DLP
LAB 5

謝侑哲

112550069

Github:

https://github.com/youzhe0305/NYCU-DLP

1. Introduction

本次 lab 為，使用 MaskGIT 來做 image inpainting 的任務，透過把完整圖片加上隨機 mask 的方式訓練 transformer，使其學習怎麼補洞，另外搭配 pre-trained VQGAN 的 encoder 做 embedding 的離散化。本次針對寵物圖片做訓練及應用。。在評估上則使用 FID 評估。

2. Implementation Details

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

Multi-Head Self-Attention 的實作上，我是直接參照 Transformer 的 paper 來實作，首先要先做一般的 dot product Attention component。用來計算不同 token 之間的關聯性，並作為 token value 融合的權重。

另外因為向量的 dot product 可以用矩陣計算，所以得到的公式如下，其中 dk 是用來防止矩陣變大時，值增長過快:

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

而我的實作如下，基本上就是遵照公式的做法，另外依據 paper 裡面可以加上一個 mask，因此我在 dot product 的後面加了一個 dropout

```python
class Attention(nn.Module):
    def __init__(self, dropout_rate, d_k):
        super().__init__()
        self.dropout = nn.Dropout(dropout_rate)
        self.d_k = d_k

    def forward(self, Q, K, V):
        # Q,K,V: (batch, num_Head, n_token, d_k) or (batch, num_Head, n_token, d_v)
        dot_product = Q @ K.transpose(-2,-1) # (batch, n_token, n_token)
        dot_product = self.dropout(dot_product)
        gate = torch.nn.functional.softmax(dot_product / (self.d_k ** -0.5), dim=-1) # 第i個query對所有的key
        attention = gate @ V # 權重*value，每一個row代表一個query的結果。 output: (batch, num_Head, n_token,
        return attention
```

實作完 Attention 之後，就要實作 Multi-Head 的部分，他是透過把原本只做一次的 Attention 拆分給很多個 Head 去做平行運算，並且這些 Head 在結果上還可以學習到不同的任務，讓 Transformer 能處理的任務更複雜。

Multi-Head 的公式如下，利用 WQ, WK, WV 矩陣把原資料轉乘 Q,K,V 並且把 feature 降為 d_k, d_v 的比較小的維度，分給每個 head 做 attention 的處理。最後再把他合在一起套一個 projection，就得到 Multi-Head 的輸出。

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, ..., \text{head}_\text{h})W^O$$
$$\text{where head}_\text{i} = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

我的實作如下，基本上也就是遵照公式來實作，W 則用 Liner Layer 實現，不過在矩陣運算的部分，為了讓他平行運算，所以 W_1~W_h 我把他全部合在一起，變成一個超長 W 才能做平行運算，得到的 Q,K,V 也會是超長 Q,K,V，可以同時做 Attention，相當於同時做所有 Head。

```
class MultiHeadAttention(nn.Module):
    def __init__(self, dim=768, num_heads=16, attn_drop=0.1):
        self.dropout_rate = attn_drop

        self.mul_W_Q = nn.Linear(self.d_model, self.num_heads * self.d_k, bias=False) # 把W^i全部橫向塞在一起，
        self.mul_W_K = nn.Linear(self.d_model, self.num_heads * self.d_k, bias=False)
        self.mul_W_V = nn.Linear(self.d_model, self.num_heads * self.d_v, bias=False)
        self.mul_W_O = nn.Linear(self.d_v*num_heads, self.d_model, bias=False)
        self.attention = Attention(attn_drop, self.d_k)

    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, n_token, dim = x.shape
        Q = self.mul_W_Q(x).view(batch_size, self.num_heads, n_token, self.d_k) # 乘完之後，把他重構成heads個
        K = self.mul_W_K(x).view(batch_size, self.num_heads, n_token, self.d_k)
        V = self.mul_W_V(x).view(batch_size, self.num_heads, n_token, self.d_v)
        attention = self.attention(Q,K,V) # shape: (batch, num_Head, n_token, d_v)
        concat_attention = attention.view(batch_size,  n_token, self.d_v * self.num_heads) # 把他橫向接回來，d
        output = self.mul_W_O(concat_attention)
        return output
```

