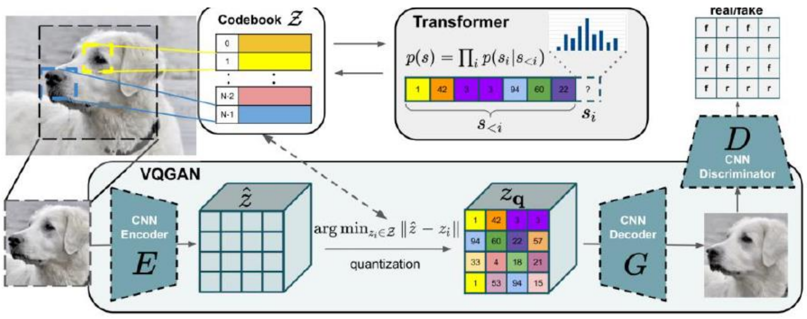
NYCU DL

Lab5 – MaskGIT for Image Inpainting

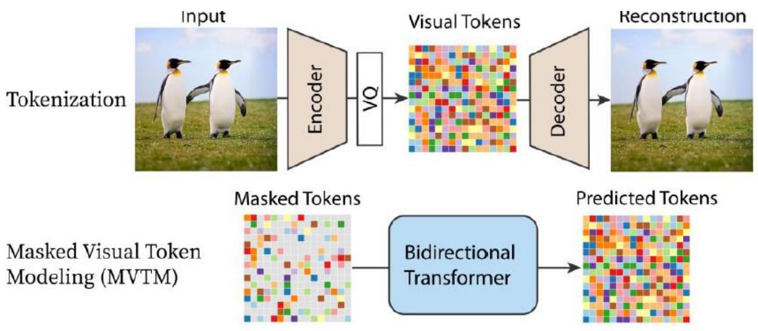
312554006 羅名志

1. Introduction of Lab 5

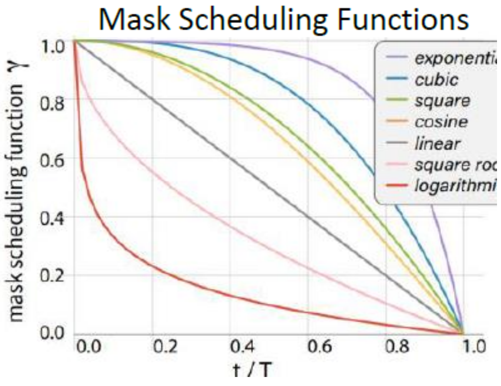
The lab focuses on implementing MaskGIT for the inpainting task. Specifically, during inference, given a masked image, the model should restore the missing information and return a clear picture. Moreover, the model used in this lab is VQGAN. Unlike VQVAE, VQGAN includes both a discriminator and a transformer, where the discriminator helps improve picture quality, and the transformer predicts the next possible token from the given embedding after passing through the CNN encoder, enhancing the restoration ability (as shown below).

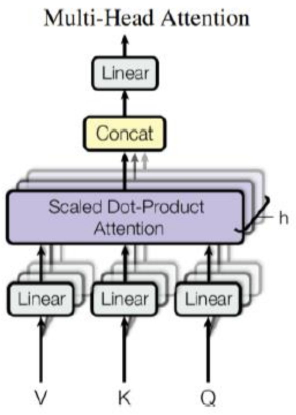


The lab is roughly divided into two parts. The first part is to train a transformer with strong predictive abilities. To achieve this, we randomly mask parts of images and train the transformers to restore the masked images to their original form. By doing this, the model gains the ability to restore unseen images.

