Lab5 MaskGIT for Image Inpainting Report

student id: 313551097
student name: 鄭淮薰

Introduction

In this lab, I implemented the MaskGIT model for image inpainting. The goal of this lab is to fill the missing part of the image with the help of the MaskGIT model. In the training stage, I used pre-trained VQGAN to encode the image and masked the image with a random mask. Then, I trained the model with the masked image and the original image. In the inference stage, I used different iterative mask scheduling methods to fill the missing part of the image. After the training and inference stages, I calculated the FID score to evaluate the performance of the model.

Implementation Details

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

First, I figure out the size of each attention head by dividing the input size (`dim`) by the number of heads (`num_heads`). This gives me `head_dim = dim // num_heads`. Then, I create a linear layer for `qkv` (queries, keys, and values), making the input size three times bigger since `q`, `k`, and `v` each need their own transformation.

To prevent the dot product (multiplication of `q` and `k`) from getting too large, which could make the softmax output too small and hurt the gradients, I scale down `q` and `k` by multiplying them by `scale = head_dim ** -0.5`. In the forward function, I apply the linear transformation to `x`, reshape it, and change its order so I can split out `q`, `k`, and `v` easily.

Next, I calculate the attention scores by multiplying `q` and `k`, scale the result, and apply softmax and dropout to get the attention weights (`attn`). Then, I multiply `attn` with `v` to get the updated `x`. Finally, I pass `x` through another linear layer (`out`) to combine the results from all heads, giving me the final output.

```
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 # 768 // 16 = 48
        self.qkv = nn.Linear(dim, dim * 3, bias=False) # 768 * 3 = 2304
        self.scale = self.head_dim ** -0.5
        self.attn_drop = nn.Dropout(attn_drop)
        self.out = nn.Linear(dim, dim)

    def forward(self, x):
        # Perform linear transformation for Q, K, V
        qkv = self.qkv(x)
        qkv = qkv.reshape(x.shape[0], x.shape[1], 3, self.num_heads,
self.head_dim)
        qkv = qkv.permute(2, 0, 3, 1, 4)
        q, k, v = qkv[0], qkv[1], qkv[2]

        # Calculate attention scores and apply softmax
        '''
        q: (batch_size, num_heads, num_image_tokens, head_dim)
        k: (batch_size, num_heads, num_image_tokens, head_dim)
        k.transpose(-2, -1): (batch_size, num_heads, head_dim,
num_image_tokens)
        '''
        attn = (q @ k.transpose(-2, -1)) * self.scale
        attn = attn.softmax(dim=-1)
        attn = self.attn_drop(attn)

        # Apply attention to V
        x = attn @ v # (batch_size, num_image_tokens, num_heads, head_dim)
        x = x.transpose(1, 2).reshape(x.shape[0], x.shape[2], self.dim)
        x = self.out(x)
        return x
```

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

1. MVTM

In the MVTM model, I'm using VQGAN as the encoder and a transformer as the decoder. I've implemented two key functions: `encode_to_z` and `gamma_func`.

The `encode_to_z` function takes input images and encodes them into latent codes: `z_q` and `z_indices`. Here, `z_q` represents the latent code produced by the VQGAN encoder, while `z_indices` represents the quantized latent code after the encoding process.

```
@torch.no_grad()
def encode_to_z(self, x):
    z_q, z_indices, _ = self.vqgan.encode(x)
    z_indices = z_indices.reshape(z_q.shape[0], -1)

    return z_q, z_indices
```

The `gamma_func` function is designed to generate the mask rate using different scheduling methods like linear, cosine, or square. During training, the function randomly generates a mask rate. During inference, however, it calculates the mask rate based on the current step and the total number of steps, adjusting it according to the chosen scheduling method.

```
def gamma_func(self, mode="cosine"):
    if mode == "linear":
        return lambda ratio: 1 - ratio
    elif mode == "cosine":
        return (lambda ratio: np.cos(ratio * np.pi / 2))
    elif mode == "square":
        return lambda ratio: 1 - ratio ** 2
    else:
        raise NotImplementedError
```

The forward function for the Masked VQ-Transformer Model (MVTM) works as follows:

1. I first encode the input image into latent code `z_q` and quantized latent code `z_indices`.
2. Next, I randomly generate a mask ratio `mask ratio` between 0 and 1.
3. Using this mask ratio, I create a mask that selectively hides some tokens in `z_indices`.
4. The masked `z_indices` are then passed through the transformer to make predictions, producing logits.
5. Finally, I return both the logits (the transformer's predicted probabilities) and `z_indices` (the ground truth).

```
def forward(self, x):
    # encode the input image to latent and quantized latent
    _, z_indices = self.encode_to_z(x)

    # In training, the mask ratio is randomly sampled
    ratio = np.random.uniform(0, 1)
    mask = torch.rand_like(z_indices.float()) < mask_ratio

    # Mask the tokens
    masked_indices = z_indices.clone()
    masked_indices[mask] = self.mask_token_id

    # Pass the masked tokens to the transformer
    logits = self.transformer(masked_indices)

    z_indices = z_indices #ground truth
    logits = logits #transformer predict the probability of tokens

    return logits, z_indices
```

2. Forward and Loss

During training, I calculate the cross-entropy loss between the model's output logits and the ground truth `z_indices`, then backpropagate the loss. I use the Adam optimizer, and I update the model's parameters every `accum_grad` steps.

```
def train_one_epoch(self, train_loader, epoch):
    self.model.train()

    total_loss = 0.0
    for i, image in (pbar := tqdm(enumerate(train_loader),
total=len(train_loader))):
        x = image.to(self.args.device)
        logits, z_indices = self.model(x)

        # compute loss
        loss = F.cross_entropy(
            logits.view(-1, logits.size(-1)), z_indices.view(-1)
        )
        total_loss += loss.item()

        # backprop
        loss.backward()

        if i % self.args.accum_grad == 0:
            self.optim.step()
            self.optim.zero_grad()

        # update progress bar
        pbar.set_description(
            f"(train) Epoch {epoch} - Loss: {loss.item():.4f}",
            refresh=False
        )

    if self.args.log:
        self.writer.add_scalar(
            "Loss/train", total_loss / len(train_loader), epoch
        )
        wandb.log({"Loss/train": total_loss / len(train_loader)})
    return total_loss / len(train_loader)
```

```
def configure_optimizers(self):
    optimizer = torch.optim.Adam(
        self.model.parameters(), lr=args.learning_rate, betas=(0.9, 0.96)
    )
    scheduler = None
    return optimizer,scheduler
```

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

When performing inpainting, I start by encoding the original image into latent code `z_q` and quantized latent code `z_indices`. Then, I run `z_indices` and the mask through the inpainting function for an initial pass, which gives me a partially restored `z_indices` and an updated mask. I repeat this process, feeding the restored `z_indices` and updated mask back into the inpainting function until the specified number of iterations is completed. Along the way, I save the results from each inpainting step for later visualization.

```
def inpainting(self,image,mask_b,i): #MakGIT inference
    #save all iterations of masks in latent domain
    maska = torch.zeros(self.total_iter, 3, 16, 16)
    #save all iterations of decoded images
    imga = torch.zeros(self.total_iter+1, 3, 64, 64)
    mean = torch.tensor([0.4868, 0.4341,
0.3844],device=self.device).view(3, 1, 1)
    std = torch.tensor([0.2620, 0.2527,
0.2543],device=self.device).view(3, 1, 1)
    ori=(image[0]*std)+mean
    imga[0]=ori #mask the first image be the ground truth of masked image

    self.model.eval()
    with torch.no_grad():
        z_indices = self.model.encode_to_z(image)[1]
        mask_num = mask_b.sum() #total number of mask token
        z_indices_predict=z_indices
        mask_bc=mask_b
        mask_b=mask_b.to(device=self.device)
        mask_bc=mask_bc.to(device=self.device)

        ratio = 0
        #iterative decoding for loop design
        for step in range(self.total_iter):
            if step == self.sweet_spot:
                break
            ratio = (step+1) / self.total_iter

            z_indices_predict, mask_bc = self.model.inpainting(
                z_indices_predict, mask_bc, ratio, self.mask_func
            )

            #(... convert z_indices_predict to image for visualization
...)
            #(... save original mask and the updated mask by scheduling
...)

        # (... save the decoded image of the sweet spot ...)
```

In each inpainting round, I start by masking `z_indices` based on `mask_bc`, setting the masked tokens to the mask token ID. Then, I pass the masked `z_indices` through the transformer to get the predicted logits.

Next, I apply softmax to the logits to get probability distributions and select the token with the highest probability for each position. If a token isn't meant to be masked (when the mask is set to False), I set its probability to infinity to prevent the transformer from modifying it.

After that, I add gumbel noise to the probabilities to generate confidence scores and sort them based on confidence. I calculate the mask ratio using the `gamma_func`, which adapts based on the chosen mask scheduling method.