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

(1) Encode

在訓練前，要先做 Encode，把原本複雜的圖片 encode 成比較小的 latent code，並且將其離散化。透過離散化的方式，可以降低記憶體的使用及加快速度，因為傳遞上不用傳遞整個 embedding，只需要傳遞 index 就好，並且可以透過該方法穩定訓練，強迫整個 codebook 的 embedding 能學到東西。

我的實作如下，直接使用 pre-trained VQGAN 的 encode，並回傳 index

```
def encode_to_z(self, x):
    codebook_mapping, codebook_indices, q_loss = self.vqgan.encode(x)
    return codebook_indices.view(codebook_mapping.shape[0],-1) # (b*h*w) -> (b,h*w)
```

(2) gamma_func

gamma function 是用來調整每個 iteration 保留的 mask 數量的，吃進去的參數是 t/T，也就是隨著 iteration 會越大。同時隨著 iteration，我們會希望留下的 mask 更少，所以這個 gamma function 應該要是逐步下降的。

我的實作如下，實作上回傳一個 function 以便在 init 就建立，且可以吃進不同 iteration 的參數。實作的 function 有三種 linear, cosin, square root，皆符合上段所述要求。

```
if mode == "linear":
    return lambda ratio : 1 - ratio # 輸入ratio，輸出1-ratio
elif mode == "cosine":
    return lambda ratio : np.cos(ratio * np.pi/2) # 在0~pi/2漸小
elif mode == "square":
    return lambda ratio : 1 - ratio ** 2
```

(3) forward

forward 正式用到 MaskGIT 的特色 MVTM，他的特色在於訓練時不需要 pair data，只要有一張圖片，就可以利用在上面蓋上隨機 mask 的方式製造有漏洞的圖片，再以 Transformer 做修復，最後以原圖做為 ground truth 計算 loss 以及 backpropagation。另外，如 paper 提到的，使用 transformer 主要是

用來預測 token 屬於某一個 embedding 的機率，因此是對 latent code 蓋上 mask，並且輸出 logit，以便之後使用機率計算常用的 cross entropy function。

我的實作如下，基本上就如上面所述，其中 mask 的形狀我是直接用 randint 隨機分配 01，並轉成 boolean 使用。這樣就可以直接透過 element-wise 的矩陣相乘得到 mask

```
forward(self, x): # for training

z_indices = self.encode_to_z(x) #ground truth # (b,h*w)
mask = torch.ones(z_indices.shape).type(torch.LongTensor).to(z_indices.device) * self.mask_token_id # 整張都
position_to_mask = torch.randint(0,2, z_indices.shape).type(torch.bool).to(z_indices.device)
masked_z_indices = position_to_mask * mask + (~position_to_mask) * z_indices

logits = self.transformer(masked_z_indices)  #transformer predict the probability of tokens
return logits, z_indices
```

