

1. Introduction

在本次的實驗報告中詳細的介紹了每個實作的過程，並且在 training 上也使用了兩個不同的 loss 計算方式並且做比對，在後半部的 inpainting 則是以三個不同的 mask function 在一定的 iteration 下對 FID 分數做比對，發現在 linear 的 mask function 且 iteration 為 2 時 FID 分數可以達到最低的分數，並且使用的 Loss 計算方式為直接比較整體的預測而不是如論文中所提到直接以 mask 的作 Loss 計算。

2. Implementation Details

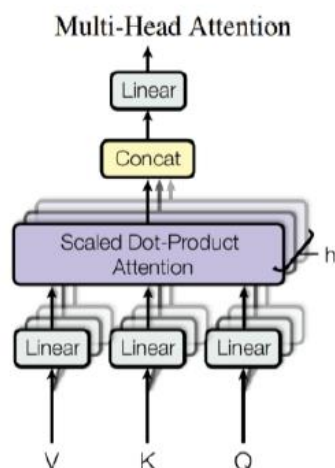
A. The details of your model (Multi-Head Self-Attention)

```

6 #TODO1
7 class MultiHeadAttention(nn.Module):
8     def __init__(self, dim=768, num_heads=16, attn_drop=0.1):
9         super(MultiHeadAttention, self).__init__()
10        self.num_heads = num_heads
11        self.dim = dim
12        self.head_dim = dim // num_heads
13        self.q_linear = nn.Linear(dim, dim)
14        self.k_linear = nn.Linear(dim, dim)
15        self.v_linear = nn.Linear(dim, dim)
16
17        self.out_linear = nn.Linear(dim, dim)
18        self.dropout = nn.Dropout(attn_drop)
19
20
21    def forward(self, x):
22        ''' Hint: input x tensor shape is (batch_size, num_image_tokens, dim),
23            because the bidirectional transformer first will embed each token to dim dimension,
24            and then pass to n_layers of encoders consist of Multi-Head Attention and MLP.
25            # of head set 16
26            Total d_k , d_v set to 768
27            d_k , d_v for one head will be 768//16.
28        '''
29        batch_size, num_image_tokens, dim = x.size()
30
31        q = self.q_linear(x).view(batch_size, num_image_tokens, self.num_heads, self.head_dim)
32        k = self.k_linear(x).view(batch_size, num_image_tokens, self.num_heads, self.head_dim)
33        v = self.v_linear(x).view(batch_size, num_image_tokens, self.num_heads, self.head_dim)
34
35        q = q.transpose(1, 2)
36        k = k.transpose(1, 2)
37        v = v.transpose(1, 2)
38
39        attn_weights = torch.matmul(q, k.transpose(-2, -1)) / (self.head_dim ** 0.5)
40        attn_weights = F.softmax(attn_weights, dim=-1)
41        attn_weights = self.dropout(attn_weights)
42        attn_output = torch.matmul(attn_weights, v)
43        attn_output = attn_output.transpose(1, 2).contiguous().view(batch_size, num_image_tokens, dim)
44        output = self.out_linear(attn_output)
45        return output

```

在 Multi-Head Self-Attention 的 Class 中以下圖做設計



輸入分為 Q(Query)、V(Value) 和 K(Key)，在程式第 31-33 行時會先將輸入的 x 進到對應的 Linear 層並且做 view 的轉換，轉成 $[\text{batch_size}, 256, 16, 48]$ ，其中 256 是輸入 latent space 的 image_token(16x16)，而 16 則是代表 num_heads 數量，48 則是根據輸入的維度(768 第 8 行定義)分配給每個 num_heads(除以 num_heads 數量)得到的數字，為每一個 heads 的維度。

為了去對 num_heads 做運算，因此會先對 V、K、Q 做 transpose，將 num_heads 的維度往前調整(程式 35-37 行)，接著要算每個 head 的 attention weight(注意力分數)，也是下方 formula 所表示的(程式 39-42 行)：