Finally, I determine which tokens to mask using the mask ratio and confidence scores, masking those with confidence lower than a set threshold. The process ends by returning the restored `z_indices` and the updated mask.

```
@torch.no_grad()
def inpainting(self, z_indices, mask_bc, ratio, mask_func):
    z_indices_masked = z_indices.clone()
    z_indices_masked[mask_bc] = self.mask_token_id

    #Pass the masked tokens to the transformer, and get the logits
    logits = self.transformer(z_indices_masked)

    #Apply softmax to convert logits into a probability distribution
    logits = 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)

    #hint: If mask is False, the probability should be set to infinity,
    # so that the tokens are not affected by the transformer's prediction
    z_indices_predict_prob[~mask_bc] = float('inf')

    #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_predict_prob.device) # gumbel noise
    temperature = self.choice_temperature * (1 - ratio)
    confidence = z_indices_predict_prob + temperature * g

    #sort the confidence for the rank
    sorted_confidence = torch.sort(confidence, dim=-1)

    z_indices_predict = z_indices_predict * mask_bc + z_indices * ~mask_bc

    #define how much the iteration remain predicted tokens by mask
scheduling
    mask_ratio = self.gamma_func(mask_func)(ratio)
    mask_len = math.floor(mask_ratio * mask_bc.sum())
    bound = sorted_confidence.values[:, mask_len].unsqueeze(-1)
    mask_bc = confidence < bound

    return z_indices_predict, mask_bc
```

Discussion

A. Comparison between different mask scheduling methods

I implemented three mask scheduling methods: linear, cosine, and square. As the results table shows, the FID score for three methods is very close, with the linear method achieving the best score. This suggests that the linear scheduling method is more effective at inpainting images than the other two methods.

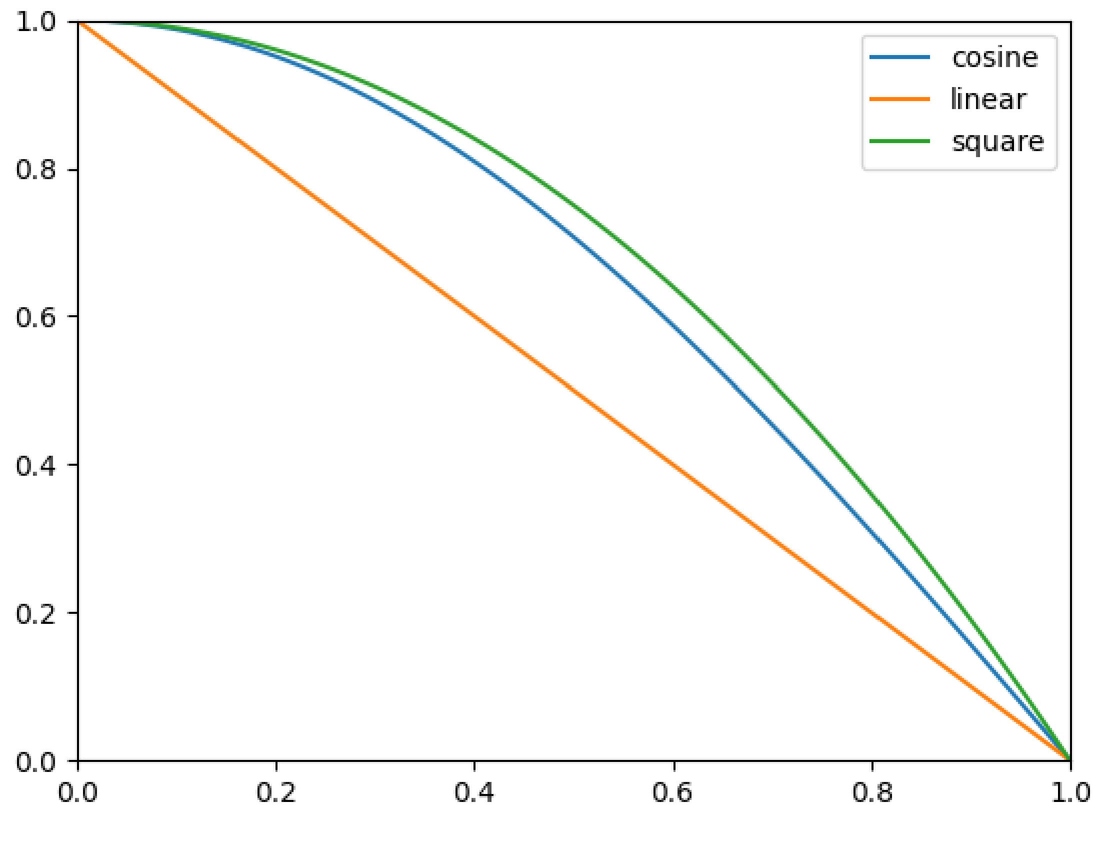
Method	Average FID Score	Standard Deviation