(4) training & validation

整個訓練的 pipeline 是先用 data loader 載入圖片，然後丟進 MaskGIT 中，encode 成離散化的 embedding，加上 mask，再用 Transformerm 預測 mask 遮住的 embedding。得到預測 embedding 之後，因為預測的是屬於某個 codebook 中的某個 embedding 的機率，因此使用 cross entropy 來計算 loss。而這裡的 loss 不是圖片對圖片的比較，而是 embedding 對 embedding 的比較，因為 decoder 是用 pre-trained VQGAN 的，所以沒必要做訓練，loss 自然也不需要算到全圖。Validation 做的事情基本上一模一樣。

```
def train_one_epoch(self, train_loader, args, ith_epoch): # one epoch

    progress_bar = tqdm(train_loader) # 要能夠獲取object長度才能顯示進度條
    total_loss = torch.zeros(1).to(args.device)
    iterations = 0
    for idx, img in enumerate(progress_bar):

        img = img.to(args.device)
        logits, z_indices = self.model(img) # (b, n_token, n_codebook_vector), (b,h*w)
        loss = F.cross_entropy(logits.view(-1, logits.shape[-1]), z_indices.view(-1)) # input: (B,C) targ
        total_loss += loss
        iterations += 1
        loss.backward()

        if idx % args.accum_grad == 0:
            self.optim.step()
            self.optim.zero_grad()
        progress_bar.set_description(f'Training, Epoch {ith_epoch}:', refresh=False)
        progress_bar.set_postfix({
            'lr': f'{self.scheduler.get_last_lr()}',
            'averge loss':f'{total_loss.item() / iterations}'
        }, refresh=False)
        progress_bar.refresh()
    self.scheduler.step()
    return total_loss.item() / iterations
```

C. The details of your inference for inpainting task

(1) inpainting for one iteration

Inpainting 要做的就是把被 mask 遮住的 embedding 還原，MaskGIT 採用了逐步還原的方法，每次還原信心最高的幾個 mask，令還原有分成好幾個 iteration，這不分先講一個 iteration 怎麼做的。

首先要創建一個跟 embedding 等大的 mask，裡面存的就是 mask 的 index，接著用當前要處理的 mask 形狀的 Boolean 矩陣與之相乘，得到特定形狀的 mask，再把沒有被 mask 遮到的地方套上原本圖片的 embedding。就可以得到黑色缺漏區塊換成 mask 的圖片 embedding。

接著使用 MaskGIT 的 transformer 去預測把 mask 填起來，得到機率之後，把機率套上 Gumbel Distribution 的雜訊，讓圖片的生成更平滑，有點像是 gaussian kernel 模糊化是圖片平滑的概念，也可以增加生成的多樣性。

最後把處理過的機率，取前 n 個最大的保留，剩下的信心不足的就把他對應到的位置重新生成一個 mask，也就是說一個 iteration 完，並不是完全補完，而是補了一部份的 mask 並回傳更小的 mask。實作如下:

```python
def inpainting(self, z_indices, mask_b, ratio, mask_num): # for inference

    mask = torch.ones(z_indices.shape).type(torch.LongTensor).to(z_indices.device) * self.mask_token_id #
    masked_z_indices = mask_b * mask + (~mask_b) * z_indices
    ramaining_mask_num = torch.floor(mask_num * self.gamma(ratio))
    # print(mask_b.sum().item(), mask_num.item(), ramaining_mask_num.item(),self.gamma(ratio), ratio)

    logits = self.transformer(masked_z_indices)
    #Apply softmax to convert logits into a probability distribution across the last dimension.
    prob = F.softmax(logits, dim=-1) # 1,256,1025

    #FIND MAX probability for each token value
    z_indices_predict_prob, z_indices_predict = torch.max(prob, dim=-1) # 產出機率跟對應的類

    z_indices_predict = mask_b * z_indices_predict + (~mask_b) * z_indices

    #predicted probabilities add temperature annealing gumbel noise as confidence
    g =  torch.distributions.gumbel.Gumbel(0, 1).sample(z_indices_predict_prob.shape).to(z_indices.device
    temperature = self.choice_temperature * (1 - self.gamma(ratio))
    confidence = z_indices_predict_prob + temperature * g # (1,256)
    confidence[(~mask_b)] = 1e9
    sorted_confidence = torch.sort(confidence, dim=-1)[0] # 由小到大,(batch,n_token)
    threshold = sorted_confidence[:,ramaining_mask_num.long()] # 0~ramaining_mask_num-1, 機率最小的幾個
    ret_mask = confidence < threshold
```