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

為了防止 training 時 overfitting 所以在計算 softmax 之後做一層的 dropout(程式第 40 行)，有效避免 overfitting 和提高泛化能力，做完之後要再次 transpose 一次，因為先前 transpose 過了這次是要為了將其 concatenate 因此在做一次，接著再把它 view 轉成原先的形狀(程式第 43 行)，最後則是經過一層 Linear 作結束。

其中 43 行部分使用 contiguous 是保證記憶體의連續性，然後再使用 view 去做 concatenate 的動作。

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

```

36  ##TODO2 step1-1: input x fed to vqgan encoder to get the latent and zq
37  @torch.no_grad()
38  def encode_to_z(self, x):
39      quant_z, indices, q_loss = self.vqgan.encode(x)
40      indices = indices.view(quant_z.shape[0], -1)
41      return quant_z, indices
42

```

在 38-41 行 `encode_to_z` 時，會先透過 vqgan 的 encoder 拿到 `quant_z`，像是否種對應需要修改地方的 codebook，但是我們並不會使用到，因為 vqgan 是以 pretrained 的 model 去使用的，在 lab05 中只需要對 bidirectional transformer 做訓練，因此在後面後續都是直接以 `indices` 去做選擇，但是從 vqgan 的 encoder 拿到的 `indices` 會是 `batch_size*16*16`，其中 `16*16` 是 latent space 的大小，因此會先轉乘 `[batch_size, 256]` 的形式做輸出，做後續的動作。

```

82  ##TODO2 step1-3:
83  def forward(self, x):
84      _, z_indices = self.encode_to_z(x)
85
86      z_sz = (16, 16)
87      mask_ratio = self.gamma(np.random.uniform())
88
89      mask = self.gen_mask(z_sz, math.floor((z_sz[0] * z_sz[1]) * mask_ratio))
90      mask = torch.tensor(mask, dtype=torch.bool).view(-1).unsqueeze(0).to(z_indices.device)
91      #print("shape:", z_indices.shape)
92      #print("mask:", mask.shape) # 16*16* batch_size
93      # z_indices ground truth
94      # logits transformer predict the probability of token
95
96      codebooks = self.mask_token_id * torch.ones_like(z_indices) # code books
97
98      a_indices = (~mask) * z_indices + mask * codebooks
99
100     logits = self.transformer(a_indices)
101
102     #raise Exception('TODO2 step1-3!')
103     # print(logits.shape, z_indices.shape)
104     return logits, z_indices, mask

```

如上述所說，第 84 行直接省略了取 `quant_z` 的變數，而是透過後續轉成 transformer (第 100 行)，在那之前會先做一系列的動作，首先先設隨機產生一個 ratio，代表 mask 的 ratio，主要是用於隨機生成 mask 的比例，而 `gen_mask` 的寫法如下圖所示，拿到 mask 之後要把 mask 從 np 轉乘 torch 的 tensor (第 90 行)，接著設定 transformer 的參數，第 96 和 98 行分別代表產生 codebook，需要去讓 transformer 學習的部分，如果有 mask 的地方則是使用 codebooks 值 (預設為 1024)，如果沒有的畫則是保留原先的 `z_indices`，最後產生新的 `a_indices` 給 transformer 去進行處理，拿到 logits，代表每個區域的預測機率分布。

```

71     def gen_mask(self, sz, masked_cnt):
72         mask = np.zeros(sz, dtype=np.int32)
73
74         tables = []
75         for i in range(sz[0]):
76             for j in range(sz[1]):
77                 tables.append([i,j])
78         random.shuffle(tables)
79         for i in range(masked_cnt):
80             mask[tables[i][0],tables[i][1]] = 1
81         return mask

