Lab3: MaskGIT for Image Inpainting

Shyang-En Weng 翁祥恩 413551036

March 30, 2025

1 Introduction

Image inpainting is a fundamental computer vision task that involves restoring missing or corrupted regions of an image in a visually coherent manner. In this lab, we explore MaskGIT [1], a transformer-based generative model for high-quality image synthesis and inpainting. We will implement and analyze MaskGIT, assessing its effectiveness in reconstructing missing image regions. Additionally, I will examine my implementation and training process. The insights gained from this study will enhance our understanding of generative models and their applications in image restoration. The highlighted points are shown below:

- MaskGIT has successfully implemented, including the Multi-head Self-Attention, training of bidirectional transformer, and iterative decoding. The performance of my implementation reached FID 35.72 and also showed the visualization of the inpainting.
- The comparisons of the sweet points and mask functions in iterative decoding are analyzed and discussed.

2 Implementation Details

2.1 Multi-head Self-Attention

Multi-head self-attention [3] is a fundamental mechanism in transformer-based models, enabling them to capture intricate dependencies across different parts of an input sequence, as shown in Listing 1. Given an input, we first project it into three distinct representations: Query (Q), Key (K), and Value (V), with the same shape (batch_size, num_image_tokens, dim). These representations are then reshaped into (batch_size, num_heads, num_image_tokens, head_dim), allowing each attention head to capture different aspects of the sequence independently. Next, we compute the attention scores by taking the scaled dot product of Q and K, followed by a softmax operation to obtain the attention weights, which determine the relative importance of each token, then applied to

V to produce the attention output: These attention weights are :

$$\text{Attention} = \text{softmax}(\frac{QK^T}{\sqrt{\text{head\_dim}}}) \cdot V. \tag{1}$$

where head_dim = dim/num_heads.

Finally, The outputs from all heads are concatenated and projected back to the original embedding dimension, yielding the final multi-head self-attention output.

Listing 1: Implementation of Multi-Head Self-Attention.

```
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.dim = dim
        self.head_dim = dim // num_heads
        self.scale = self.head_dim ** -0.5

        self.qkv = nn.Linear(dim, dim*3)
        self.linear = nn.Linear(dim, dim)
        self.attn_drop = nn.Dropout(attn_drop)

    def forward(self, x):
        qkv = self.qkv(x)
            # shape = [batch_size, num_image_tokens, 3*dim]
        qkv = qkv.reshape(
            x.shape[0], x.shape[1], 3,
            self.num_heads, self.head_dim)
            # shape = [batch_size, num_image_tokens, 3, num_heads, head_dim]
        qkv = qkv.permute(2, 0, 3, 1, 4)
        q, k, v = qkv[0], qkv[1], qkv[2]
            # shape = [batch_size, num_heads, num_image_tokens, head_dim]

        attn = self.scale * (q @ k.transpose(-2, -1))
            # shape = [batch_size, num_heads,
                num_image_tokens, num_image_tokens]
        attn = torch.softmax(attn, dim=-1)
        attn = self.attn_drop(attn)

        attn = attn @ v # shape = [
            batch_size, num_heads, num_image_tokens, head_dim]
        attn = attn.permute(1, 2, 0, 3).reshape(
            x.shape[0], x.shape[1], self.dim)
                # shape = [batch_size, num_image_tokens, dim]
        return self.linear(attn)
```

2.2 Stage 2 Training

In Stage 2 training, we utilize a Bidirectional Transformer to model the latent representations obtained from VQGAN [2]. This process involves learning the dependencies between latent tokens, allowing the transformer to capture long-range correlations and generate coherent outputs. Masked Visual Token Modeling (MVTM) employs a mask token to facilitate masked token prediction, enabling the model to infer missing information and enhance the learned latent representations. This approach improves the robustness and expressiveness of the latent space, ultimately leading to higher-quality outputs.

First of all, the image is mapped by VQGAN encoder to get the latent representation. Next, we apply a random masking controled by random ratio, and get a masked latent. Then, we input masked latent to bidirectional transformer for restoration. Finally, the cross-entropy loss is applied to compare the ground truth (the tokens before masking) and the predicted tokens. The process is shown in Listings 2 and 3.

