Deep learning lab3 陳品翰 report

1. Introduction (5%)

   這份 lab 在一開始會用 VQGAN encoder 把圖片轉成 token，之後讓 transformer 去學習 token 之間的相互關係，讓 transformer 可以還原圖片，最後把 transformer 的 output 傳給 VQGAN 的 decoder，讓他還原圖片。

2. Implementation Details

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

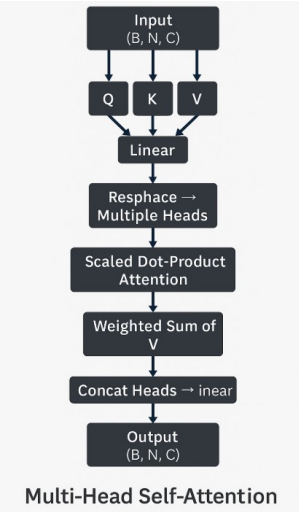
```python
class MultiHeadAttention(nn.Module):
    def __init__(self, dim=768, num_heads=16, attn_drop=0.1):
        super(MultiHeadAttention, self).__init__()
        self.num_heads = num_heads
        self.head_dim = dim // num_heads  # 每個 head 的維度：768 // 16 = 48
        self.scale = self.head_dim ** -0.5  # 用來做 scaled dot-product

        # 三個線性層：Q, K, V 對輸入做線性投影
        self.q_proj = nn.Linear(dim, dim)
        self.k_proj = nn.Linear(dim, dim)
        self.v_proj = nn.Linear(dim, dim)

        # 合併所有 head 之後的線性層
        self.out_proj = nn.Linear(dim, dim)

        # dropout for attention weights
        self.attn_drop = nn.Dropout(attn_drop)
```

```python
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.
    '''
    B, N, C = x.shape  # Batch, Num Tokens (256), Channel Dim (768)

    # 線性投影並 reshape 成多個 head
    q = self.q_proj(x).reshape(B, N, self.num_heads, self.head_dim).transpose(1, 2)  # (B, h, N, d)
    k = self.k_proj(x).reshape(B, N, self.num_heads, self.head_dim).transpose(1, 2)
    v = self.v_proj(x).reshape(B, N, self.num_heads, self.head_dim).transpose(1, 2)

    # Scaled Dot-Product Attention
    attn_scores = (q @ k.transpose(-2, -1)) * self.scale  # (B, h, N, N)
    attn_weights = torch.softmax(attn_scores, dim=-1)
    attn_weights = self.attn_drop(attn_weights)

    out = attn_weights @ v  # (B, h, N, d)
    out = out.transpose(1, 2).reshape(B, N, C)  # 合併所有 heads 回原始 dim

    return self.out_proj(out)  # 最後一層線性投影
```

   Multihead__init__():先計算每個 head 的維度，之後建立三個線性層，最後再合併後 dropout

   Forward:會得到三個值，分別為 q、k、v，之後把它們 reshape 以後，讓每個 head 有自己的 q、k、v，之後每個 head 做 dot-product，之後加權求和得到 context vector，最後合併所有的 head->linear->output。

   流程圖:



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

```python
def train_one_epoch(self, train_loader, epoch, device):
    self.model.train()
    total_loss = 0

    pbar = tqdm(train_loader)
    for i, data in enumerate(pbar):
        x = data.to(device)
        logits, target, mask = self.model(x)  # (B, 256, 1024), (B, 256), (B, 256)
        assert (target < self.vocab_size - 1).all(), "target 包含 mask_token_id!"
        # 只取被遮掉的部分計算 cross entropy
        logits = logits[mask]          # shape (N_masked, 1024)
        target = target[mask]          # shape (N_masked,)

        loss = F.cross_entropy(logits, target)

        loss.backward()

        if (i + 1) % self.args.accum_grad == 0:
            #print(self.args.accum_grad)
            self.optim.step()
            self.scheduler.step()
            self.optim.zero_grad()

        total_loss += loss.item()
        pbar.set_description(f"Epoch {epoch} Loss {loss.item():.4f}")

    return total_loss / len(train_loader)
```

```python
def forward(self, x):

    z_indices = self.encode_to_z(x)  # (B, 256)
    B, N = z_indices.shape
    device = z_indices.device
    mask_ratio = random.uniform(0.4, 0.55)
    mask = torch.rand((B, N), device=device) < mask_ratio

    x_input = z_indices.clone()
    x_input[mask] = self.mask_token_id

    logits = self.transformer(x_input)  # (B, 256, 1024)

    return logits, z_indices, mask
```

```python
optimizer = torch.optim.AdamW(
    self.model.transformer.parameters(),
    lr=self.args.learning_rate,
    betas=(0.9, 0.95),
    weight_decay=0.01
)
```