```

建立一個 table 儲存每一個 (i,j) 對之後做打亂，根據前 masked_cnt 項去設定 mask 值，回傳給 forward function 中。

```

43     #TODO2 step1-2:
44     def gamma_func(self, mode="cosine"):
45         """Generates a mask rate by scheduling mask functions R.
46
47         Given a ratio in [0, 1), we generate a masking ratio from (0, 1].
48         During training, the input ratio is uniformly sampled;
49         during inference, the input ratio is based on the step number divided by the total iteration number: t/T.
50         Based on experiements, we find that masking more in training helps.
51
52         ratio: The uniformly sampled ratio [0, 1) as input.
53         Returns: The mask rate (float).
54
55         """
56         if mode == "linear":
57             #raise Exception('TODO2 step1-2!')
58             return Lambda r : 1 - r
59         elif mode == "cosine":
60             #raise Exception('TODO2 step1-2!')
61             return Lambda r : np.cos(r * np.pi / 2)
62         elif mode == "square":
63             #raise Exception('TODO2 step1-2!')
64             return Lambda r : 1 - r ** 2
65         else:
66             raise NotImplementedError

```

在 gamma_func 時則是簡單寫對應的 linear、cosine、square function，寫成 lambda 形式可以在後續使用時直接帶入 function。

```

28     def tqdm_bar(self, mode, pbar, loss, avg_loss, lr):
29         pbar.set_description(f"({mode}) Epoch {self.current_epoch}, lr:{lr:.0e}" , refresh=False)
30         pbar.set_postfix(loss=float(loss), avg_loss=float(avg_loss))
31         pbar.refresh()
32     def train_one_epoch(self, train_loader, cur_epoch):
33         self.current_epoch = cur_epoch
34         total_loss = 0
35         cnt = 0
36         self.model.train()
37         for (img) in (pbar := tqdm(train_loader, ncols=180)) : #
38             cnt += 1
39             self.optim.zero_grad()
40
41             img = img.to(self.args.device)
42             logits, z_indices, mask = self.model(img)
43             # [10,1025] | [10,256]
44
45
46             #loss = self.loss(logits.reshape(-1, logits.size(-1)), target.reshape(-1))
47             batch_size, _ = z_indices.size()
48             log_probs = F.log_softmax(logits, dim=-1)
49
50             masked_positions = mask.squeeze(0).repeat(batch_size,1)
51             masked_log_probs = log_probs[masked_positions, z_indices[masked_positions]]
52
53             loss = -masked_log_probs.mean()
54             if (masked_log_probs.shape[0] == 0):
55                 continue
56                 #skip
57             #loss = self.loss(logits,target)
58             total_loss += loss
59             avg_loss = total_loss / cnt
60             loss.backward()
61             self.optim.step()
62
63             self.tqdm_bar('train ', pbar, loss=loss.detach().cpu(), avg_loss=avg_loss, lr=self.scheduler.get_last_lr()[0])

```

在 training 階段時本實驗嘗試了兩個不同的 loss 計算方式，第一個是採用論文中所提到的：

$$\mathcal{L}_{\text{mask}} = - \mathbb{E}_{\mathbf{Y} \in \mathcal{D}} \left[\sum_{\forall i \in [1, N], m_i = 1} \log p(y_i | Y_{\overline{M}}) \right]$$

第二個則是不針對 $Y_{\overline{M}}$, $m_i = 1$ 的情況，不針對有 mask 的情況，而是直接以 logits 和 $z_indices$ 做 loss function，兩者在訓練 50 個 epoch 後差異很大，會在後續 discussion 時做討論，上述說到的兩種 loss function，第一種採用論文內部寫法的是於程式碼內 42 到 53 行，會先求出 logits 和 $z_indices$ ，然後要帶入 mask (程式第 50 行)，根據 batch_size 去產生同樣大小的 mask tensor，做後續只針對 mask 的比較 (第 51 行)，但是可能會存在沒有 mask 的部分因此第 54 行為避免出現沒有 loss 可以計算的情形而直接略過，接著再算期望值就可以了。而另一種方式以 cross_entropy 的方式，在程式碼第 46 行註解掉的部分，直接計算 $z_indices$ (vqgan encoder 結果) 和 logits (transformer 計算後的結果) 的 loss。

