Lab3 313605019 方敏

1. Introduction

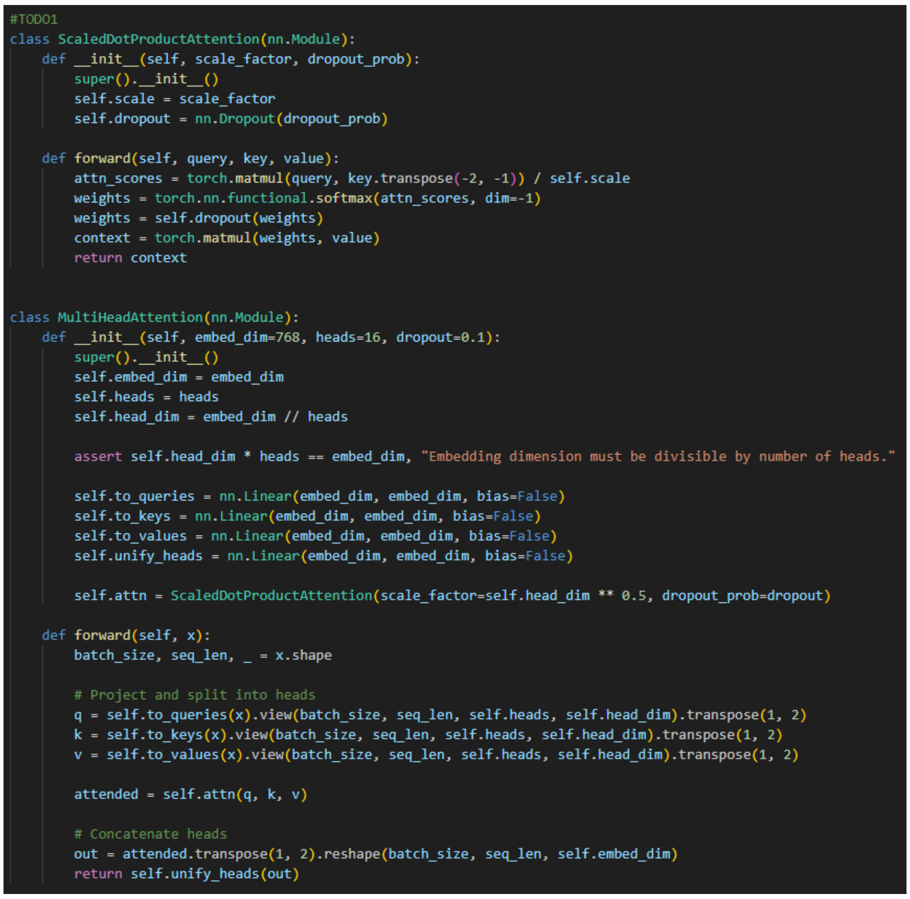
In this lab, we implement the task of image inpainting using MaskGIT—a fast, parallel image generation model. The goal of image inpainting is to fill in missing or masked-out regions of an image in a way that looks realistic and coherent with the surrounding content.

Our pipeline is built in two stages. First, we use a pretrained VQGAN to encode images into discrete latent tokens. Then, we train a Bidirectional Transformer that predicts these tokens based on a partially masked version of the input. Unlike traditional autoregressive models, MaskGIT predicts multiple tokens in parallel, making the process significantly faster while maintaining high generation quality.

We implement the multi-head attention mechanism from scratch—without using PyTorch's built-in modules, and evaluates using FID scores.

2. Implementation

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



```python
#TODO1
class ScaledDotProductAttention(nn.Module):
    def __init__(self, scale_factor, dropout_prob):
        super().__init__()
        self.scale = scale_factor
        self.dropout = nn.Dropout(dropout_prob)

    def forward(self, query, key, value):
        attn_scores = torch.matmul(query, key.transpose(-2, -1)) / self.scale
        weights = torch.nn.functional.softmax(attn_scores, dim=-1)
        weights = self.dropout(weights)
        context = torch.matmul(weights, value)
        return context


class MultiHeadAttention(nn.Module):
    def __init__(self, embed_dim=768, heads=16, dropout=0.1):
        super().__init__()
        self.embed_dim = embed_dim
        self.heads = heads
        self.head_dim = embed_dim // heads

        assert self.head_dim * heads == embed_dim, "Embedding dimension must be divisible by number of heads."

        self.to_queries = nn.Linear(embed_dim, embed_dim, bias=False)
        self.to_keys = nn.Linear(embed_dim, embed_dim, bias=False)
        self.to_values = nn.Linear(embed_dim, embed_dim, bias=False)
        self.unify_heads = nn.Linear(embed_dim, embed_dim, bias=False)

        self.attn = ScaledDotProductAttention(scale_factor=self.head_dim ** 0.5, dropout_prob=dropout)

    def forward(self, x):
        batch_size, seq_len, _ = x.shape

        # Project and split into heads
        q = self.to_queries(x).view(batch_size, seq_len, self.heads, self.head_dim).transpose(1, 2)
        k = self.to_keys(x).view(batch_size, seq_len, self.heads, self.head_dim).transpose(1, 2)
        v = self.to_values(x).view(batch_size, seq_len, self.heads, self.head_dim).transpose(1, 2)

        attended = self.attn(q, k, v)

        # Concatenate heads
        out = attended.transpose(1, 2).reshape(batch_size, seq_len, self.embed_dim)
        return self.unify_heads(out)
```

i. Embedding Dimension Splitting

The total embedding dimension (embed_dim, default 768) is divided into heads (default 16) subspaces. Each head operates on vectors of size head_dim = embed_dim // heads.