在我的 train_one_epoch 裡面，我會去使用到 model 的 forward，在 forward 裡面，他會把 training 的圖片先 encode 成 token，之後隨機 mask 40%~55%的圖片，之後把原圖片中 mask 的地方改成 mask token(定為 1024)，之後去給 transformer 學習，並且 return transformer 預測的結果、原本的 ground truth 跟 mask 的位子給 train_one_epoch，之後 train_one_epoch 會去計算 mask 位子的 loss，我的 Optimizer 是用
Adamw，讓 model 避免 overfitting 並且提升 generalization。
另外，因為我在訓練的時候自己定義了 mask token，所以 codebook 裡面也要加上一個位子給 mask，讓他知道目前的位子為 mask。

```python
with torch.no_grad():
    old_weight = self.vqgan.codebook.embedding.weight  # 原本的 [1024, dim]
    new_embed = nn.Embedding(self.vocab_size, old_weight.shape[1])
    new_embed.weight[:-1] = old_weight  # 保留原本的 embedding
    new_embed.weight[-1].zero_()        # 最後一個 index 給 mask token（初始化爲 0）
    self.vqgan.codebook.embedding = new_embed
```

另外我的 transformer 裡面也要有一個位子給 mask，可以看到 tok_emb 有加 1

```python
class BidirectionalTransformer(nn.Module):
    def __init__(self, configs):
        super(BidirectionalTransformer, self).__init__()
        self.num_image_tokens = configs['num_image_tokens']

        self.tok_emb = nn.Embedding(configs['num_codebook_vectors'] + 1, configs['dim'])
        self.pos_emb = nn.init.trunc_normal_(nn.Parameter(torch.zeros(configs['num_image_tokens'], configs['dim'])), 0., 0.02)

        self.blocks = nn.Sequential(*[Encoder(configs['dim'], configs['hidden_dim']) for _ in range(configs['n_layers'])])
        self.Token_Prediction = TokenPredictor(configs['dim'])
        self.LN = nn.LayerNorm(configs['dim'], eps=1e-12)
        self.drop = nn.Dropout(p=0.1)

        #self.bias = nn.Parameter(torch.zeros(self.num_image_tokens, configs['num_codebook_vectors'] + 1))
        self.bias = nn.Parameter(torch.zeros(configs['num_codebook_vectors'] + 1))  # vocab_size
        self.apply(weights_init)
```