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

```

108 ##TODO3 step1-1: define one iteration decoding
109 ## masked_token Y_M*(t) current masked token from mask_bc
110 @torch.no_grad()
111 def inpainting(self, z_indices, org_mask_token, masked_token, current_iteration, total_iteration, gamma_func): # iterative decoding
112     # current masked_token (mask_bc)
113
114     masked_indices = self.mask_token_id * torch.ones_like(z_indices, device=z_indices.device)
115     a_indices = (-masked_token) * z_indices + masked_token * masked_indices
116
117     #raise Exception('TODO3 step1-1')
118     logits = self.transformer(a_indices) # get current latent predict
119
120     #Apply softmax to convert logits into a probability distribution across the last dimension.
121     # print("logits shape:", logits.shape)
122     logits = F.softmax(logits, dim=-1) # convert to distribution
123
124     #FIND MAX probability for each token value
125     z_indices_predict_prob, z_indices_predict = torch.max(logits, dim=-1) # get maximum loc as pred
126
127     #
128     ratio = current_iteration / total_iteration
129     #predicted probabilities add temperature annealing gumbel noise as confidence
130     g = torch.empty_like(z_indices_predict_prob).uniform_(0, 1)
131     g = -torch.log(-torch.log(g))
132     #g = 1
133     temperature = self.choice_temperature * (1 - ratio)
134     confidence = z_indices_predict_prob + temperature * g
135
136     #hint: If mask is False, the probability should be set to infinity, so that the tokens are not affected by the transformer's prediction
137     masked_confidence = confidence.masked_fill((~masked_token), float('inf')) # set ~mask as inf
138     #sort the confidence for the rank
139     sorted_confidence, sorted_indices = torch.sort(masked_confidence, dim=-1) # sort them
140     #define how much the iteration remain predicted tokens by mask scheduling
141
142     mask_cnt = int(torch.sum(org_mask_token)) # get masked count
143
144     gamma = self.gamma_func(gamma_func)
145     total_mask = int(gamma(ratio) * mask_cnt) # 根據 ratio 逐漸增加
146     new_mask = torch.zeros_like(org_mask_token)
147     new_mask.scatter_(1, sorted_indices[:, :total_mask], 1) # new mask
148
149     #At the end of the decoding process, add back the original token values that were not masked to the predicted tokens
150     original_token = z_indices.clone()
151     next_token = torch.where(new_mask, z_indices_predict, original_token)
152     #print(torch.count_nonzero(masked_token), torch.count_nonzero(new_mask), mask_cnt)
153
154     return next_token, new_mask, z_indices

```

此段程式為講解在 inpainting 時，每一個 iterative decoding 的動作，輸入為：

- `z_indices`：上一個 iteration 的預測值
- `org_mask_token`：原先要修復的 mask 圖(於 16x16 的 latent space 中)
- `masked_token`：上一個 iteration 的 mask 圖(於 16x16 的 latent space 中)
- `current_iteration`：這一次的 iteration，主要計算 mask ratio
- `total_iteration`：總共的 iteration 次數，主要計算 mask ratio
- `gamma_func`：所挑選的 mask scheduling function

