DLP Lab5 Report
MaskGIT for Image Inpainting

Name: Yu-Cheng Ting
ID: 0812212

1 Introduction

In this lab, our task is to implement the inpainting task using MaskGIT model. For inpainting task, the main purpose is to restore the masking part of the image to make the image complete. MaskGIT model is the improvement version of VQGAN. The main architecture of MaskGIT is a GAN structure and the Generator part is the CNN-based Encoder and Decoder with codebook for the latent. Since the GAN-based model is hard to train because of the mode collapse and dynamic equilibrium, TA has provided the weight of the VQGAN, the rest part is to train the bi-directional transformer which is to predict the masking latent code. The main reason why we use MaskGIT instead of VQGAN is because of the better inference time. The VQGAN needs to predict token by token and MaskGIT predicts all the tokens and uses confidence to choose which predicted tokens to be remained. The reason for this technique works is due to the transformer. VQGAN uses transformer and MaskGIT uses bi-directional transformer.

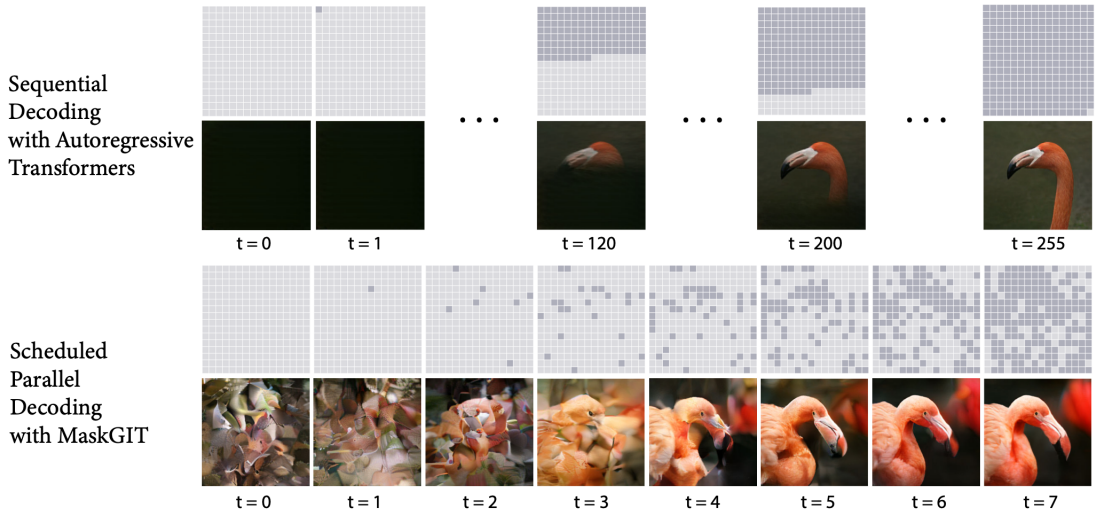


Figure 1: Above is the sampling example of VQGAN, Below is the sampling example of MaskGIT

1.1 Model Architecture

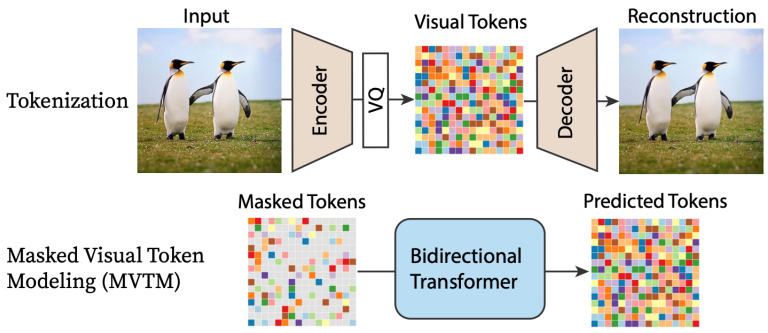


Figure 2: Model Architecture

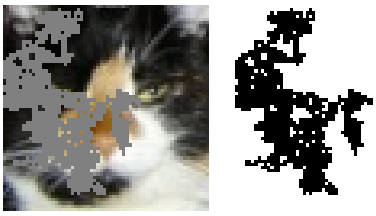
1.2 Dataset

The dataset provided is the cat faces dataset.