C. The details of your inference for inpainting task

```python
class MaskGIT:
    def inpainting(self,image,mask_b,i):
        self.model.eval()
        with torch.no_grad():
            z_indices = self.model.encode_to_z(image)
            z_indices_predict = z_indices.clone()
            mask_bc = mask_b.to(device=self.device)
            original_mask = mask_b.to(self.device).view(B, 16, 16)

            # 初始 mask 畫入第一張（mask 爲 True 表示需要補 → 顯示爲黑色）
            initial_mask = (~original_mask).float().unsqueeze(1).expand(-1, 3, -1, -1)
            maska[:, 0] = initial_mask

            for step in range(self.total_iter):
                if step == self.sweet_spot:
                    break
                z_indices_predict, mask_bc = self.model.inpainting(z_indices_predict, mask_bc, step, self.total_iter)

                # 正確 mask 視覺化邏輯：原本要填且現在還沒填 → 黑色，其餘白色
                current_mask = (original_mask & mask_bc.view(B, 16, 16))
                mask_image = (~current_mask).float().unsqueeze(1).expand(-1, 3, -1, -1)
                maska[:, step + 1] = mask_image

                z_q_origin = self.model.vqgan.codebook.embedding(z_indices).view(B, 16, 16, 256)
                z_q_predict = self.model.vqgan.codebook.embedding(z_indices_predict).view(B, 16, 16, 256)

                mask_reshape = original_mask.unsqueeze(-1)
                z_q = torch.where(mask_reshape, z_q_predict, z_q_origin).permute(0, 3, 1, 2)

                decoded_img = self.model.vqgan.decode(z_q)
                dec_img_ori = torch.clamp(decoded_img * std + mean, 0.0, 1.0)
                imga[:, step+1] = dec_img_ori
```

```python
class MaskGit(nn.Module):
    def inpainting(self, z_indices, mask_bc, t, T):

        z_pred = z_indices.clone()
        z_pred = z_pred.masked_fill(mask_bc, self.mask_token_id)  # 替換成 mask token

        logits = self.transformer(z_pred)  # (B, N, vocab_size)
        probs = torch.softmax(logits, dim=-1)
        pred_probs, pred_ids = torch.max(probs, dim=-1)  # (B, N)

        print("vocab size:", self.vocab_size)
        print("pred_ids max:", pred_ids.max().item())
        assert pred_ids.max() < self.vocab_size, f"pred_ids overflow! max={pred_ids.max().item()}, vocab size={self.vocab_size}"
        pred_ids = torch.clamp(pred_ids, 0, self.vocab_size - 1)

        # Gumbel Noise
        gumbel_noise = -torch.empty_like(pred_probs).exponential_().log()
        #self.choice_temperature = 2.0
        temperature = self.choice_temperature * (1 - t / T)
        confidence = pred_probs + temperature * gumbel_noise

        # 排序 & 要重保留的 index
        _, sorted_indices = torch.sort(confidence, dim=-1)
        gamma = self.gamma(t / T)
        num_mask = int(gamma * N)

        new_mask = torch.ones_like(mask_bc, dtype=torch.bool)
        for b in range(B):
            keep_idx = sorted_indices[b, num_mask:]
            keep_idx = keep_idx.clamp(0, N-1).long()
            new_mask[b, keep_idx] = False  # 保留的設爲 False

        z_pred = torch.where(mask_bc, pred_ids, z_pred)
        return z_pred, new_mask
```

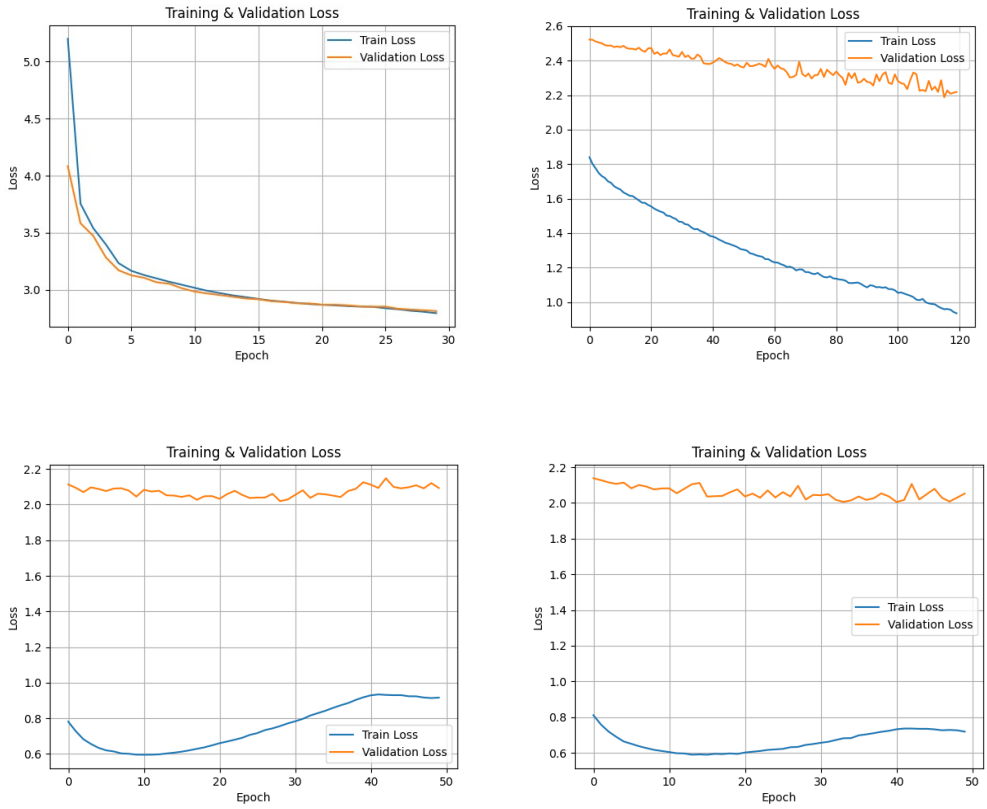
上面的圖片是 inpainting.py 的函式，下面是 VQGAN_Transformer 的 inpainting，當我在畫圖時，會使用 inpainting.py，執行的流程為，一開始先把圖片 encode 成 token 後初始化 z_indices_predict，之後在 iterative 畫圖時，會去呼叫到下面圖片中的 inpainting function，他會把圖片中 mask 的部分轉換為 mask_token_id，之後去給 transformer 做預測，加上 Gumbel Noise 是為了要讓模型有隨機性，之後選擇最有信心的 token 先填上，並且依照 gamma function 決定還要剩下多少 mask，之後把預測好的 token 填回去，回到 inpainting.py，最後把 token decode 回圖像。