ii. Linear Projections to Q, K, V

The input x is linearly projected to obtain the query (Q), key (K), and value (V) tensors.

iii. Head Splitting and Transposition

Q, K, and V are reshaped into shape [batch_size, heads, seq_len, head_dim], enabling each head to process data independently.

iv. Scaled Dot-Product Attention

Each head applies ScaledDotProductAttention module to compute attention weights, incorporating dropout for regularization.

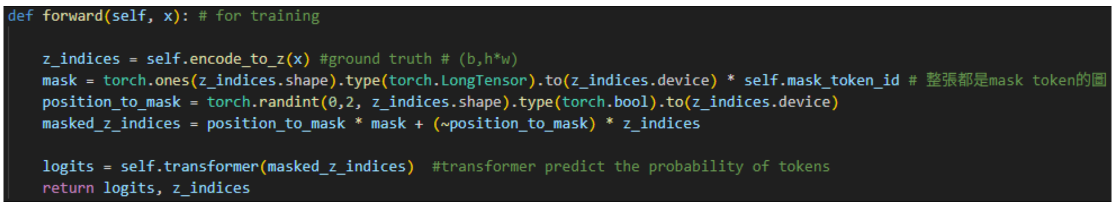
v. Concatenation and Output Projection

Outputs from all heads are concatenated and passed through a final linear layer unify_heads to restore the original embedding dimension.

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

i. Forward

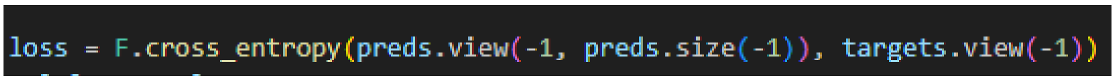
The input image is encoded into discrete latent tokens (z_indices). Then, masked versions of these tokens are passed to a transformer which predicts the probabilities (logits) of all tokens. Finally, it returns both the predicted logits and the ground truth tokens.



```python
def forward(self, x): # for training

    z_indices = self.encode_to_z(x) #ground truth # (b,h*w)
    mask = torch.ones(z_indices.shape).type(torch.LongTensor).to(z_indices.device) * self.mask_token_id # 整張都是mask token的圖
    position_to_mask = torch.randint(0,2, z_indices.shape).type(torch.bool).to(z_indices.device)
    masked_z_indices = position_to_mask * mask + (~position_to_mask) * z_indices

    logits = self.transformer(masked_z_indices)  #transformer predict the probability of tokens
    return logits, z_indices
```

ii. Loss

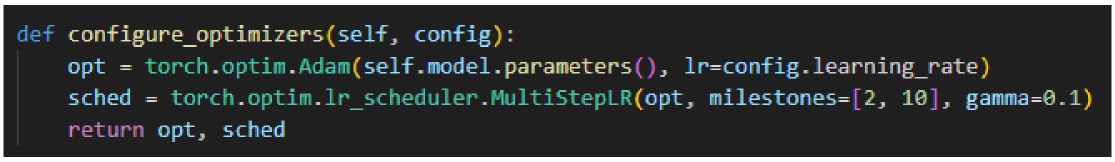
The loss is computed using cross-entropy between the transformer's predicted logits and the true token indices. This trains the model to fill in the missing tokens based on the visible ones.



```
loss = F.cross_entropy(preds.view(-1, preds.size(-1)), targets.view(-1))
```

iii. MVTM

The training uses the Adam optimizer and a MultiStepLR scheduler to adapt the learning rate and improve model convergence.



```
def configure_optimizers(self, config):
    opt = torch.optim.Adam(self.model.parameters(), lr=config.learning_rate)
    sched = torch.optim.lr_scheduler.MultiStepLR(opt, milestones=[2, 10], gamma=0.1)
    return opt, sched
```

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

i. Initial masking

Randomly mask a portion of the codebook indices.