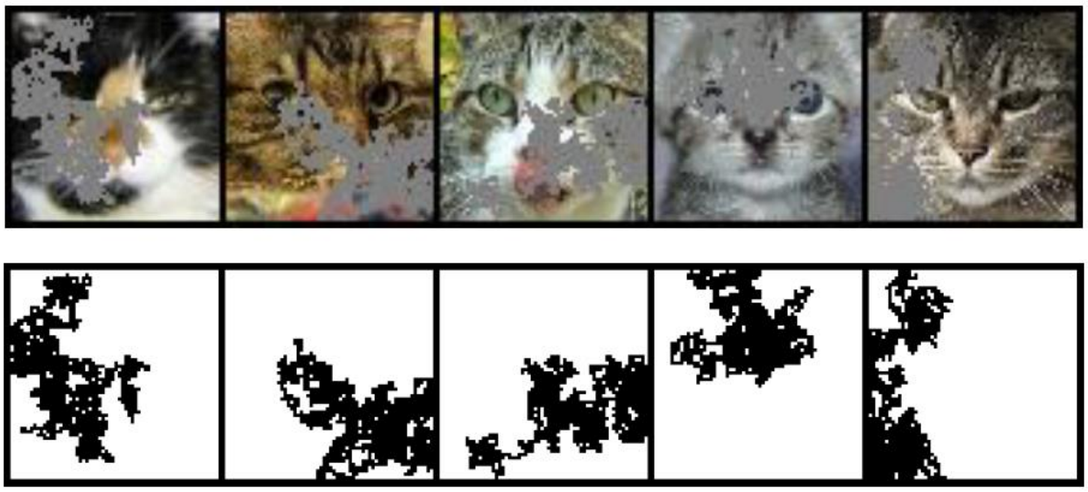


The second part involves constructing the inference environment. The model should be fed a masked image along with the masked area label to output the restored image. This includes strategies such as mask scheduling design, masking design, multi-head attention, etc.



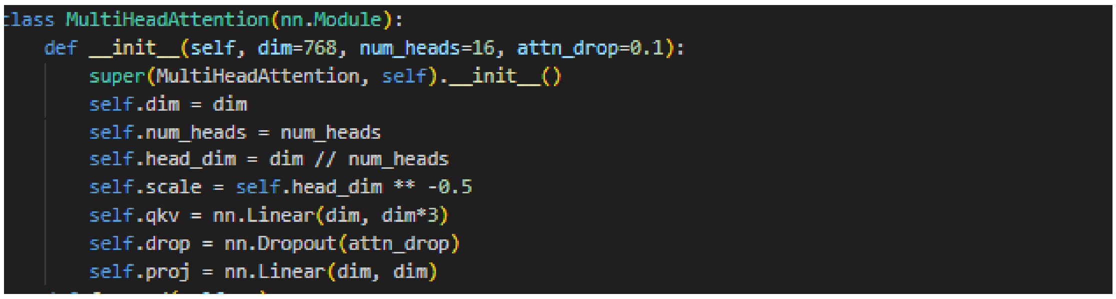


The dataset in this lab contains 12,000 cat face training images, 3,000 cat face validation images, and 747 masked images along with their corresponding ground truth masks.



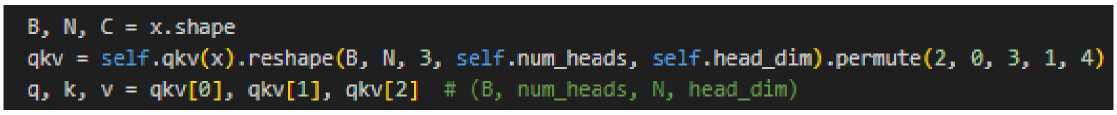
2. Implementation Details

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



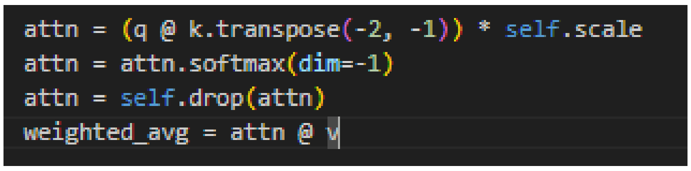
```python
class MultiHeadAttention(nn.Module):
    def __init__(self, dim=768, num_heads=16, attn_drop=0.1):
        super(MultiHeadAttention, self).__init__()
        self.dim = dim
        self.num_heads = num_heads
        self.head_dim = dim // num_heads
        self.scale = self.head_dim ** -0.5
        self.qkv = nn.Linear(dim, dim*3)
        self.drop = nn.Dropout(attn_drop)
        self.proj = nn.Linear(dim, dim)
```