3. Discussion(bonus: 10%)

在一開始的時候我是把我 train 時候的 mask 都固定設為 60%，但是我發現我的 loss 都會一直卡在 2.8 左右，所以我之後把我的 mask ratio 調整為

0.4~0.6，最後又調整成 0.4~0.55，我的 loss 可以降為 2.0 左右。

下面是我的 loss/epoch 的圖片，因為我是 load model 後又繼續 train，所以圖片有 4 張。



順序為左上、右上、左下、右下。

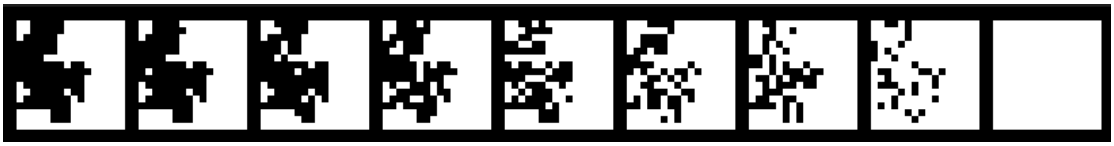
4. Experiment Score (50%)

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

(a) Mask in latent domain

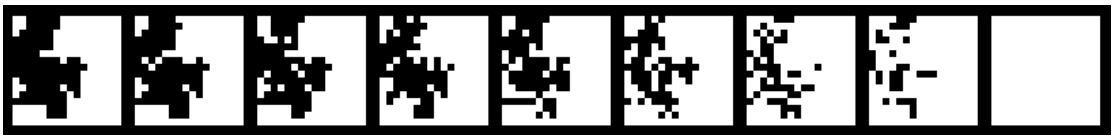
Cosine:

Linear:



Square:

Logarithm:



(b) Predicted image

Cosine:

Linear:

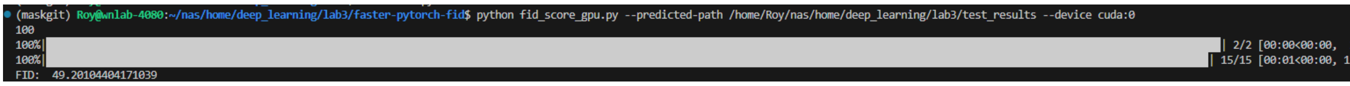
Square:

Logarithm:

Part2: The Best FID Score (20%)

Sweet point:3   total_iter:8

Score:



```
(maskgit) Roy@wnlab-4080:~/nas/home/deep_learning/lab3/faster-pytorch-fid$ python fid_score_gpu.py --predicted-path /home/Roy/nas/home/deep_learning/lab3/test_results --device cuda:0
100
100%|                                                                      | 2/2 [00:00<00:00,
100%|                                                                      | 15/15 [00:01<00:00, 1
FID:  49.20104404171039
```