(2) inpainting for all iterations

得到一個 iteration 之後，就可以做多次 Iteration 修復出完整的圖，其中每次修復剩下的 mask 量是上文提到的 gamma function 決定，參數是 t/T，也就是第 t 個 iteration 除上總共 T 個 iteration，總體而言會越剩越少。在每個 iteration 都會得到一個補完的 embedding，這些就可以在後面丟給 VQAGN 的 decoder 做，讓他能還原成圖，實作如下:

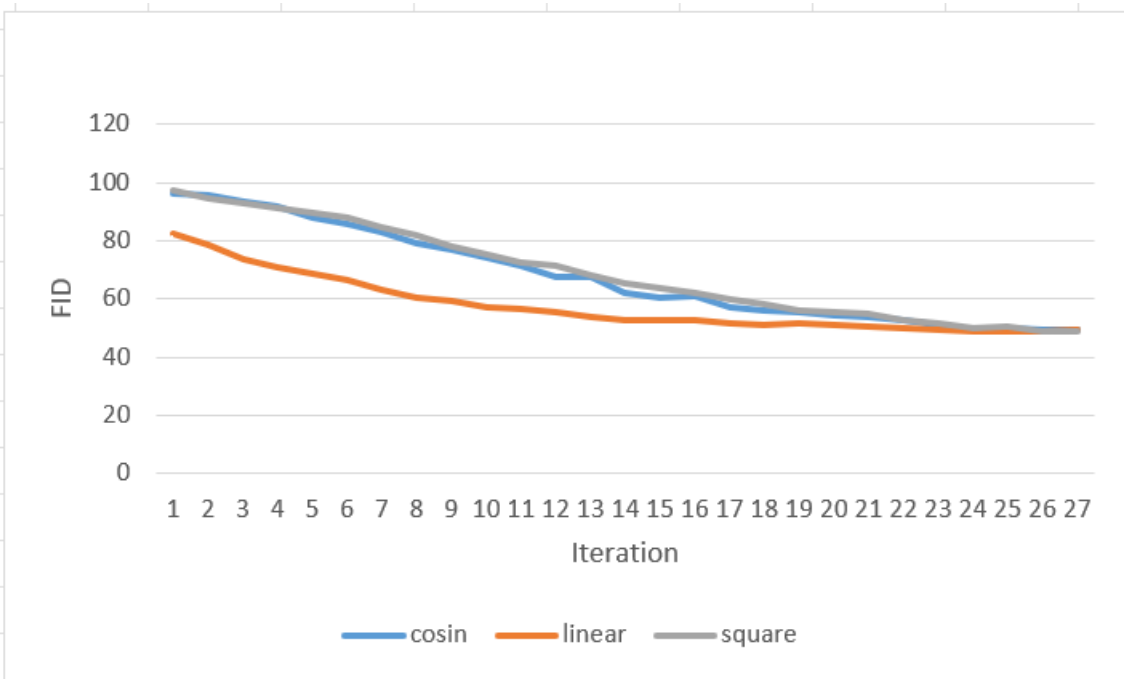
```
def inpainting(self,image,mask_b,i): #MakGIT inference
        #iterative decoding for loop design
        #Hint: it's better to save original mask and the updated mask by scheduling separately
        for step in range(self.total_iter):
            if step == self.sweet_spot:
                break
            ratio = (step+1) / self.total_iter #this should be updated 從1開始 t/T

            z_indices_predict, mask_bc = self.model.inpainting(z_indices, mask_b, ratio, mask_num)
            #static method yon can modify or not, make sure your visualization results are correct
            mask_b = mask_bc
            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_predict).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
```