- self.num_heads: the number of attention heads
- self.head_dim: the dimension of each head should equal to dim//num_heads
- self.scale: help to stabilize the gradients, equal to inverse square root of head_dim
- self.qkv: linear layer of Query, Key, Value
- self.drop: dropout function
- self.proj: project the output from each head to the input dimension



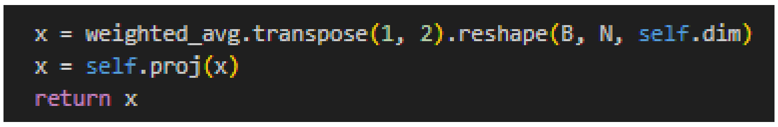
```python
B, N, C = x.shape
qkv = self.qkv(x).reshape(B, N, 3, self.num_heads, self.head_dim).permute(2, 0, 3, 1, 4)
q, k, v = qkv[0], qkv[1], qkv[2]  # (B, num_heads, N, head_dim)
```

In the forward process, the output of qkv function will be dim*3, where 3 stands for query, key, value. Also, reshape it to follow the multi-head setting, shaped as (head_dim*num_heads)*3. The permute function here is just for easier calculation.
*now the size of q, k, v is [batch size, num_heads, num_image_tokens, head_dim]



```python
attn = (q @ k.transpose(-2, -1)) * self.scale
attn = attn.softmax(dim=-1)
attn = self.drop(attn)
weighted_avg = attn @ v
```

Calculating attention score: First calculate the dot product between q and k (here the transpose function aims for aligning dimensions). Following the scaling and softmax function, we get the attention score. The last step will be calculating the attention score with each value tokens. Here we have each multi-head attention outputs.

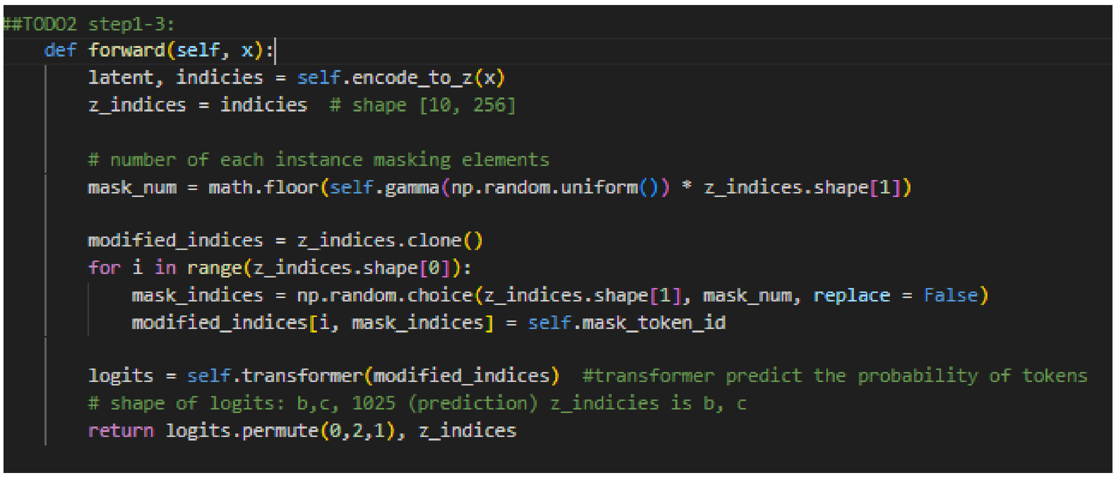


```python
x = weighted_avg.transpose(1, 2).reshape(B, N, self.dim)
x = self.proj(x)
return x
```

Finally, we reshape the attention outputs and reshape to original dimension (B, N, dim). Also, it is important to project the concatenated outputs from the different attention heads, considering the information from different heads.