Listing 2: Implementation of MaskGIT.

```
class MaskGIT(nn.Module):
def __init__(self, configs):
    super().__init__()
    self.vqgan = self.load_vqgan(configs['VQ_Configs'])

    self.num_image_tokens = configs['num_image_tokens']
    self.mask_token_id = configs['num_codebook_vectors']
    self.choice_temperature = configs['choice_temperature']
    self.gamma = self.gamma_func(configs['gamma_type'])
    self.transformer = BidirectionalTransformer(
        configs['Transformer_param'])

def load_transformer_checkpoint(self, load_ckpt_path):
    self.transformer.load_state_dict(torch.load(load_ckpt_path))

@staticmethod
def load_vqgan(configs):
    cfg = yaml.safe_load(
        open(configs['VQ_config_path'],
        'r'))
    model = VQGAN(cfg['model_param'])
    model.load_state_dict(torch.load(configs['VQ_CKPT_path']), strict=True)
    model = model.eval()
    return model

##TODO2 step1-1: input x fed to vqgan encoder to get the latent and zq
@torch.no_grad()
def encode_to_z(self, x):
```

```
        codebook_mapping, codebook_indices, q_loss = self.vqgan.encode(x)
        return codebook_mapping, codebook_indices

##TODO2 step1-2:
    def gamma_func(self, mode="cosine"):
        if mode == "linear":
            def masking_ratio(ratio):
                return 1 - ratio
            return masking_ratio
        elif mode == "cosine":
            def masking_ratio(ratio):
                return math.cos(math.pi * ratio * 0.5)
            return masking_ratio
        elif mode == "square":
            def masking_ratio(ratio):
                return 1-ratio ** 2
            return masking_ratio
        else:
            raise NotImplementedError

##TODO2 step1-3:
    def forward(self, x, ratio):

        z, z_indices = self.encode_to_z(x)
        z_indices = z_indices.view(-1, self.num_image_tokens)

        # Mask the tokens
        z_indices_input = self.apply_masking(ratio, z_indices)

        logits = self.transformer(z_indices_input)

        return logits, z_indices

    def apply_masking(self, ratio, z_indices):
        mask_token = torch.bernoulli(torch.ones_like(z_indices, ) * ratio)
        mask_token_id = torch.tensor(self.mask_token_id).to(z_indices.device)
        z_indices = torch.where(mask_token == 1, mask_token_id, z_indices)
        return z_indices
```

Listing 3: Stage 2 Training.

```
    def train_one_epoch(self, train_loader):
        self.model.train()
        total_loss = 0
        loader = tqdm(train_loader)
        for i, x in enumerate(loader):
```

```
            x = x.to(args.device)
            self.optim.zero_grad()
            ratio = np.random.rand()
            with autocast(device_type=args.device, enabled=self.args.amp):
                logits, z_indices = self.model(x, ratio)
                loss = self.loss(logits.permute(0, 2, 1), z_indices)
            self.scalar.scale(loss).backward()

            self.scalar.step(self.optim)

            self.scalar.update()

            total_loss += loss.item()

            loader.set_description(f"Training Loss: {total_loss / (i + 1)}")
        self.scheduler.step()
        return total_loss / (i + 1)
```

2.3 Iterative Decoding (Inpainting)

In the inference phase, MaskGIT employs an iterative decoding process to reconstruct masked tokens progressively. Starting with an initial mask ratio, the model predicts all tokens simultaneously, retaining the most confident predictions while masking the remaining tokens for re-prediction in subsequent iterations, refining the image with each cycle. Additionally, MaskGIT incorporates temperature-annealed Gumbel noise to adjust token confidence, and the mask ratio follows dynamic scheduling functions (linear, cosine, and square), which control the proportion of masked tokens and significantly impact the efficiency and quality of image generation. The implementation is shown in Listing 4.

Listing 4: Inference.

```
    def inpainting(self, z_indices, mask_b, ratio,
        mask_num, mask_func="cosine"):
        # raise Exception('TODO3 step1-1!')
        z_indices = torch.where(mask_b == 1,
            torch.tensor(self.mask_token_id).to(z_indices.device), z_indices)
        z_indices = z_indices.view(-1, self.num_image_tokens)
        logits = self.transformer(z_indices)
        #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)
```

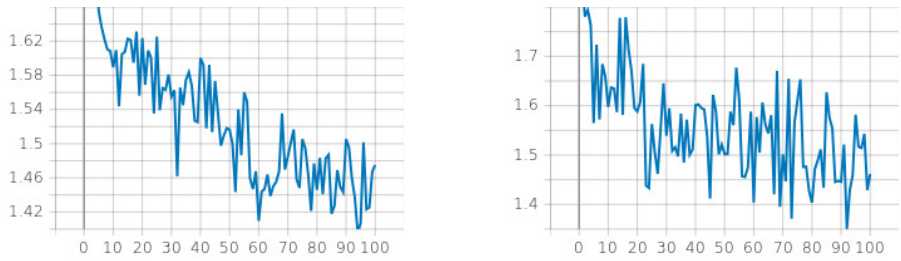
```
ratio = self.gamma_func(mask_func)(ratio)
#predicted probabilities add temperature annealing gumbel noise
# as confidence

# gumbel noise
g = -torch.log(-torch.log(torch.rand_like(z_indices_predict_prob) + 1e-9
temperature = self.choice_temperature * (1 - ratio)
confidence = z_indices_predict_prob + temperature * g

mask_num = (mask_num * ratio).long()
sorted_confidence, _ = torch.sort(confidence, dim=-1)
cut_off = sorted_confidence[:,mask_num].unsqueeze(-1)
mask_bc= (confidence < cut_off)
return z_indices_predict, mask_bc
```