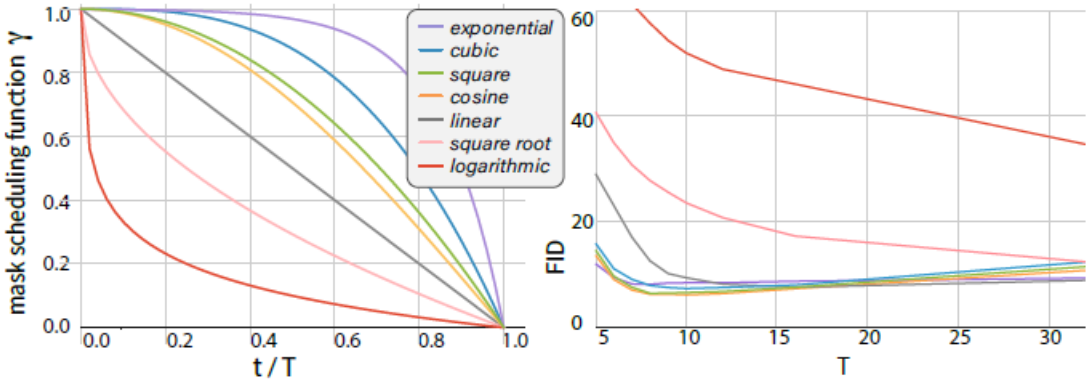
最終得到的效果如下圖，可以拿到每個 iteration 的產出圖片:

3. Discussion

這部分，我想針對使用不同 gamma function 對於每個 iteration 的影響進行討論。下圖使用了 3 種 gamma function 做實驗。可



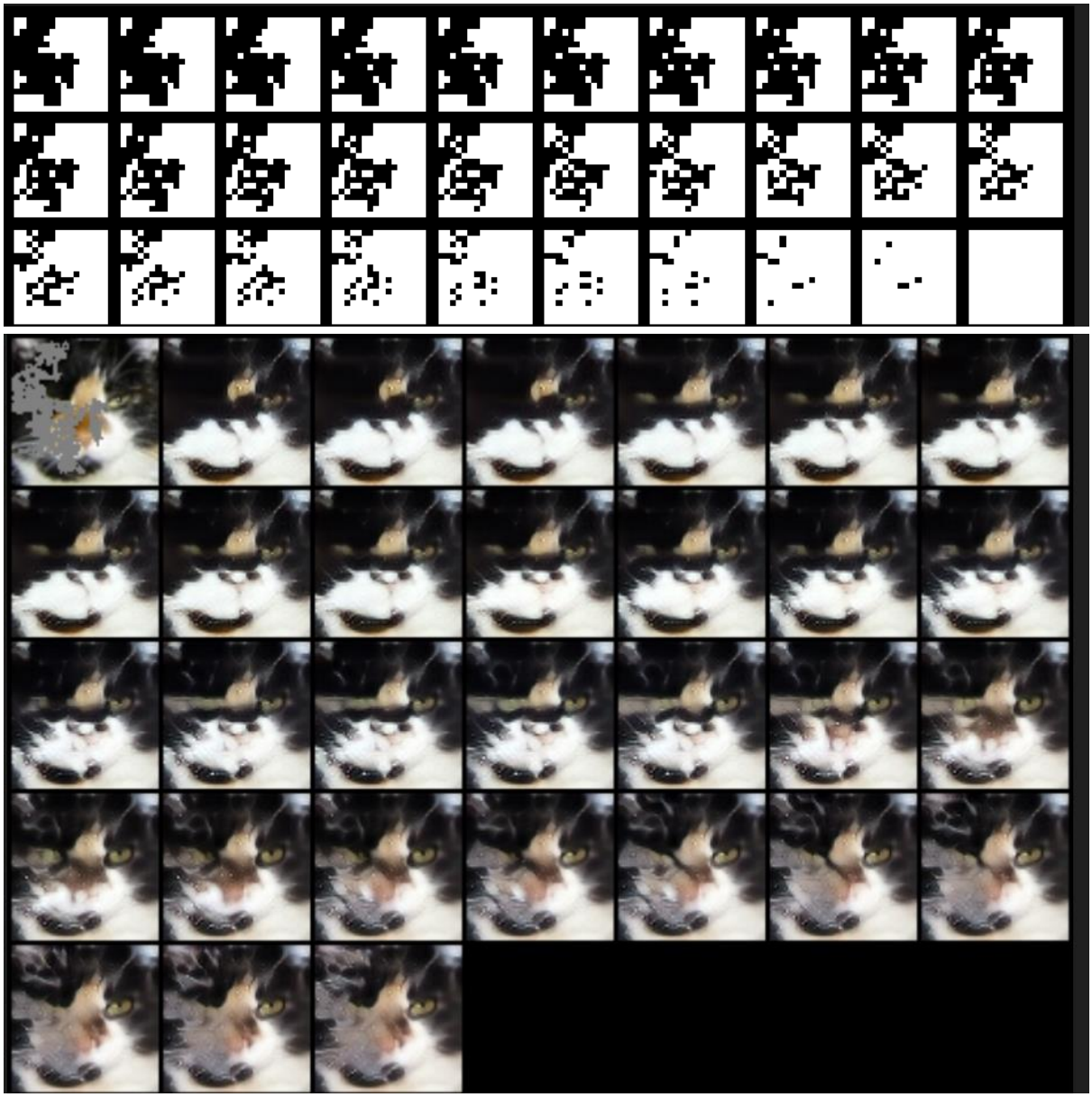
從圖中可以發現，linear 在 FID 下降的速度非常快，而 cosine 與 square 的下降曲線幾乎相同。而收斂狀況則幾乎一樣，這部分的實驗結果，基本上跟論文中的圖片(如下)有些差距。從細部數據來看，linear 一樣收斂得更好，但是 cosine 跟 square 的狀況卻比論文中差一點，論文的圖明顯看出他們兩個的下降速度應該是更快的。而我認為可能的原因是，我的 model 並沒有訓練到非常好，導致兩者都是先慢慢鋪再快速鋪的作法沒有生效，一開始的慢慢鋪並不能給到足夠好的 context，所以下降的速度就沒這麼快。