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

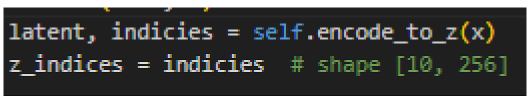
Overview of the forward function:



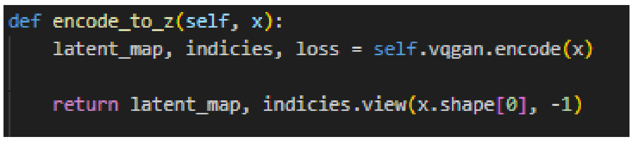
```python
##TODO2 step1-3:
    def forward(self, x):
        latent, indicies = self.encode_to_z(x)
        z_indices = indicies  # shape [10, 256]

        # number of each instance masking elements
        mask_num = math.floor(self.gamma(np.random.uniform()) * z_indices.shape[1])

        modified_indices = z_indices.clone()
        for i in range(z_indices.shape[0]):
            mask_indices = np.random.choice(z_indices.shape[1], mask_num, replace = False)
            modified_indices[i, mask_indices] = self.mask_token_id

        logits = self.transformer(modified_indices)  #transformer predict the probability of tokens
        # shape of logits: b,c, 1025 (prediction) z_indicies is b, c
        return logits.permute(0,2,1), z_indices
```

The following is the explanation of the forward function and methods I use:

(1) Get the indices from encode_to_z function

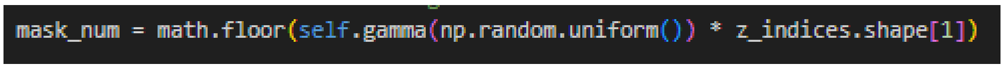


```python
latent, indicies = self.encode_to_z(x)
z_indices = indicies  # shape [10, 256]
```



```python
def encode_to_z(self, x):
    latent_map, indicies, loss = self.vqgan.encode(x)

    return latent_map, indicies.view(x.shape[0], -1)
```

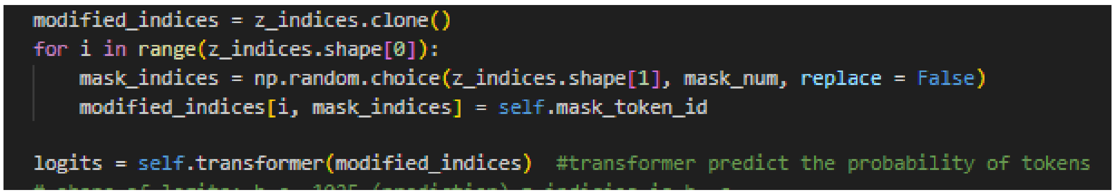
(2) Decide the number of random masks



```python
mask_num = math.floor(self.gamma(np.random.uniform()) * z_indices.shape[1])
```

Where the gamma function returns the linear one, so the masking methods I use here is a random number from 0 to 1 and multiply the total number of indices.

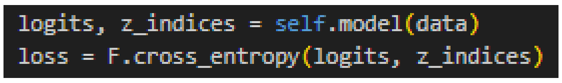
(3) copy the indices and mask by each batch instance. The for loop is to randomly choose the indices to be masked. Specifically, the mask indices will be given a value mask_token_id.



```
modified_indices = z_indices.clone()
for i in range(z_indices.shape[0]):
    mask_indices = np.random.choice(z_indices.shape[1], mask_num, replace = False)
    modified_indices[i, mask_indices] = self.mask_token_id

logits = self.transformer(modified_indices)  #transformer predict the probability of tokens
```

After modified the indices, the matrix will be fed into the transformer and get the prediction output, then the forward function returns them to calculate the loss.