3 Discussion

3.0.1 Training and inference details

For training, the epoch count is set to 100, using Adam optimization with a learning rate of 10^{-4}. The CosineAnnealingLR scheduler is utilized, and Automatic Mixed Precision (AMP) is implemented to accelerate the training process. The loss function is cross-entropy loss. The learning curve is shown in Figure 1.

For inference, the model uses the weights from epoch 92, with the sweet spot set to 2 and a total of 4 iterations. The cosine scheduling function is applied as the masking strategy to achieve the best FID score.



(a) Training Loss (b) Validation Loss

Figure 1: Training and validation loss curves.

3.1 Comparison of different scheduling functions

Table 1 presents the results of varying scheduling functions. We observe that the cosine and square functions yield comparable performance, whereas the linear

function performs significantly worse. These results highlight the importance of selecting an appropriate scheduling function for better image generation quality.

Scheduling function	FID
cosine	42.37
linear	55.26
square	42.28

Table 1: Comparison of different scheduling functions. The sweet spot is set to 3, and the total iteration is set to 10.

3.2 Comparison of different sweet spots and total iterations

The comparison is shown in Table 2. We observed that setting the sweet spot to 2 with a total of 4 iterations achieves the best FID score of 35.72. As the sweet spot increases, the FID score worsens, indicating that a lower sweet spot leads to better image quality. Similarly, increasing the total number of iterations to 10 generally results in higher FID scores, with a significant drop in performance when the sweet spot is set to 10. Additionally, choosing a sweet spot that is too large may cause the generated images to appear overly smooth, leading to a loss of fine details and texture, as shown in Figure 2. These results highlight the importance of choosing an optimal sweet spot and iteration count to balance image quality and inference efficiency.

Number of sweet spot	Total iteration	FID
2	4	35.72
3	4	38.80
4	4	41.13
3	10	42.37
5	10	41.56
10	10	53.78

Table 2: Comparison of different sweet spots. The scheduling function for the experiments is the cosine function.

4 Experiment Score

4.1 Iterative Decoding

The following experiments are conducted with the sweet spot set to 3 and the total number of iterations set to 10.

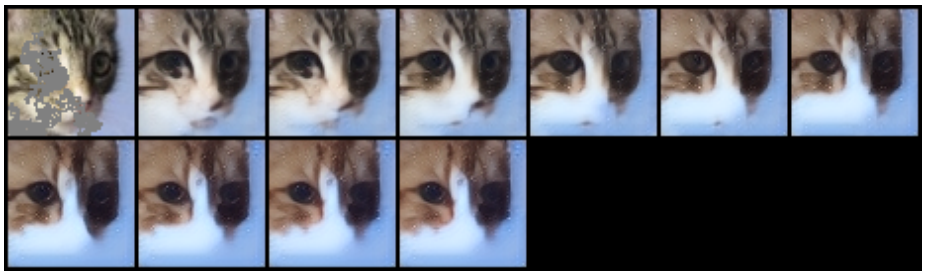


Figure 2: Predicted images of each iteration and each sweet spot.

4.1.1 Cosine

The predicted images and masks of iterative decoding using the cosine scheduling function are shown in Figures 3 and 4.

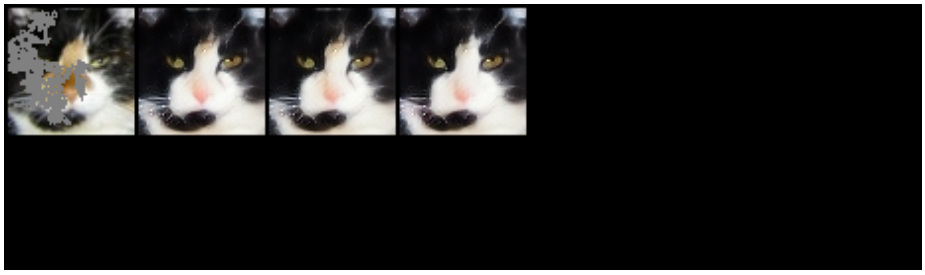


Figure 3: Predicted images of each iteration using the cosine scheduling function.