- Training dataset contains 12000 RGB images with resolution 64*64
- Validation dataset contains 3000 RGB images with resolution 64*64
- Testing dataset contains 747 masked RGB images with resolution 64*64 and 747 masks

(a) (b)

Figure 3: Training and Validation dataset Example



(a) (b)

Figure 4: Testing Example

2 Implementation details

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

For self attention, it uses Q, K and V to work. It first calculate the attention score by applying inner product of every token and then scale the result. Then we apply softmax to the result. Finally we multiply the softmax result with the V value then we finish the self attention. For the Multi-Head Self Attention, it just divide the original Query and Key to #head sub-queries and sub-keys. Then do the same procedure as in the self attention. The reason for doing this is it outcomes a better performance.
Implementation details:
I first use **nn.Linear** to transform the input x to obtain Q, K and V. After having Q, K and V, we divide them using **torch.view**. After setting all the multihead Q, K and V, I calculate the attention score of the Q and K by applying attention function. Finally, we concatenate all the results back and go through another **nn.Linear** layer to obtain the final result.

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

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_i, ..., \text{head}_n)W^o$$

$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)$$

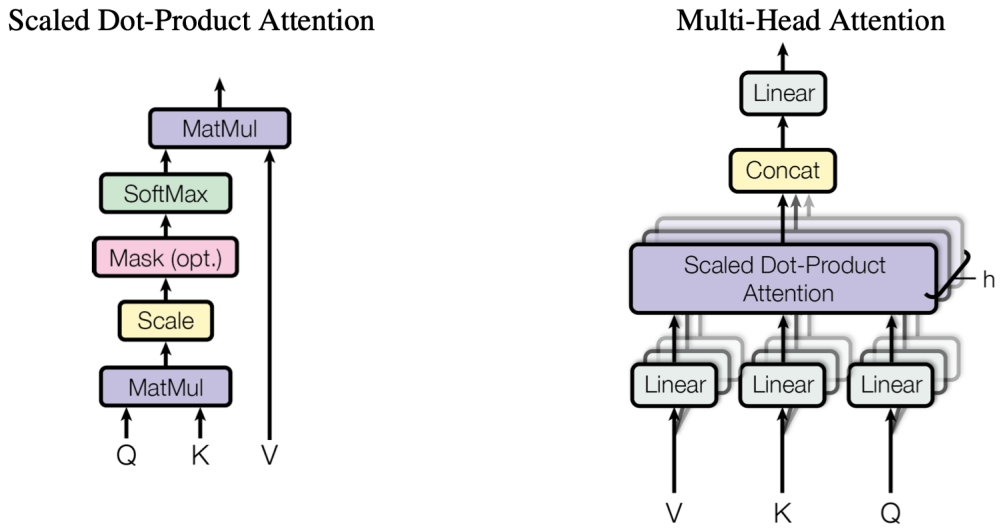


Figure 5: Self Attention (Left), Multi-Head Self Attention (Right)

```
class MultiHeadAttention(nn.Module):
    def __init__(self, dim=768, num_heads=16, attn_drop=0.1):
        super(MultiHeadAttention, self).__init__()
        self.token_dim = dim
        self.num_heads = num_heads
        self.head_dim = dim // num_heads
        assert self.head_dim * self.num_heads == self.token_dim, 'Token
            ↪ dimension must be divisible by number of heads'

        # Linear transformation: Weight of matrices Q, K, V
        self.w_q = nn.Linear(in_features=self.token_dim, out_features=self.
            ↪ token_dim)
        self.w_k = nn.Linear(in_features=self.token_dim, out_features=self.
            ↪ token_dim)
```

```
        self.w_v = nn.Linear(in_features=self.token_dim, out_features=self.
            ↪ token_dim)

        self.dropout = nn.Dropout(attn_drop)

        # Linear transformation of concatenated tensor
        self.linear_output = nn.Linear(in_features=self.token_dim,
            ↪ out_features=self.token_dim)

    def attention(self, q, k ,v, drop=None):
        dim = q.size(-1)
        QK_T = torch.matmul(q, k.transpose(-1, -2))
        QK_T = nn.functional.softmax((QK_T / math.sqrt(dim)), dim=-1)
        if drop is not None:
            QK_T = drop(QK_T)
        output = torch.matmul(QK_T, v)
        return output


    def forward(self, x):
        batch_size, seq_len = x.size(0), x.size(1)
        # Linear transformation to multihead (batch_size, seq_len, dim) -> (
            ↪ batch_size, num_heads, seq_len, head_dim)
        q = self.w_q(x).view(batch_size, seq_len, self.num_heads, self.
            ↪ head_dim).transpose(1,2)
        k = self.w_k(x).view(batch_size, seq_len, self.num_heads, self.
            ↪ head_dim).transpose(1,2)
        v = self.w_v(x).view(batch_size, seq_len, self.num_heads, self.
            ↪ head_dim).transpose(1,2)

        # Concat back (batch_size, num_heads, seq_len, head_dim) -> (
            ↪ batch_size, seq_len, dim)
        output = self.attention(q, k, v, self.dropout)
        output = output.transpose(1, 2).contiguous().view(batch_size, seq_len,
            ↪  self.token_dim)

        return self.linear_output(output)
```