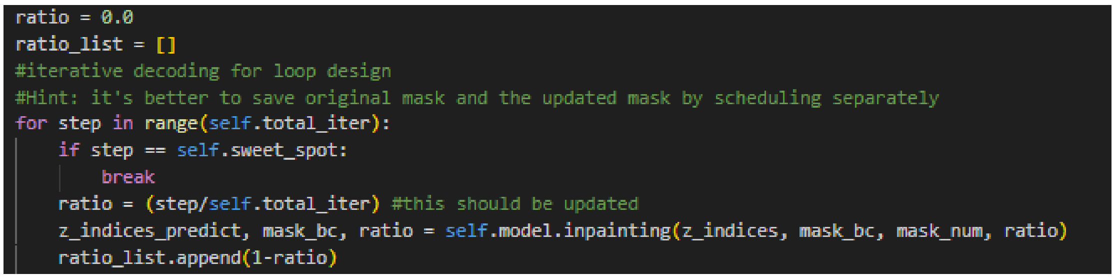
(4) The loss is calculated by cross-entropy loss:



```
logits, z_indices = self.model(data)
loss = F.cross_entropy(logits, z_indices)
```

C. The details of your inference for inpainting task

The overview of inference function”

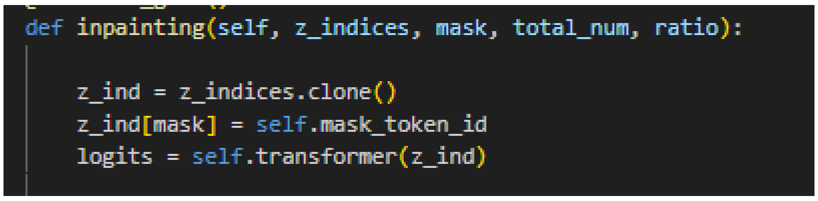


```
ratio = 0.0
ratio_list = []
#iterative decoding for loop design
#Hint: it's better to save original mask and the updated mask by scheduling separately
for step in range(self.total_iter):
    if step == self.sweet_spot:
        break
    ratio = (step/self.total_iter) #this should be updated
    z_indices_predict, mask_bc, ratio = self.model.inpainting(z_indices, mask_bc, mask_num, ratio)
    ratio_list.append(1-ratio)
```

In each step the ratio will be the current step / total iteration, and be fed into the inpainting function to decode the image. For demo simplicity, here I also record the ratio value in each step. (Will shown in the experiment)

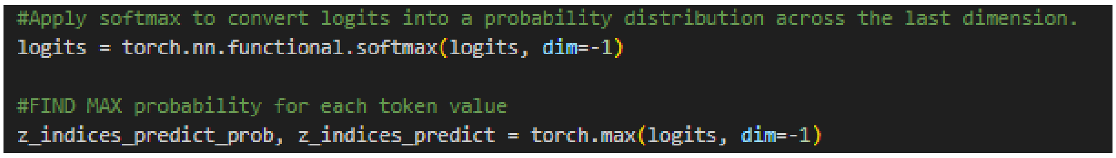
For model.inpainting:

(1) First get the image indices from encoder and mask those indcies based on given ground truth mask. And fed into the transformer we trained in step 2 to get the prediction outputs.



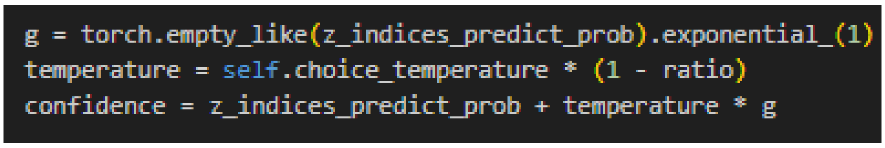
```
def inpainting(self, z_indices, mask, total_num, ratio):

    z_ind = z_indices.clone()
    z_ind[mask] = self.mask_token_id
    logits = self.transformer(z_ind)
```

(2) Calculate the logits for the output prediction from transformer, and use the max function to get the most possible predictions and corresponding probability.



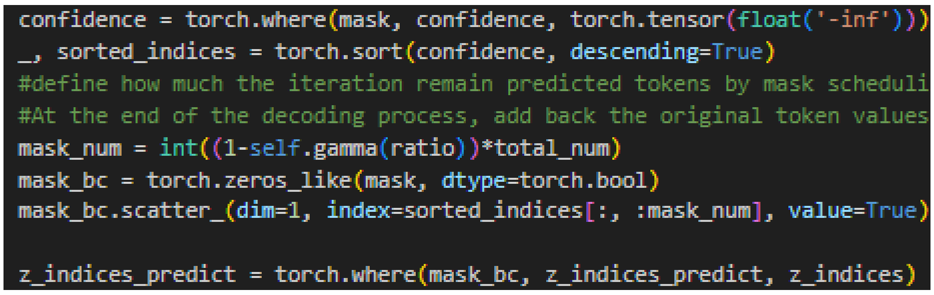
```python
#Apply softmax to convert logits into a probability distribution across the last dimension.
logits = torch.nn.functional.softmax(logits, dim=-1)

#FIND MAX probability for each token value
z_indices_predict_prob, z_indices_predict = torch.max(logits, dim=-1)
```