程式 114-117 行做的跟 forward 時差不多的動作，先丟給 transformer 做預測得到 logits，接著再將 logits 轉成機率分布(程式 122 行)，再對這些分布做排序，排出信心值由高至低的排列(程式 125 行)，在 130-134 時則是會針對排序好的信心值加些額外的 gumbel noise，這一部分是有助於加速 inpainting 的結果，在 137 行時會針對沒有 mask 的位置改成 inf，這樣在挑選時都會一直挑選到，保證會一直更新現有需要修正的 mask，接著由原圖(`org_mask_token`)去計算所需的 mask 數量(第 142 行)，這樣做是因為如果都以上一張圖為例的話，可能會導致一個 token 中 mask 的數量越來越少，因此找一個定值(原圖)去乘上一個會隨著 iteration 而變動的 ratio(第 145、146 行)去計算總共需要 mask 的數量，最後設產出新的 mask 和針對該 mask 產出新的 logits 給下一個 iteration(當作下一個 iteration 的 `z_indices`，如程式碼 147 到 159 行)。

```

40  ##TODO3 step1-1: total iteration decoding
41  #mask_b: iteration decoding initial mask, where mask_b is true means mask
42  def inpainting(self, image, mask_b, i): #MakGIT inference
43      maska = torch.zeros(self.total_iter+1, 3, 16, 16) #save all iterations of masks in latent domain
44      imga = torch.zeros(self.total_iter+2, 3, 64, 64) #save all iterations of decoded images
45      mean = torch.tensor([0.4868, 0.4341, 0.3844], device=self.device).view(3, 1, 1)
46      std = torch.tensor([0.2628, 0.2527, 0.2543], device=self.device).view(3, 1, 1)
47      ori = (image[0]*std)+mean
48      imga[0]=ori #mask the first image be the ground truth of masked image
49
50      self.model.eval()
51      with torch.no_grad():
52          z_indices = None #z_indices: masked tokens (b,16*16)
53          mask_num = mask_b.sum() #total number of mask token
54          z_indices_predict = z_indices
55          mask_bc = mask_b
56          mask_b = mask_b.to(device=self.device)
57          mask_bc = mask_bc.to(device=self.device)
58          # mask_b -> masked latent space
59          # image -> masked image
60          #raise Exception('TODO3 step1-1')
61          _, z_indices = self.model.encode_to_z(image)
62          ratio = 0
63          #iterative decoding for loop design
64          #Hint: it's better to save original mask and the updated mask by scheduling separately
65          for step in range(self.total_iter+1): # ratio will be 1
66              if step == self.sweet_spot:
67                  break
68              ratio = step / self.total_iter
69
70              # mask_bc Y_M^t) masked token
71              # current
72              #
73              # get
74              z_indices_predict, mask_bc, z_indices = self.model.inpainting(z_indices, mask_b, mask_bc, step, self.total_iter, self.mask_func)
75              # mask_bc -> last masked latent space
76              # mask_b -> original masked latent space
77              # z_indices -> last predict token
78              z_indices = z_indices_predict # update last
79              #static method you can modify or not, make sure your visualization results are correct
80              mask_i = mask_bc.view(1, 16, 16)
81              mask_image = torch.ones(3, 16, 16)
82              indices = torch.nonzero(mask_i, as_tuple=False) #label mask true
83              mask_image[:, indices[:, 1], indices[:, 2]] = 0 #3,16,16
84              maska[step]=mask_image
85              shape=(1,16,16,256)
86              z_q = self.model.vqgan.codebook.embedding(z_indices_predict).view(shape)
87              z_q = z_q.permute(0, 3, 1, 2)
88              decoded_img = self.model.vqgan.decode(z_q)
89              dec_img_ori = (decoded_img[0]*std)+mean
90              imga[step+1]=dec_img_ori #get decoded image

```

在外層的 inpainting 時是針對整個 iteration 去呼叫內部的 inpainting 做每次的 iterative decoding，唯一有修改的地方是 第 61 行會先產出一開始的 `z_indices`，在進到 iterative decoding 去做一次的 iteration，拿到新的預測值(`z_indices_predict`) 和新的 mask token (`mask_bc`)，接著在到下一個 iteration 去做(程式第 74 行)，在 78 行時則是針對 `z_indices` 去做更新，因為每一次都是利用上一次的結果，mask token 也是使用上一次的結果，程式的後半段則是保證了在非 mask 區域是由 vqgan 所產生的預測(程式 86 行)。

3. Discussion

A. Anything you want to share

在設計 Loss function 時候有提到，使用兩種不同的 Loss function，分別是針對 mask 的情況下去訓練 transformer 和直接對所有值(無針對 mask 的情況下) logits、`z_indices` 去做 cross entropy，發現針對所有值去做 cross

entropy 的效果遠大於針對 mask 情況下去做 loss function 和 backward。

在訓練這兩個 loss function 過程中發現收斂速度都很快，收斂的點 loss 也很大，但是兩者經過 inpainting 的 inference 後結果卻大相逕庭。