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

For MVTM, it stands for Masked Visual Token Modeling. The main idea of this comes from the human drawing logic. Instead of masking every token one by one of the whole sequence, we use bi-directional transformer to masking all the tokens. It is just like the drawing logic, we do not draw the painting area by area, we first have the whole concept and then do the details. The number of the tokens to be masked is decided by the mask scheduling function $\gamma(r) \in [0, 1)$. We sample a ratio r and calculate the numbers of tokens by the function $\lceil \gamma(r) \cdot N \rceil$ where N is the number of all tokens. After masking the tokens, we feed the masked tokens into transformer to predict the new tokens. The transformer will predict the probability of each latent code to decide which one has the highest probability. We then use the **Cross Entropy Loss** to compute the loss between the original unmasked tokens and the predicted tokens.
Implementation:
(Forward function of MaskGIT) We first get the latent code by calling **self.encode_to_z()** to get **z_indices**. Then we calculate the number of tokens to be masked. Then we initial a mask with all *False* in the mask and randomly replace the masking parts. After having the correct mask, we put it on the *z_indices* and feed it to the transformer to get the *logits*.
(Traning code) For training code, we do the simple classification training by calling **F.cross_entropy()** to compute the loss between predicted tokens and the original tokens.

```
# forward function of MaskGIT
def forward(self, x):
```

```
        _, z_indices = self.encode_to_z(x)
        n = math.floor(self.gamma(np.random.uniform()) * z_indices.shape[1])
        sample = torch.rand(z_indices.shape, device=z_indices.device).topk(n,
            ↪ dim=1).indices
        mask = torch.zeros(z_indices.shape, dtype=torch.bool, device=z_indices
            ↪ .device)
        mask.scatter_(dim=1, index=sample, value=True)

        masked_indices = self.mask_token_id * torch.ones_like(z_indices,
            ↪ device=z_indices.device)
        a_indices = mask * masked_indices + (~mask) * z_indices

        logits = self.transformer(a_indices)
        return logits, z_indices
```

```
# Training code:
for epoch in range(args.start_from_epoch+1, args.epochs+1):
    total_loss = 0
        for img in train_loader:
        train_transformer.optim.zero_grad()
        img = img.to(args.device)
        logits, target = train_transformer.model(img)
        loss = F.cross_entropy(logits.reshape(-1, logits.size(-1)), target.
            ↪ reshape(-1))
        loss.backward()
        train_transformer.optim.step()

        total_loss += loss.cpu().detach().item()
    train_transformer.scheduler.step()
```

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

After training the model, it is time for checking our outcome by doing the inference part. The inference part of MaksGIT is called iterative decoding. The originally iterative decoding is to mask all the tokens and feed into the transformer and get the result. We then calculate the confidence of each latent code to decide which token to be stayed or to be re-predict. The number of tokens to be stayed is decided by the mask scheduling function $\gamma(r)$. Different form training, the ratio r is obtained by t/T where t is the current step and T is the whole steps. By iteratively doing this, we will get the final result at step T. For the inpainting task, we just focus on the masked part and the rest remains the same. The confidence part of non-mask, we set them as inf so that when ordering the confidence, they will not effect the outcome.

```
# Inpainting function:
def inpainting(self, z_indices, mask_bc, ratio):
    masked_indices = self.mask_token_id * torch.ones_like(z_indices, device=
        ↪ z_indices.device)
    z_indices[mask_bc] = masked_indices[mask_bc]

    logits = self.transformer(z_indices)
    prob_dist = torch.softmax(logits, dim=-1)
    z_indices_predict_prob, z_indices_predict = torch.max(prob_dist, dim=-1)

    g = torch.distributions.Gumbel(0, 1).sample(z_indices_predict_prob.shape).
        ↪ to(z_indices.device)
    temperature = self.choice_temperature * (1 - ratio)
    confidence = z_indices_predict_prob + temperature * g
    confidence[~mask_bc] = float('inf')

    num_mask = torch.sum(mask_bc)
    n = math.floor(self.gamma(ratio) * num_mask)
```

```
    _, indices = torch.sort(confidence, dim=-1)
    new_mask = torch.zeros_like(mask_bc, dtype=torch.bool, device=mask_bc.
        ↪ device)
    new_mask.scatter_(dim=-1, index=indices[:, :n], value=True)
    new_mask[~mask_bc] = mask_bc[~mask_bc]

    return z_indices_predict, new_mask
```

```
# Complete inpainting function
def inpainting(self, image, mask_b, i):
    self.model.eval()
    with torch.no_grad():
        _, z_indices = self.model.encode_to_z(image)
        z_indices_predict = z_indices
        mask_bc = mask_b.to(self.device)

        for step in range(self.total_iter):
            ratio = step / self.total_iter
            z_indices_predict, mask_bc = self.model.inpainting(
                ↪ z_indices_predict, mask_bc, ratio)
            z_indices_predict[~mask_b] = z_indices[~mask_b]
```

3 Experiment results

3.1 The best testing fid

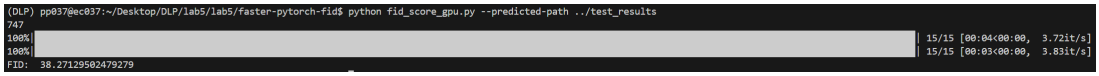


Figure 6: Best FID result 38.271 with Total Step 10

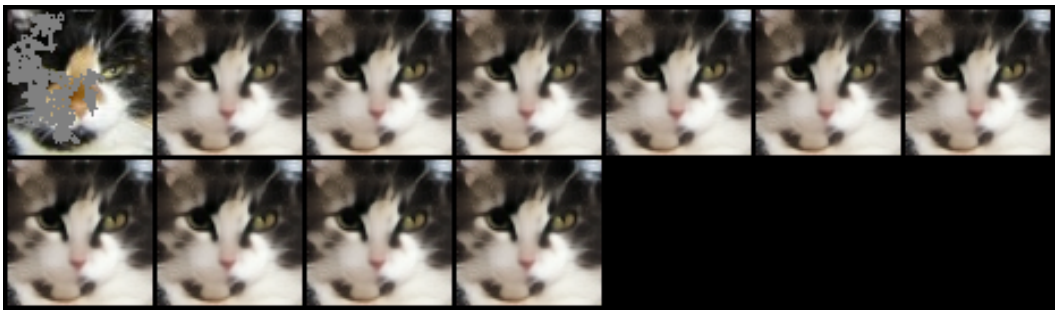


Figure 7: Predicted image with Total Step 10

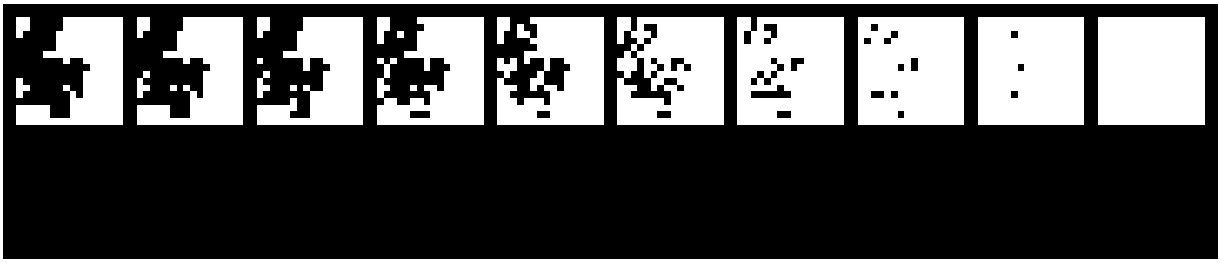


Figure 8: Mask scheduling in latent domain with Total Step 10

For the training strategy, I used gradient accumulation. Since my machine is not good and the batch size cannot be set big. So the gradient accumulation technique will help me train model well. I set the total step 10.

3.2 Comparison figures with different mask scheduling parameters setting

We can clearly see that different settings will have different results. Compared with other two functions, Linear function will have the least steps to have all masks disappear but it will not have a good result in image. For cosine and square, they have quite the same result but square is faster.

Figure 9: $\gamma(r)$ Cosine with Total Step 10 FID 38.271

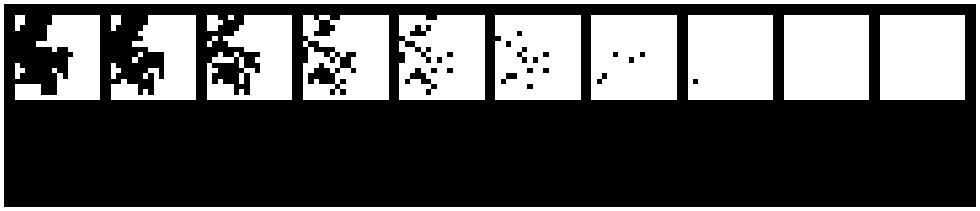


Figure 10: $\gamma(r)$ Linear with Total Step 10 FID 38.301

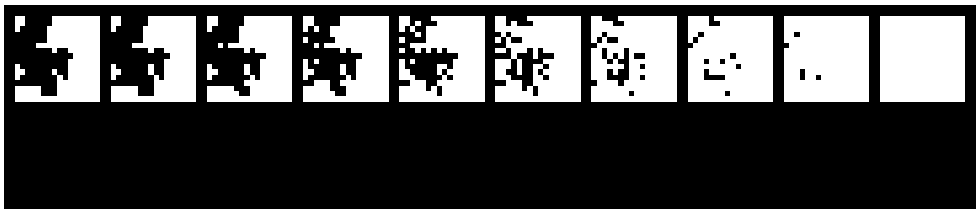
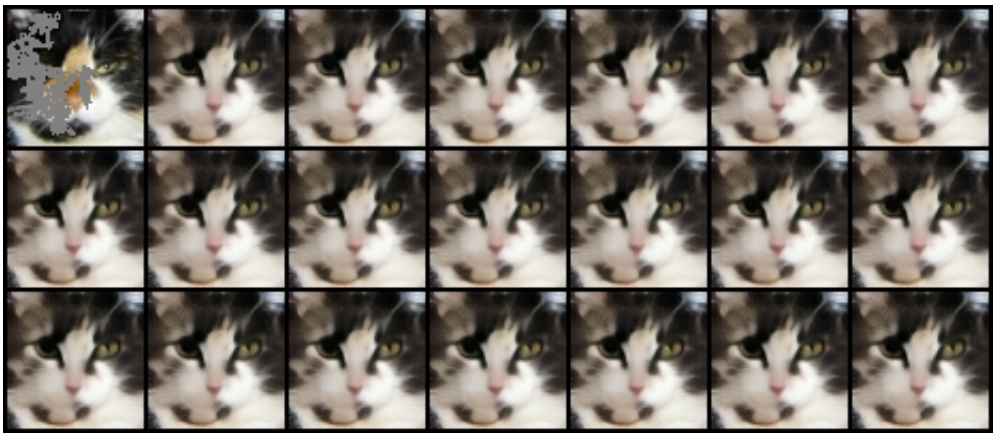


Figure 11: $\gamma(r)$ Square with Total Step 10 FID 38.271

4 Discussion

For different total steps settings, the best model I learned cannot see the changes between steps. However, for the bad model checkpoint, I can see the small changes between steps. Below are the comparison figures.



(a) Good Model with Total Step 20 FID 38.301

(b) Bad Model with Total Step 20 FID 82.431

Figure 12: Comparison

For the first image and the last image generated by the good model, we cannot see the difference. But for bad model, we can clearly see the difference between the first and the last generated model. The right eye of the cat initially is green and the last one is all black.