(3) Calculate the confidence with temperature and random noise. The purpose here is to learn a more varied prediction probability in early stages.



```python
g = torch.empty_like(z_indices_predict_prob).exponential_(1)
temperature = self.choice_temperature * (1 - ratio)
confidence = z_indices_predict_prob + temperature * g
```

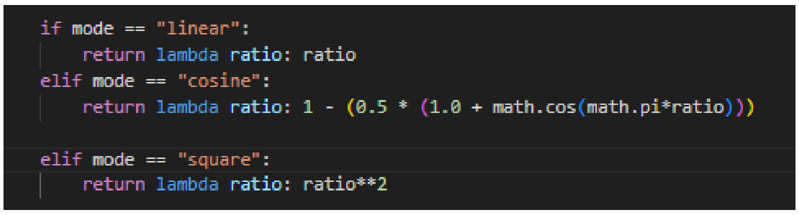
(4) Finally, make those unmasked indices' probability be -inf, meaning that it won't be selected to restore (or say fed into the transformer). And sort the indices for us to select the highest indices to replace.



```python
confidence = torch.where(mask, confidence, torch.tensor(float('-inf')))
_, sorted_indices = torch.sort(confidence, descending=True)
#define how much the iteration remain predicted tokens by mask scheduli
#At the end of the decoding process, add back the original token values
mask_num = int((1-self.gamma(ratio))*total_num)
mask_bc = torch.zeros_like(mask, dtype=torch.bool)
mask_bc.scatter_(dim=1, index=sorted_indices[:, :mask_num], value=True)

z_indices_predict = torch.where(mask_bc, z_indices_predict, z_indices)
```

Here, the mask_bc will be set to true if the indices is the top mask_num highest probability. Where the mask_num is calculated by (1-gamma(ratio)) * total number in the indices.
*As explained in the previous part, the ratio = current step / total step

(5) The gamma function is defined as below:



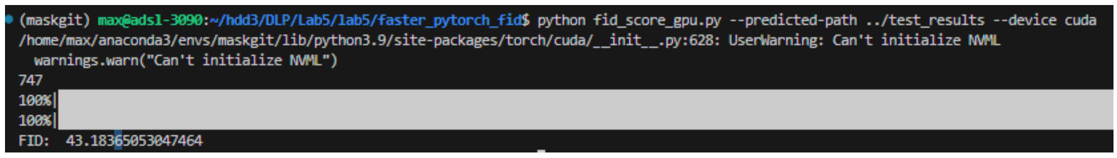
```python
if mode == "linear":
    return lambda ratio: ratio
elif mode == "cosine":
    return lambda ratio: 1 - (0.5 * (1.0 + math.cos(math.pi*ratio)))

elif mode == "square":
    return lambda ratio: ratio**2
```

3. Experimental results

// Since the model can’t generate the same result due to the random components, the result in part A and part B will be slightly different.