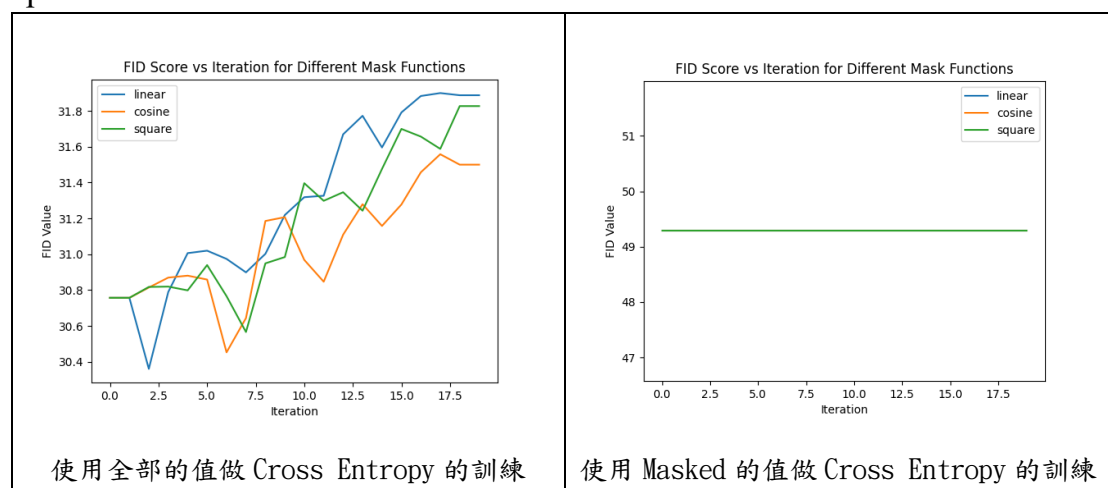
在 training 時的 optimizer 和 scheduler 都是以 AdamW，初始 lr 為 0.0001，scheduler 用 ReduceLROnPlateau，factor 為 0.1，最小為 1e-6。

```

96 def configure_optimizers(self):
97     optimizer = torch.optim.AdamW(self.model.transformer.parameters(), lr=0.0001, betas=(0.9, 0.96), weight_decay=4.5e-2)
98     scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, factor=0.1, min_lr=1e-6, patience=2)
99     # scheduler = None
100     return optimizer, scheduler

```

在結果上為下面兩張圖(所用的種子碼都是一樣的)，分別都訓練 50 個 epoch。



發現在 Masked 時會有 train 不起來的情況，並且 FID 在每個 iteration 都是 49 左右。

4. Experiment

A. Show iterative decoding (Prove your code implementation is correct)

在此都是以最佳 FID 的結果做表示，並且下方展示的圖都已第 42 號圖為例子。

Cosine

(Mask in latent domain)

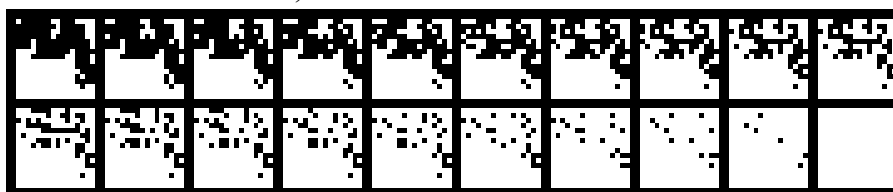


(Predicted image)



Linear

(Mask in latent domain)

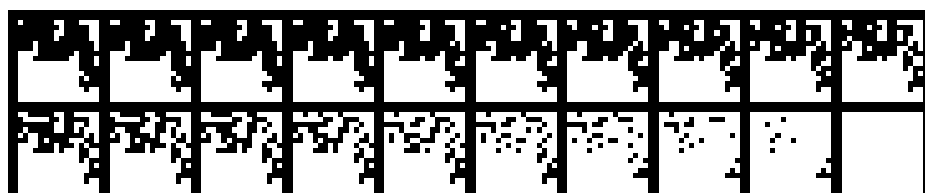


(Predicted image)