Method	Average FID Score	Standard Deviation
Cosine	27.06	0.32
Linear	26.98	0.28
Square	27.00	0.28

Experiment Score

Part1: Prove your code implementation is correct

Show iterative decoding

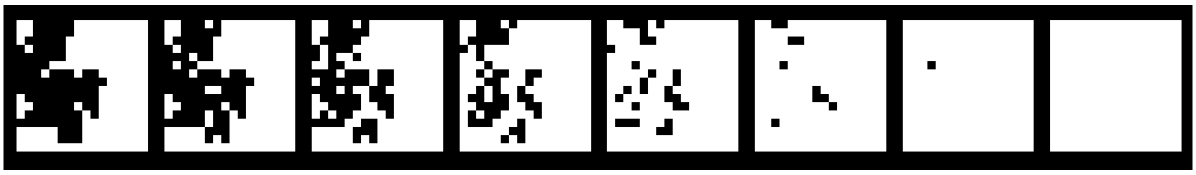


- cosine
- linear
- square

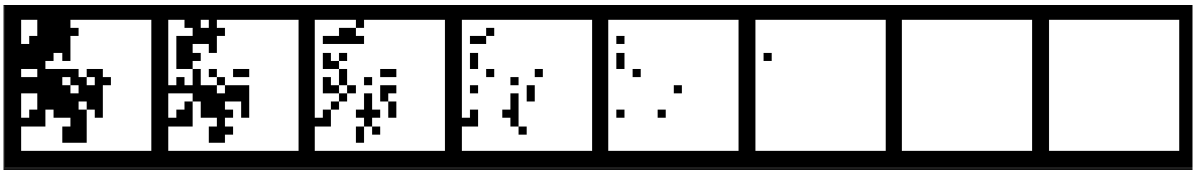
(a) Mask in latent domain

From the following images, we can observe that the cosine and square methods fill less mask in the early stage, and gradually increase the amount of filling in the later stage, while the linear method maintains a certain amount of filling.