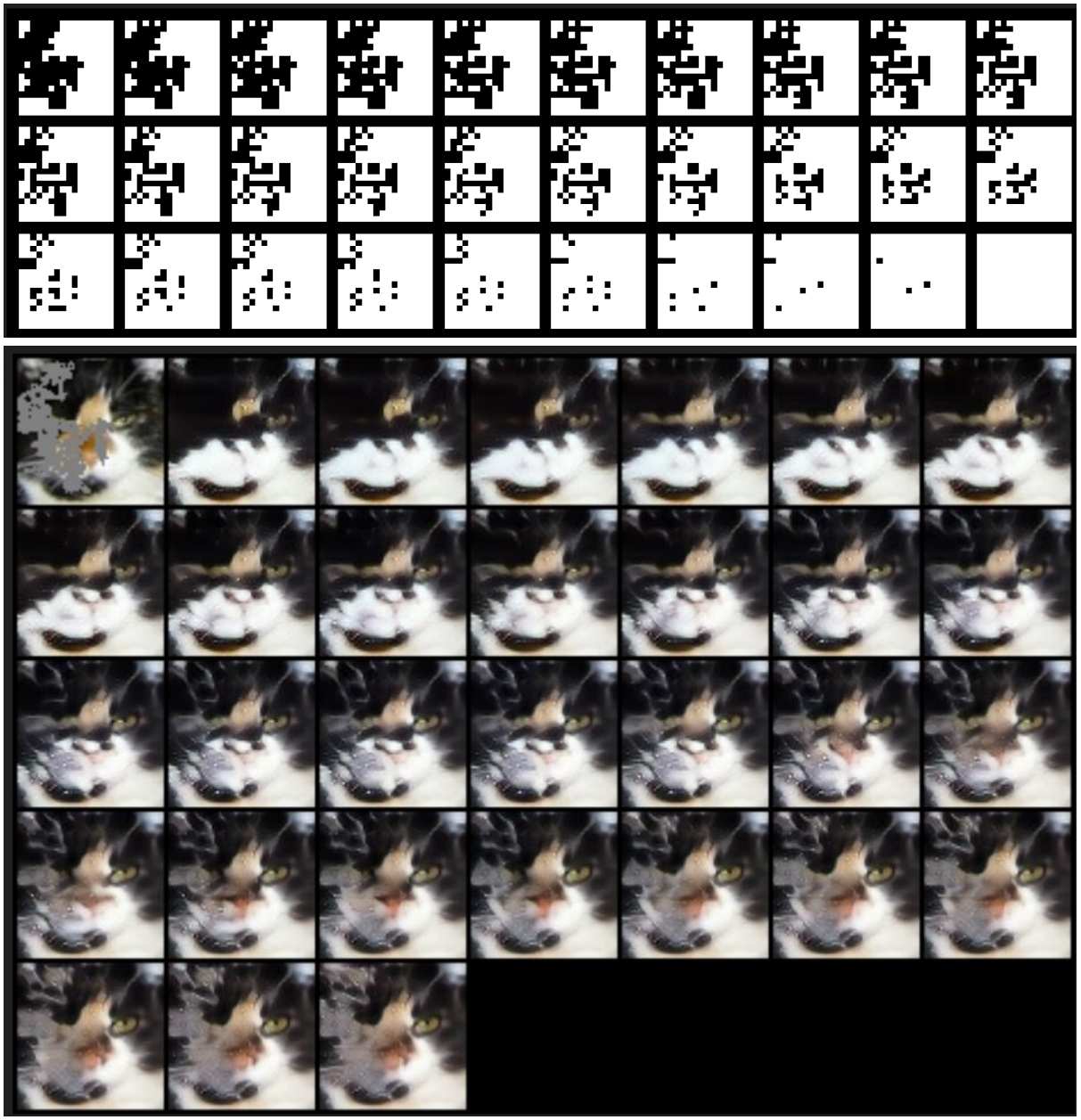
4. Experiment

A. Show Iteration decoding

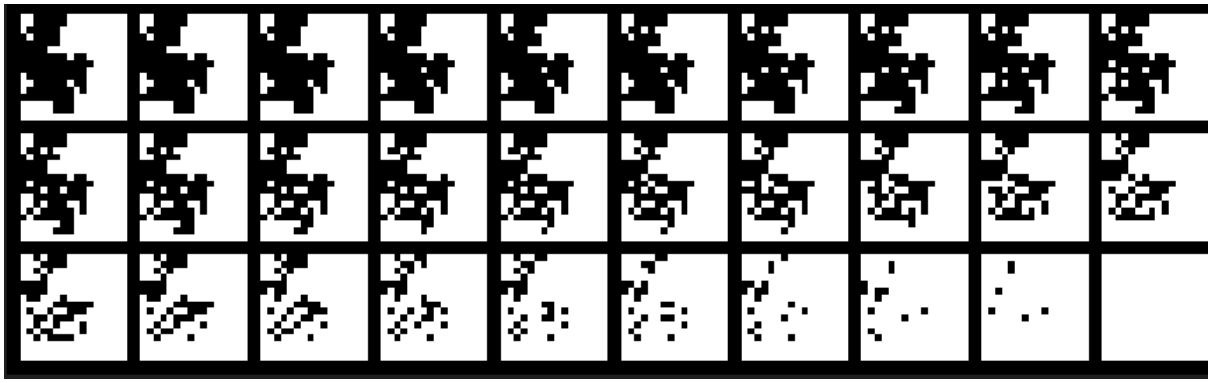
Consie:

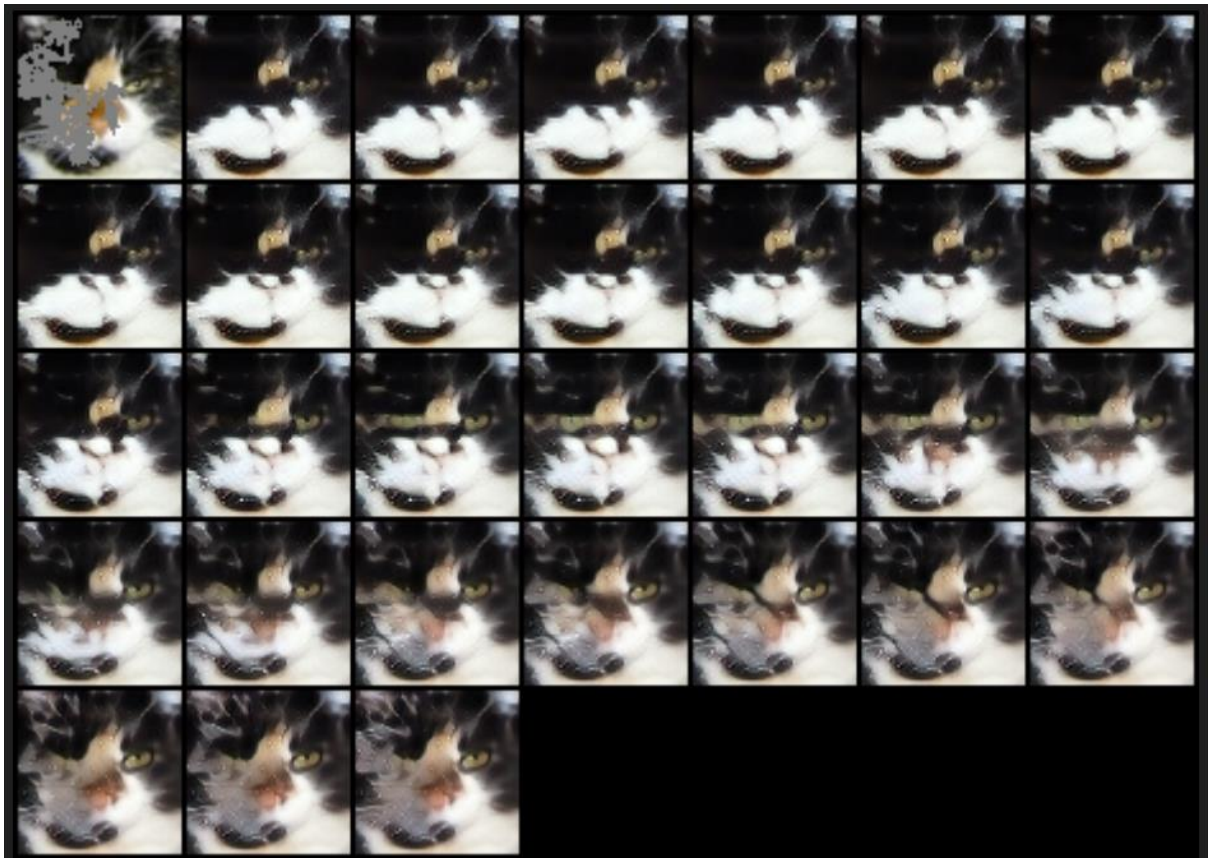


Linear:



Square:





B. The Best FID Score

使用 python3 fid_score_gpu.py --predicted-path ../test_results/iteration_28 --device cuda 指令，對於使用 linear 的第 28 個 iteration 計算 FID，也是我做出的最好結果，結果如下:

```
youzhe0305@gpu1:~/NYCU-DLP/lab5/faster-pytorch-fid$ python3 fid_score_gpu.py --predicted-path ../test_results/iter
ation_28 --device cuda
747
100%|                                                                                           | 15/15 [00:00<00:00, 19.74it/s]
100%|                                                                                           | 15/15 [00:00<00:00, 28.30it/s]
FID:  48.64988486480081
```

生成圖對於 mask 比對圖如下: