1. Introduction

本次實驗實作目標將圖片補全，須完成 multi-head attention、訓練細節和拼接不同模型等任務，最後透過 decoder 將 transformer 的猜測結過對應回原始圖片，並用 fid 當作模型效能的指標。

2. Implementation Details

A. 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__()
        assert dim % num_heads == 0

        self.num_heads = num_heads
        self.head_dim = dim // num_heads
        self.scale = math.sqrt(self.head_dim)
        self.W_q = nn.Linear(dim, dim, bias=False)
        self.W_k = nn.Linear(dim, dim, bias=False)
        self.W_v = nn.Linear(dim, dim, bias=False)
        self.W_o = nn.Linear(dim, dim, bias=False)
        self.dropout = nn.Dropout(attn_drop)

    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, num_tokens, dim = x.shape

        Q = self.W_q(x)
        K = self.W_k(x)
        V = self.W_v(x)

        Q = Q.view(batch_size, num_tokens, self.num_heads, self.head_dim)
        K = K.view(batch_size, num_tokens, self.num_heads, self.head_dim)
        V = V.view(batch_size, num_tokens, self.num_heads, self.head_dim)

        attn_scores = torch.matmul(Q.transpose(1, 2), K.transpose(1, 2).transpose(2, 3)) / self.scale
        attn_probs = torch.softmax(attn_scores, dim=3)
        attn_probs = self.dropout(attn_probs)

        attn_output = torch.matmul(attn_probs, V.transpose(1, 2))
        attn_output = attn_output.transpose(1, 2).contiguous().view(batch_size, num_tokens, dim)
        output = self.W_o(attn_output)

        return output
```

設置將 input x 轉換成 QKV 的矩陣，計算好每個 head 的維度(768/16=48)。在 froward 中，生成出 QKV 矩陣後，將 Q 和每一個 K 做內積，此用矩陣乘法表示，而鑲成矩陣的維度為(b, head, token, dim) x (b, head, dim, token) = (b, head, token, token)，接著過 softmax 和 dropout 後和 V 相乘，之後把 head 拼起來再經過 linear 以融合不同 head。

B. stage2 training

a. MVTM

```
def forward(self, image):
    _, z_indices = self.encode_to_z(image)
    batch_size, num_tokens = z_indices.size()

    mask_ratio = 0.4 + torch.rand(1).item() * 0.1
    num_masked = int(mask_ratio * num_tokens)
    mask_token_id = self.mask_token_id
    mask = torch.ones(batch_size, num_tokens, device=image.device).long()
    for i in range(batch_size):
        indices = torch.randperm(num_tokens)[:num_masked]
        mask[i, indices] = 0
    mask_z_indices = mask * z_indices + (1 - mask) * mask_token_id
    logits = self.transformer(mask_z_indices)
    return logits, z_indices
```

將 image 丟進 vqgan 得到 z_indices 後，對 batch size 中每一筆資料做

40%~50%的獨立遮罩，並餵給transformer。

```
def linear_gamma(r):
    return 1.0 - r

def cosine_gamma(r):
    return 0.5 * (1 + math.cos(math.pi * r))

def square_gamma(r):
    return 1 - (r) ** 2

def stair_gamma(r):
    return 1 - (r // 0.1) * 0.1

def god_gamma(r):
    return 0
```

gamma 設計新增階段式和一次全部猜測等模式。

b. training

```
def train_one_epoch(self, train_loader, epoch, args):
    self.model.train()
    train_loss = []
    tq = tqdm(train_loader)
    for i, images in enumerate(tq):
        images = images.to(args.device)
        logits, z_indices = self.model(images)
        loss = F.cross_entropy(logits.reshape(-1, logits.size(-1)), z_indices.reshape(-1))
        loss.backward()
        train_loss.append(loss.item())
        if (i + 1) % args.accum_grad == 0:
            self.optim.step()
            self.optim.zero_grad()
        tq.set_description(f"Epoch {epoch}/{args.epochs}, Training Loss: {np.mean(train_loss):.6f}")
    return np.mean(train_loss)
```

loss 使用了 cross_entropy，由於擔心如果只計算遮罩的 loss，會使模型減弱記住原始圖片的能力。

```python
if torch.cuda.device_count() > 1:
    print(f"Using {torch.cuda.device_count()} GPUs!")
    self.model = nn.DataParallel(self.model)
```

training 時候如果有多顆 GPU，則全部使用。

c. inference for inpainting task

```python
def inpainting(self, image, mask_b, i):
    maska = torch.zeros(self.total_iter, 3, 16, 16) #save all iterations of masks in latent domain
    imga = torch.zeros(self.total_iter+1, 3, 64, 64)#save all iterations of decoded images
    mean = torch.tensor([0.4868, 0.4341, 0.3844],device=self.device).view(3, 1, 1)
    std = torch.tensor([0.2620, 0.2527, 0.2543],device=self.device).view(3, 1, 1)

    ori=(image[0]*std)+mean
    imga[0]=ori #mask the first image be the ground truth of masked image
    print('inpainting...')
    self.model.eval()
    with torch.no_grad():
        #z_indices: masked tokens (b,16*16)
        z, z_indices = self.model.encode_to_z(image)
        mask_num = mask_b.sum() #total number of mask token

        mask_bc=mask_b.clone()
        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 = (step + 1) / (self.total_iter)
            # print("inpainting...")
            z_indices, mask_bc = self.model.inpainting(z_indices.clone(), mask_bc.clone(), mask_num, self.model.gamma(ratio))

            mask_i=mask_bc.view(1, 16, 16)
            mask_image = torch.ones(3, 16, 16)
            indices = torch.nonzero(mask_i, as_tuple=False)#label mask true
            mask_image[:, indices[:, 1], indices[:, 2]] = 0 #3,16,16
            maska[step]=mask_image
            shape=(1,16,16,256)
            z_q = self.model.vqgan.codebook.embedding(z_indices).view(shape)
            z_q = z_q.permute(0, 3, 1, 2)
            decoded_img=self.model.vqgan.decode(z_q)
            dec_img_ori=(decoded_img[0]*std)+mean
            imga[step+1]=dec_img_ori #get decoded image

        ##decoded image of the sweet spot only, the test_results folder path will be the --predicted-path for fid score calculation
        vutils.save_image(dec_img_ori, os.path.join("test_results", f"image_{i:03d}.png"), nrow=1)

        #demo score
        vutils.save_image(maska, os.path.join("mask_scheduling", f"test_{i}.png"), nrow=10)
        vutils.save_image(imga, os.path.join("imga", f"test_{i}.png"), nrow=7)
```

inpainting.py 當中，先將圖片經過 vqgan 模型，交給 VQGANTransformer 做一定比例的猜測(0~1023 分類)，慢慢跌代將圖片還原。

```python
@torch.no_grad()
def inpainting(self, z_indices, mask, mask_num, ratio):
    ismask =  mask == True
    nomask = mask == False
    z_indices_mask = ismask * self.mask_token_id + (~ismask) * z_indices
    logits = self.transformer(z_indices_mask)
    probs = torch.softmax(logits[:, :, :-1], dim=-1)
    z_indices_predict_prob, z_indices_max = probs.max(dim=-1)
    z_indices_predict_prob[nomask] = torch.inf
    g = -torch.log(-torch.log(torch.rand_like(probs)))
    temperature = self.choice_temperature * (1 - ratio)
    confidence = z_indices_predict_prob + temperature * g.max(dim=-1)[0]

    num_to_mask = int(mask_num * ratio)
    sorted_confidence, _ = torch.sort(confidence, dim=-1)
    threshold = sorted_confidence[:, num_to_mask].unsqueeze(-1)
    new_mask = (confidence < threshold).bool()
    return z_indices_max, new_mask
```

在 VQGAN_Transformer.py 當中，先將目前得到的 z_indice 套上 mask 後丟給 transformer，並算出各類別的機率，但由於要猜測的類別不會包含 mask_token_index 所以第 1024 類不能包含在內，之後 sort 取得 confidence 排序將最前面一定比例的遮罩去除變成 transformer 所猜測的值。

3. Discussion

A. Loss Graph

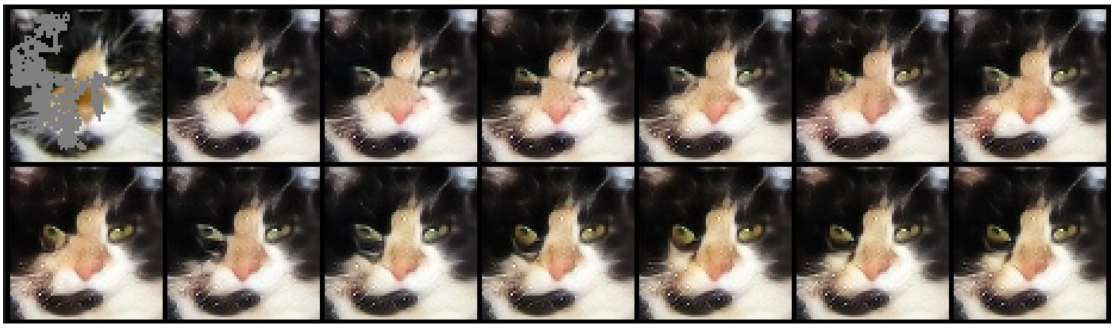
代慢慢修復，因此實驗增加了2種修復策略。

- god：一次全部修復

```
> python ./faster-pytorch-fid/fid_score_gpu.py
747
100%|████████████████████████████████████████| 15/15 [00:01<00:00,  8.03it/s]
100%|████████████████████████████████████████| 15/15 [00:01<00:00,  8.80it/s]
FID:  27.52830742116126
```

- stair：一次修復 10%

```
> python ./faster-pytorch-fid/fid_score_gpu.py
747
100%|████████████████████████████████████████| 15/15 [00:01<00:00,  7.97it/s]
100%|████████████████████████████████████████| 15/15 [00:01<00:00,  8.77it/s]
FID:  27.70896341784629
```



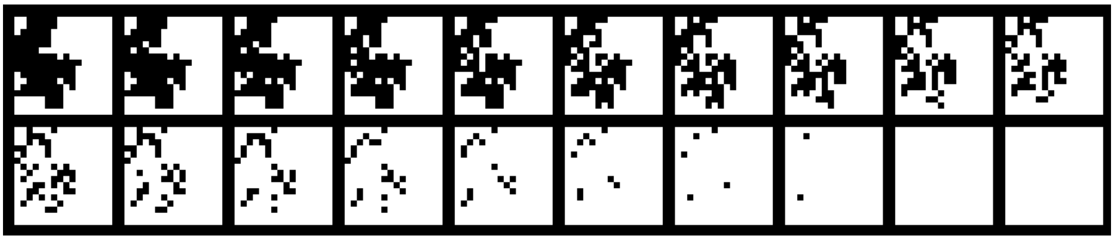
看似策略不比其他的還要差，推測主要的影響因素還是取決於 model。

4. Experiment Score

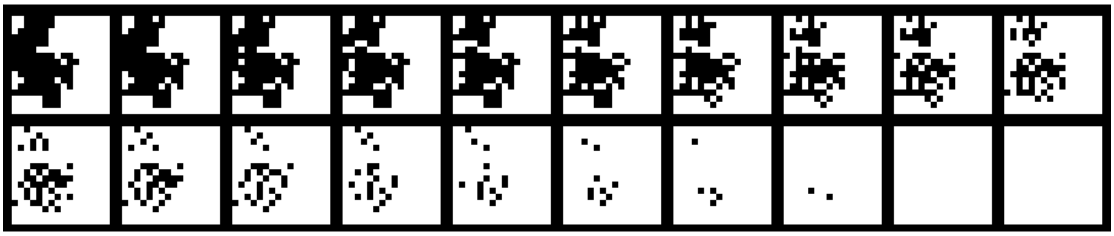
A. iterative decoding

a. Mask in latent domain

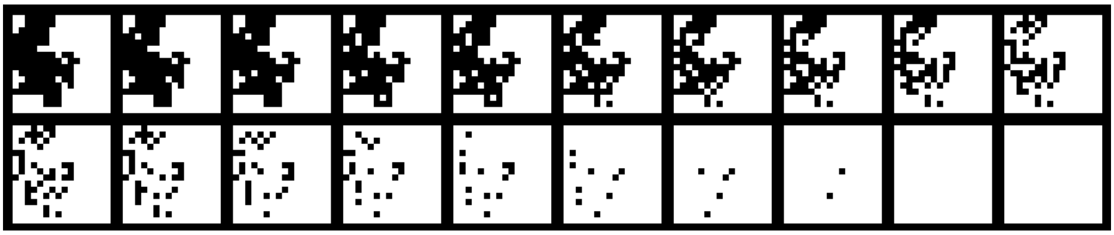
I. cosine



II. linear



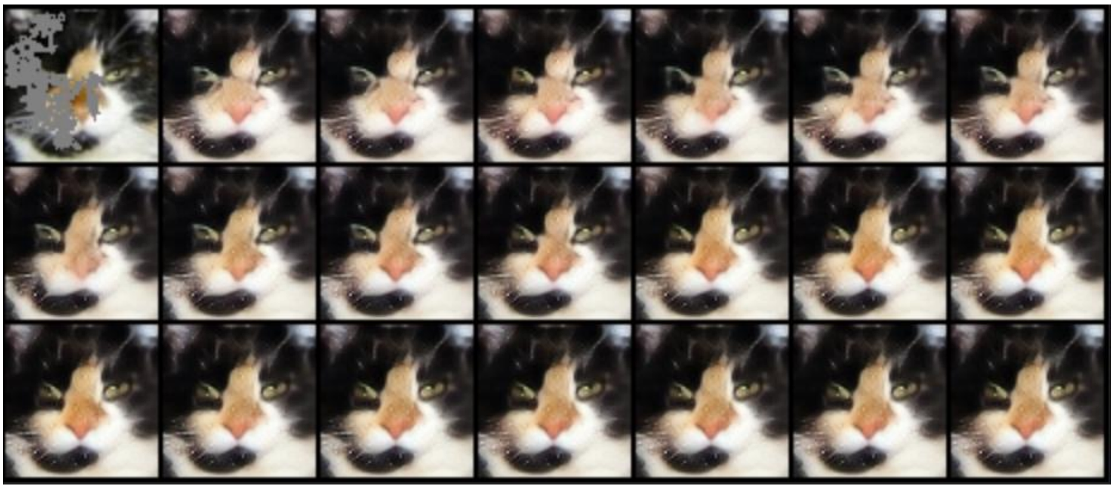
III. square



可以看到不同的 inpainting 策略在最後都有完成所有猜測。

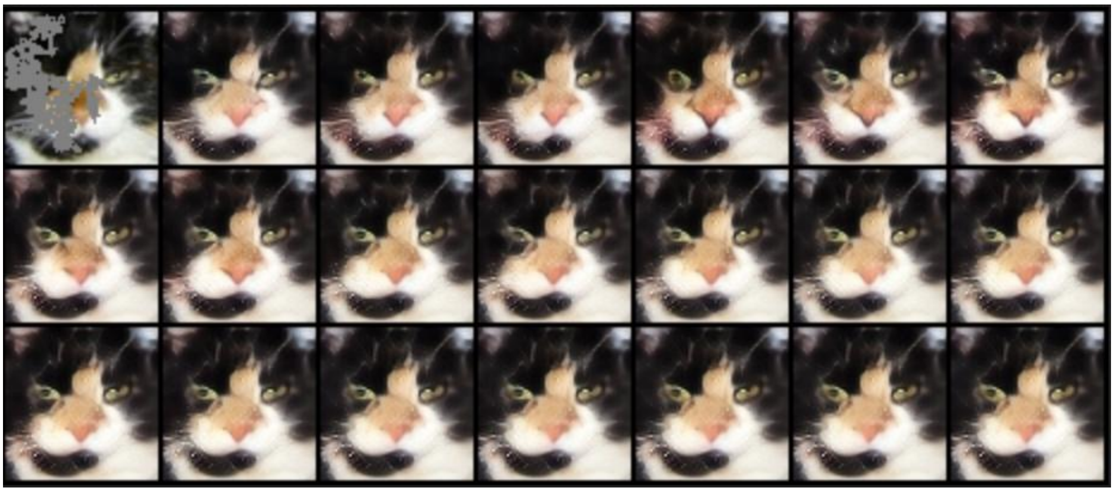
b. Predict Image

I. cosine



II. linear

III. square



B. Best fid score

- 最高的 fid 為 square 27.22

a. cosine

```
> python ./faster-pytorch-fid/fid_score_gpu.py
747
100%|██████████████████████████████████████████████| 15/15 [00:01<00:00,  8.05it/s]
100%|██████████████████████████████████████████████| 15/15 [00:01<00:00,  8.88it/s]
FID:  27.655679654191147
```

b. linear

```
> python ./faster-pytorch-fid/fid_score_gpu.py
747
100%|██████████████████████████████████████████████| 15/15 [00:01<00:00,  7.92it/s]
100%|██████████████████████████████████████████████| 15/15 [00:01<00:00,  8.79it/s]
FID:  27.775280724682545
```

c. square

```
> python ./faster-pytorch-fid/fid_score_gpu.py
747
100%|██████████████████████████████████████████████| 15/15 [00:01<00:00,  8.04it/s]
100%|██████████████████████████████████████████████| 15/15 [00:01<00:00,  8.84it/s]
FID:  27.22421125477223
```

d. masked image v.s. result

C. setting

Training: 在 workspace 下執行

```
python training_transformer.py
```

Inpainting: 在 workspace 下執行

```
python inpainting.py
```

fid score：在 workspace 下執行

```
python ./faster-pytorch-fid/fid_score_gpu.py
```

(不用 cd 進去)