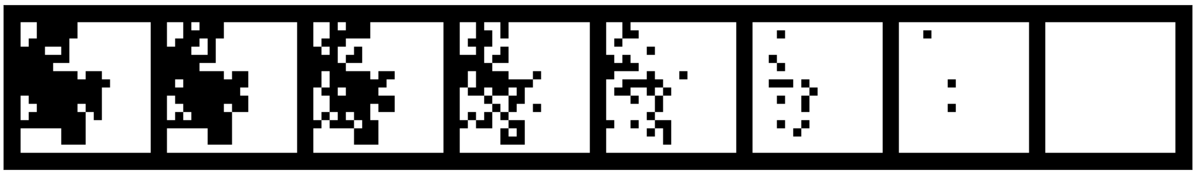
cosine



linear

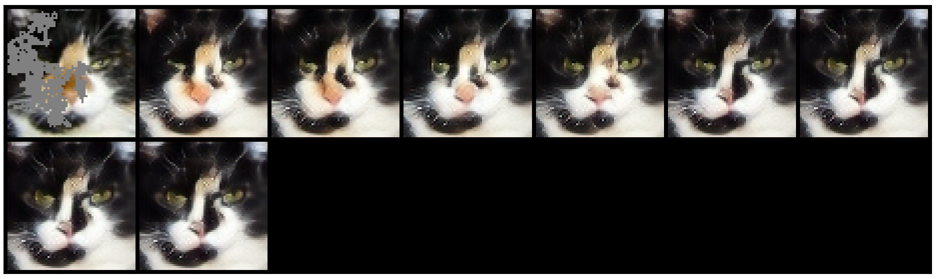


square

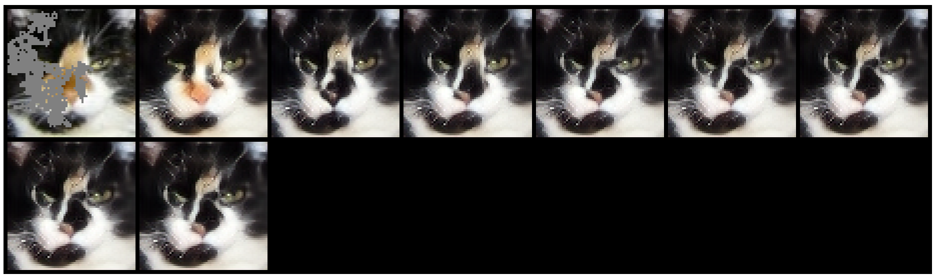


(b) Predicted image

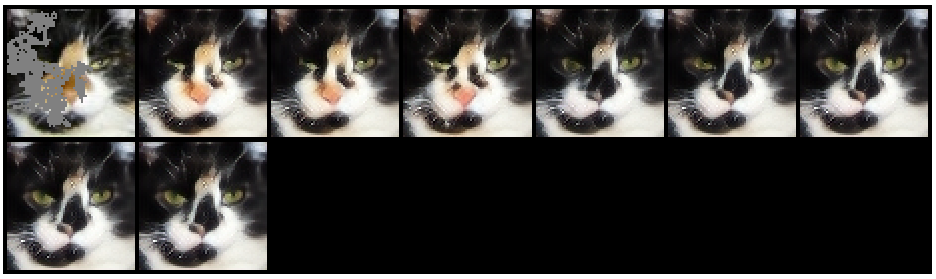
cosine



linear

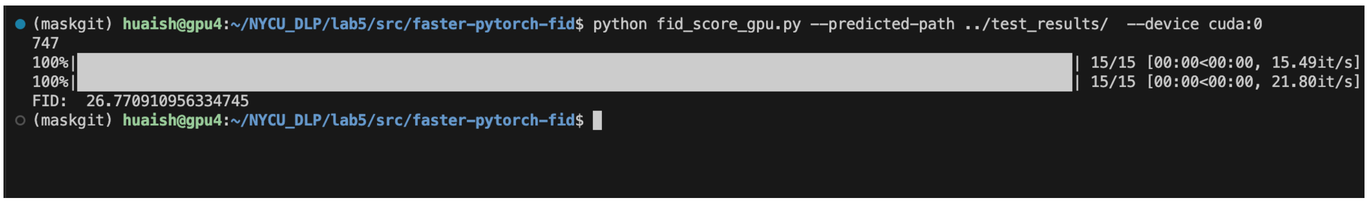


square



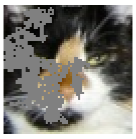
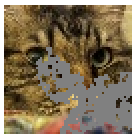
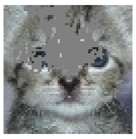
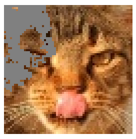
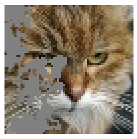
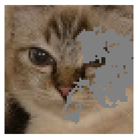
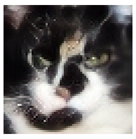
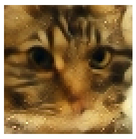
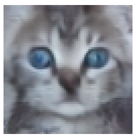
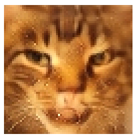
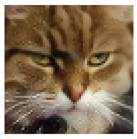
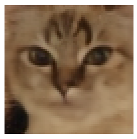
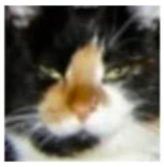
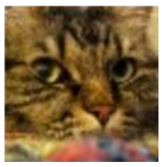
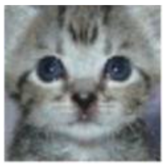
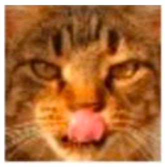
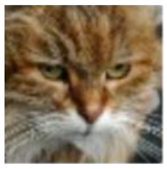
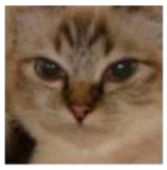
Part2: The Best FID Score

Screenshot



```
● (maskgit) huaish@gpu4:~/NYCU_DLP/lab5/src/faster-pytorch-fid$ python fid_score_gpu.py --predicted-path ../test_results/  --device cuda:0
  747
  100%|██████████████████████████████████████████████████████████████████| 15/15 [00:00<00:00, 15.49it/s]
  100%|██████████████████████████████████████████████████████████████████| 15/15 [00:00<00:00, 21.80it/s]
  FID:  26.770910956334745
○ (maskgit) huaish@gpu4:~/NYCU_DLP/lab5/src/faster-pytorch-fid$ 
```

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

Masked Images						
MaskGIT Inpainting Results						
Ground Truth						

The setting about training strategy, mask scheduling parameters, and so on

- learning rate: 1e-4
- batch size: 10
- epochs: 300
- optimizer: Adam
- sweet spot: 8
- total iteration: 8
- mask function: linear