大約會在31.xx。後來去查原因, 大概是因為我用的lr相對來說已經算小(1e-4), 然後再使用ExponentialLR, 會讓lr下降的太快, 所以lr太快衰減, 後期其實model沒什麼再更新。

我有把loss curve做出來, 如下圖, 可以看到train loss大致趨勢是有往下降的, valid loss的大致方向也有往下, 只是沒有那麼明顯, 所以我在推測其實是有一點overfit了, 所以在training dataset loss一直減少, 但是validation dataset 就沒有那麼顯著。



4. Experiment Score (50%)

Part1: Prove your code implementation is correct (30%)

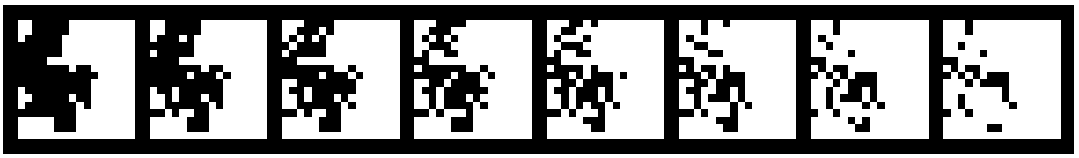
底下列出的result皆為使用了自動混合精度, 以 float16 進行部分計算的結果。
hyper parameters： epoch：200 ， lr ：1e-4 , iteration：10 ， sweetspot：8

- cosine

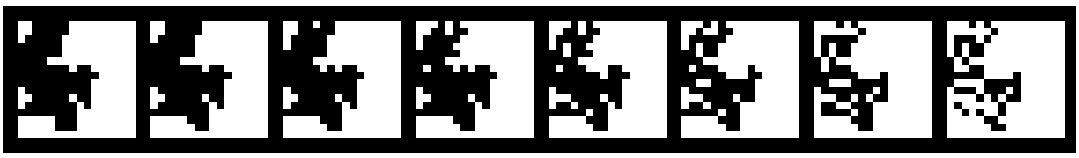
Mask in latent domain	

Predicted image	

- linear

Mask in latent domain	
Predicted image	

- square

Mask in latent domain	
Predicted image	

Part2: The Best FID Score (20%)

fig10.fid_score 是 mask scheduling 使用 cosine
且使用了自動混合精度, 以 float16 進行部分計算的結果。其hyper parameters :
epoch : 200 ， lr ：1e-4 , iteration : 10 ， sweetspot : 8
另外下表為同樣的hyperparameters只是更換mask scheduling 的FID score, 可以看出cos的最好

cosine	linear	square
27.438951	27.460169	27.749368

```
(dlp_lab2) winston@gpu7:/project2/winston/lab/lab3/faster-pytorch-fid$ python fid_score_gpu.py
747
100%|                                                  | 374/374 [00:02<00:00, 125.52it/s]
100%|                                                  | 374/374 [00:02<00:00, 129.97it/s]
FID:  27.438951091131003
```

fig10.fid_score

下圖為設定與前述相同所得之result,上排為masked images , 下排為MaskGIT Inpainting Results

fig11.Masked Images v.s MaskGIT Inpainting Results

如果要training,可以照fig12中的的指令打, 因為大部分的args 我直接寫在default裡, 所以指令比較短, 有需要可以自己修改

```
parser.add_argument('--train_d_path', type=str, default="./lab5_dataset/train/", help='Training Dataset Path')
    parser.add_argument('--valid_path', type=str, default="./lab5_dataset/val/", help='Validation Dataset
                                                Path')
      parser.add_argument('--save_path', type=str, default='./transformer_checkpoints/', help='Path to
                                               checkpoint.')
    parser.add_argument('--device', type=str, default="cuda:0", help='Which device the training is on.')
            parser.add_argument('--num_workers', type=int, default=4, help='Number of worker')
        parser.add_argument('--batch-size', type=int, default=10, help='Batch size for training.')
  parser.add_argument('--partial', type=float, default=1.0, help='Number of epochs to train (default: 50)')
    parser.add_argument('--accum-grad', type=int, default=10, help='Number for gradient accumulation.')
                                    #you can modify the hyperparameters
        parser.add_argument('--epochs', type=int, default=200, help='Number of epochs to train.')
  parser.add_argument('--save-per-epoch', type=int, default=1, help='Save CKPT per ** epochs(default: 1)')
      parser.add_argument('--start-from-epoch', type=int, default=1, help='Number of epochs to train.')
       parser.add_argument('--ckpt-interval', type=int, default=0, help='Number of epochs to train.')
          parser.add_argument('--learning-rate', type=float, default=1e-4, help='Learning rate.')
```

```
    parser.add_argument('--MaskGitConfig', type=str, default='config/MaskGit.yml', help='Configurations for
                                        TransformerVQGAN')
```

```
(dlp_lab2) winston@gpu7:/project2/winston/lab/lab3$ python training_transformer.py --device cuda:5
```

fig12.training command

如果要執行inpainting.py ,可以照下面的指令打, args 的部份我有直接寫好default, 有需要改再特別改即可(test-maskedimage-path 我是照著spec裡面重新命名為lab5_dataset)

- --load-transformer-ckpt-path, default＝'./transformer_checkpoints/fp16_epoch_200.pt'
- --test-maskedimage-path, default＝'./lab5_dataset/masked_image'
- --test-mask-path, default＝'./lab5_dataset/mask64'
- --sweet-spot, default＝8
- --total-iter , default＝10
- --mask-func, default＝'0'

```
(dlp_lab2) winston@gpu7:/project2/winston/lab/lab3$ python inpainting.py --sweet-spotting.py --sweet-spot 8
```

fig13.inpainting command

如果要測試inpaint完的result之fid, 就到faster-pytorch-fid資料夾底下, 並打
$ python fid_score_gpu.py , 即可。