Square

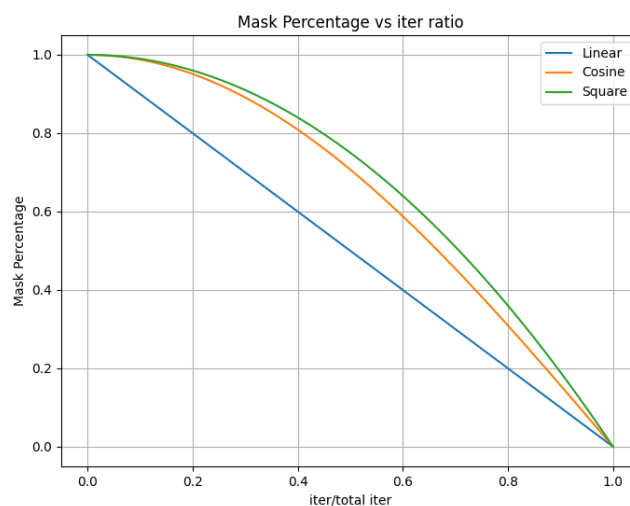
(Mask in latent domain)



(Predicted image)



由 linear 的圖中可以看到，mask 減少的範圍是定量的，在 square 和 cosine 的比較可以由下圖來看：



可以看到整體覆蓋比率(Mask Percentage)會隨著 iteration 次數增加而衰減至 0，而 Linear 算是衰減速度較快的，Square 在後期才會開始衰減，Cosine 則是介於兩者之間，由上方的圖中也可以看到 Square 在後期一次減少的 mask 數量較多，Linear 較平均等。

B. The Best FID Score



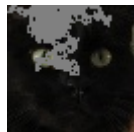
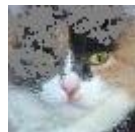
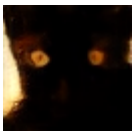

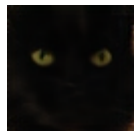
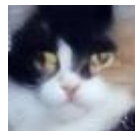




Screenshot(Best FID Score)



此挑選的結果為使用直接比對所有的值做 cross entropy 的 loss，和使

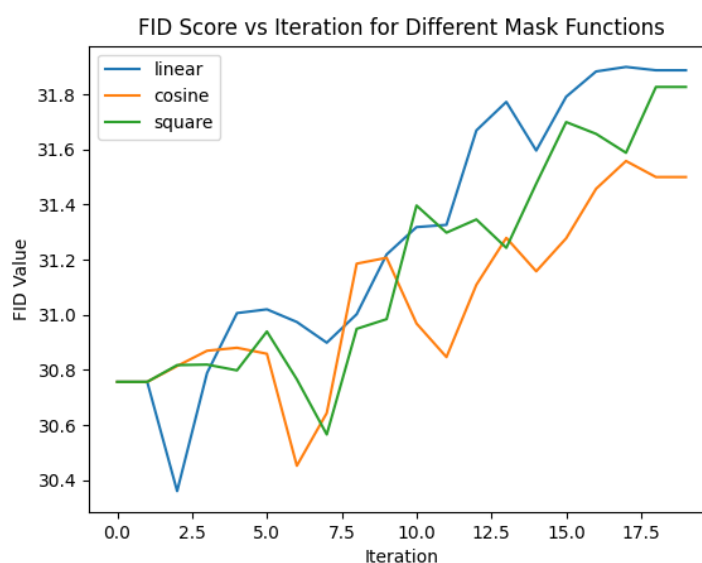
用 linear 作為 inpainting 的 mask function，此總共的 iteration 數為 19，在第 2 個 iteration 次得到的結果(sweet point = 2)。

Masked Images v.s MaskGIT Inpainting Results v.s Ground Truth

Linear iteration 2 type	Image 073	Image 086	Image 094	Image 410
Masked Images				
MaskGIT Inpainting Results				
Ground Truth				

The setting about training strategy, mask scheduling parameters

在 Discussion 中有提到使用了兩個不同的 Loss function 作為 training 的 strategy 比較，此外在訓練時設定的 mask scheduling gamma function 都是以 cosine 的方式去生成。在 inference 的 inpainting 時有使用三種不同的 gamma function (Cosine、Linear、Square)，如下圖所示，根據所有的 iteration 得到不同的 FID 結果



在 iteration 增加時表現的 FID 會越來越差，其中發現 linear 在前期的 mask function 表現最好，但隨著 ratio 的增加，cosine 的表現效果是最好的。