A. The best testing fid

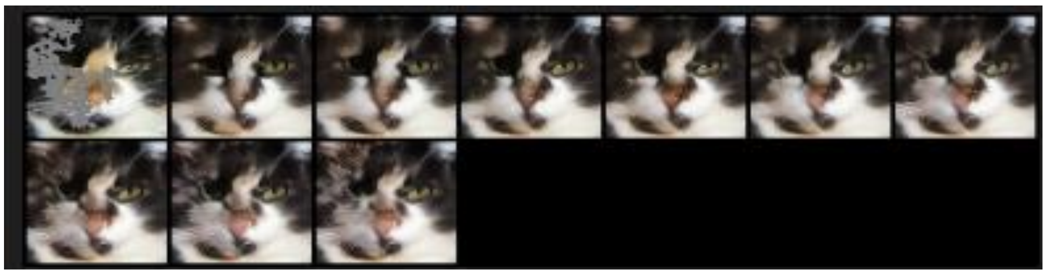
(1) Screenshot



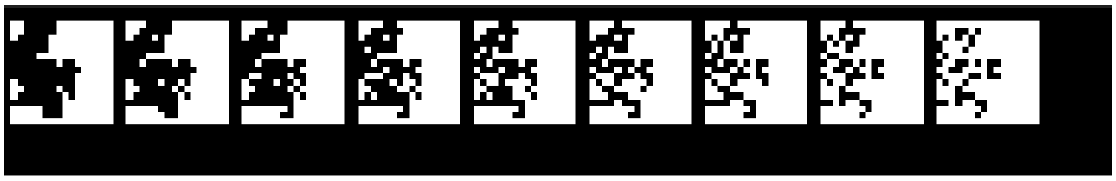
```
(maskgit) max@adsl-3090:~/hdd3/DLP/Lab5/lab5/faster_pytorch_fid$ python fid_score_gpu.py --predicted-path ../test_results --device cuda
/home/max/anaconda3/envs/maskgit/lib/python3.9/site-packages/torch/cuda/__init__.py:628: UserWarning: Can't initialize NVML
  warnings.warn("Can't initialize NVML")
747
100%|
100%|
FID:  43.18365053047464
```

(2) Predicted image, Mask in latent domain with mask scheduling

a. Predicted image



b. Mask in latent domain



(3) The setting about training strategy, mask scheduling parameters…

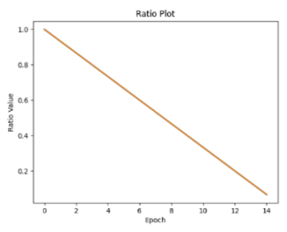
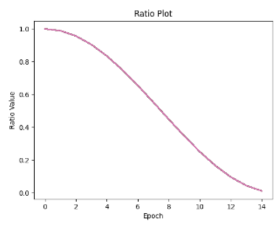
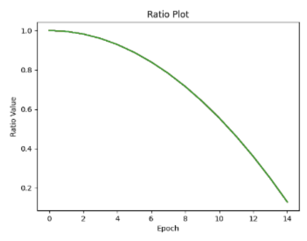
a. Training strategy: try to use different iterations and find the best sweet spot with the highest FID score

b. mask scheduling parameters:

total iterations = 13, sweet spot = 9, masking function = cosine

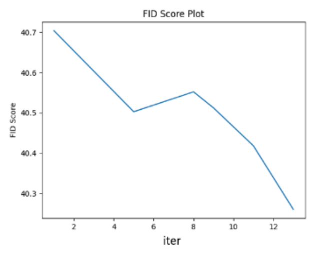
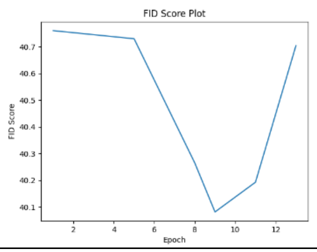
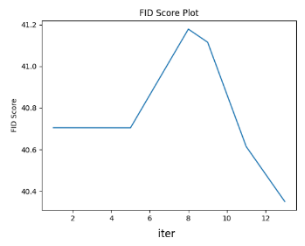
B. Comparison figures with different mask scheduling parameters setting

(1) Ratio Plot

	Linear	Cosine	Square
Ratio Plot			

Both cosine and square masking schedules will exhibit increased decay in the later iterations and a more gradual decrease in the early iterations.