Figure 4: Masks of each iteration using the cosine scheduling function.

4.1.2 Linear

The predicted images and masks of iterative decoding using the linear scheduling function are shown in Figures 5 and 6.

4.1.3 Square

The predicted images and masks of iterative decoding using the square scheduling function are shown in Figures 7 and 8.

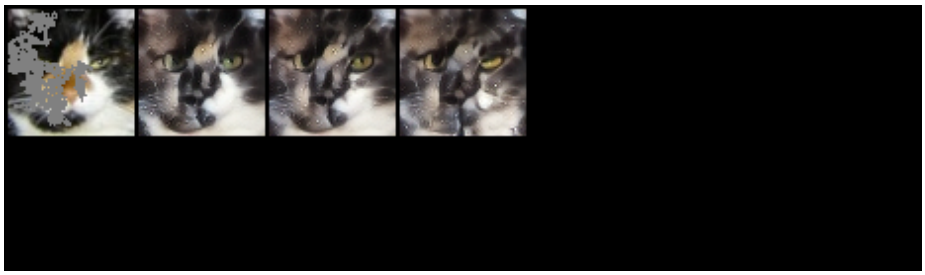


Figure 5: Predicted images of each iteration using the linear scheduling function.

Figure 6: Masks of each iteration using the cosine scheduling function.

4.2 The best FID score

The evaluation screenshot of the best FID score is presented in Figure 9, while Figure 10 compares the masked and restored images. Although the inpainting results are not perfect, the model successfully learns the characteristics of a cat and reconstructs the missing parts, demonstrating the effectiveness of inpainting with the FID score of 35.72.

References

[1] 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*, pages 11315–11325, 2022.

[2] Patrick Esser, Robin Rombach, and Bjorn Ommer. Taming transformers for high-resolution image synthesis. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 12873–12883, 2021.

[3] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

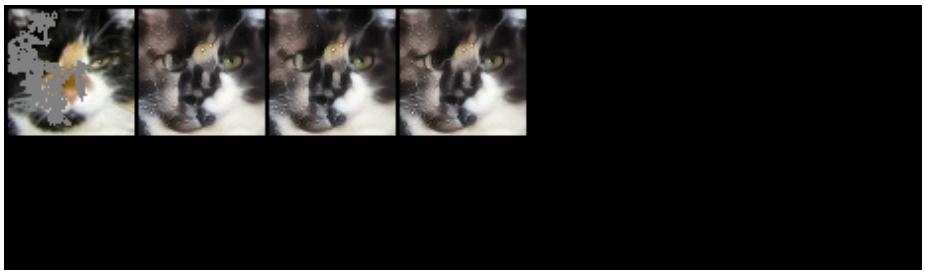


Figure 7: Predicted images of each iteration using the square scheduling function.

Figure 8: Masks of each iteration using the square scheduling function.

Figure 9: Best result of my implementation with FID score 35.72. The sweet spot and total iteration are set to 2 and 4, respectively. The scheduling function is the cosine scheduling function.

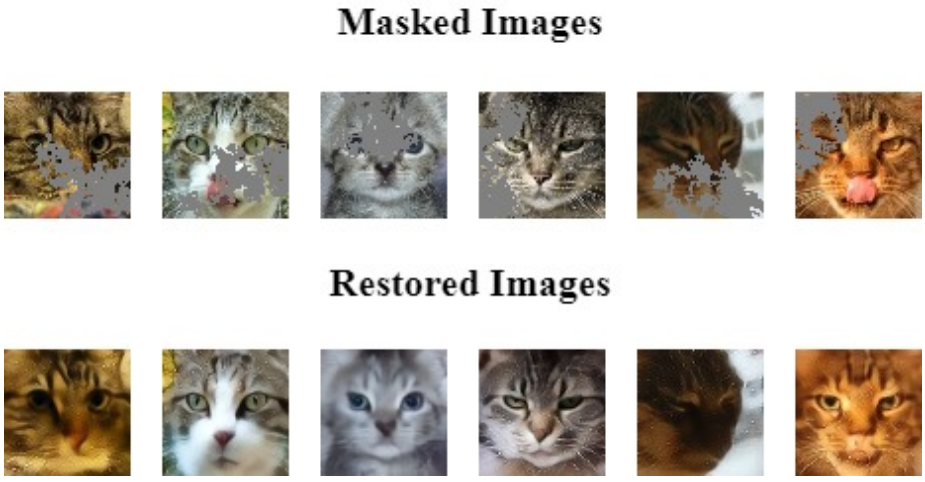


Figure 10: Result of MaskGIT inpainting.