(2) FID Score Plot on each sweet spot

	Linear	Cosine	Square
FID Plot			
Best FID	40.12	40.04	40.23

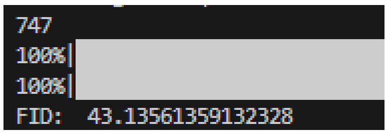
// Here the total iterations is 13, and x axis stands for different sweet spots.
It is evident that using cosine masking yields the best result with an FID score of 40.04. Additionally, all masking mechanisms exhibit distinct FID curve patterns. Cosine masking achieves the best result at the middle iteration, specifically iteration 9. Both linear and square masking reach their optimal results when the last iteration is selected as the sweet spot. However, square masking displays an unusual pattern. Ideally, the curve should show the best FID score with increasing iterations, as the model refines the restored indices. Nonetheless, the curve spikes in the middle and then declines, likely because the trained transformer fails to restore the correct indices at that point.

(3) FID of different total_iterations settings

// Under same setting:

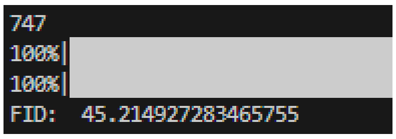
- No sweet spot, get the final inference output.
- Same transformer weight
- Same masking function: cosine

a. 5 epochs



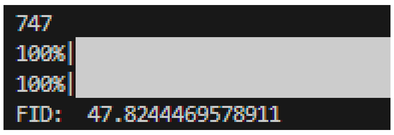
```
747
100%|
100%|
FID:  43.13561359132328
```

b. 8 epochs



```
747
100%|
100%|
FID:  45.214927283465755
```

c. 13 epochs



```
747
100%|
100%|
FID:  47.8244469578911
```

The results are quite impressive; increasing the total number of epochs does not necessarily improve the final inference results. Instead, it is crucial to identify the sweet spot for each setting. However, with more iterations, the grids become finer, allowing for the identification of a better sweet spot among the iterations.

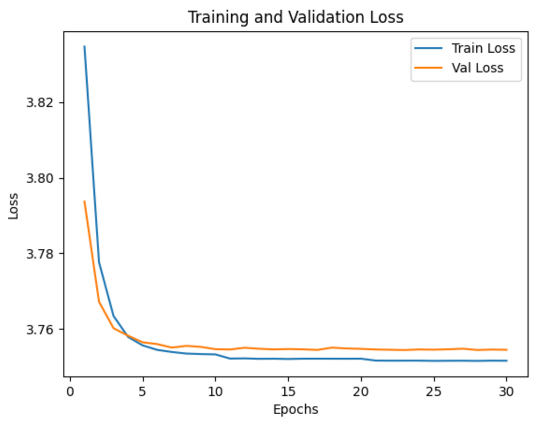
4. Discussion

(1)

The inpainting inference results were not as expected, as the FID score consistently fluctuated with each iteration across all masking mechanisms. One possible reason for this is that the transformer in stage 2 may not have been trained adequately, resulting in the generation of low-quality indices.

(2)

Upon examining the loss function in stage 2, it was observed that after 2 epochs, the training and validation losses plateaued and did not decrease further, as shown below. This suggests that the model may not have been trained successfully, leading to poor results in the final inference stage.



3. The cosine masking function likely yields the best result because it decreases more gradually in the early stages, allowing the model to better adapt to the masked indices. However, while the square function also gradually decreases in the early stages, it does not produce better results. This could be due to various factors such as random noise components in the model and inadequate training of the transformer.

4. To identify potential issues with the transformer, exploring different mapping mechanisms might resolve the problem but I am still trying ...

5. Reference

Reference
1. Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In NeurIPS, 2017. https://arxiv.org/pdf/2202.04200.pdf
2. A. van den Oord, O. Vinyals, et al., “Neural discrete representation learning,” in Advances in Neural Information Processing Systems, pp. 6306–6315, 2017. https://arxiv.org/abs/1711.00937
3. Esser, P., Rombach, R., and Ommer, B.: Taming Transformers for HighResolution Image Synthesis. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 12873–12883 (2021) https://arxiv.org/abs/2012.09841
4. Huiwen Chang, Han Zhang, Lu Jiang, Ce Liu, and William T. Freeman. Maskgit: Masked generative image transformer. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, June 2022. https://arxiv.org/abs/2202.04200
5. TA’s spec and DL slides
6. ChatGPT: https://chat.openai.com