ii. Iteratively decode

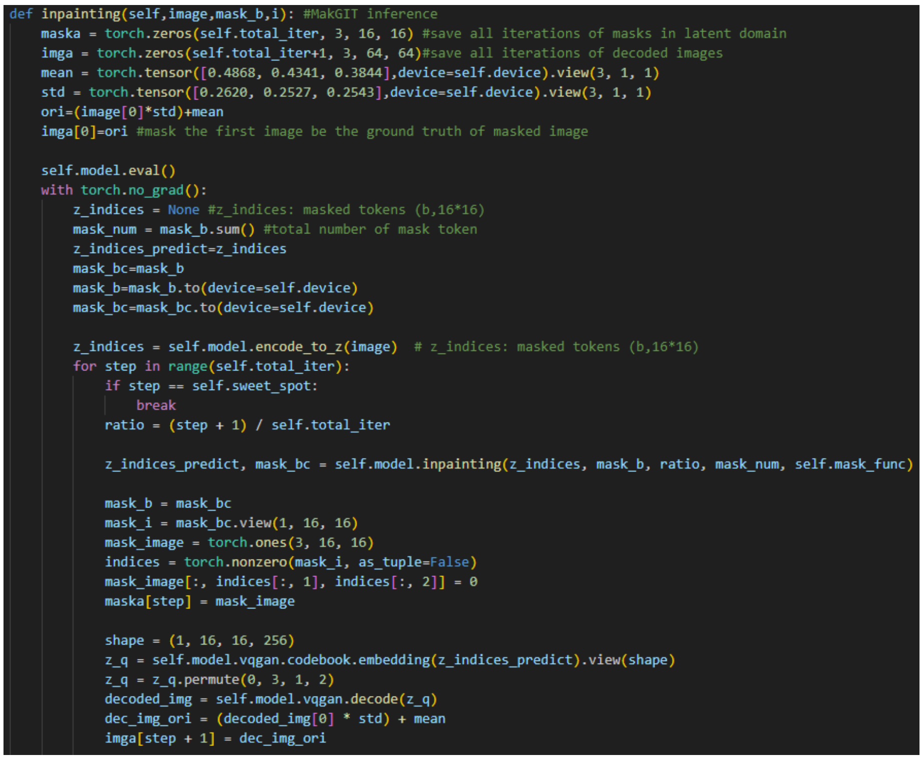
a. Feed the masked indices into the transformer.

b. The transformer predicts logits at each position, which are converted into token probabilities using softmax.

c. A confidence score is computed using max probabilities and Gumbel noise.

d. New masked positions are selected based on confidence scores.

e. The predicted tokens are used to update the masked indices.



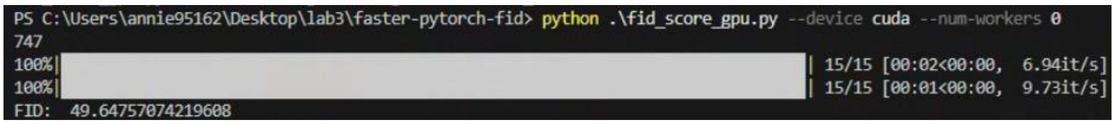
```
def inpainting(self,image,mask_b,i): #MakGIT inference
    maska = torch.zeros(self.total_iter, 3, 16, 16) #save all iterations of masks in latent domain
    imga = torch.zeros(self.total_iter+1, 3, 64, 64)#save all iterations of decoded images
    mean = torch.tensor([0.4868, 0.4341, 0.3844],device=self.device).view(3, 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 = None #z_indices: masked tokens (b,16*16)
        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)

        z_indices = self.model.encode_to_z(image)  # z_indices: masked tokens (b,16*16)
        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, mask_b, ratio, mask_num, self.mask_func)

            mask_b = mask_bc
            mask_i = mask_bc.view(1, 16, 16)
            mask_image = torch.ones(3, 16, 16)
            indices = torch.nonzero(mask_i, as_tuple=False)
            mask_image[:, indices[:, 1], indices[:, 2]] = 0
            maska[step] = mask_image

            shape = (1, 16, 16, 256)
            z_q = self.model.vqgan.codebook.embedding(z_indices_predict).view(shape)
            z_q = z_q.permute(0, 3, 1, 2)
            decoded_img = self.model.vqgan.decode(z_q)
            dec_img_ori = (decoded_img[0] * std) + mean
            imga[step + 1] = dec_img_ori
```

3. Discussion

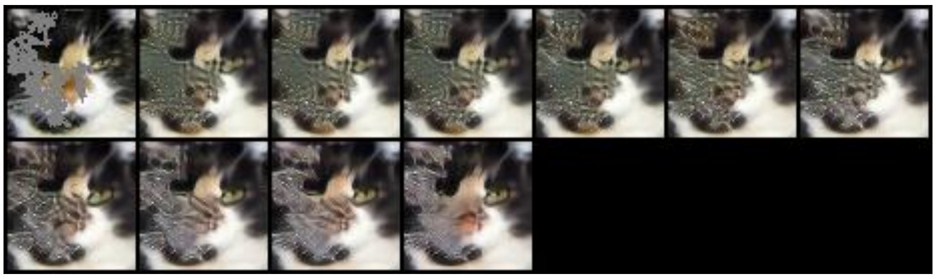
In theory, the cosine schedule should perform better than linear and square schedules because it starts with a high masking ratio to encourage context learning and gradually reduces it for finer predictions. However, in my experiment, the results showed Linear > Square > Cosine. I think this is because training epochs I set is too small, which might not have been sufficient for the model to adapt to the high initial masking ratio of the cosine schedule. Additionally, the initial masking ratio for cosine may set too high, making the early learning phase too challenging. But the scores of all three schedules were quite close, indicating that each performed reasonably well overall.

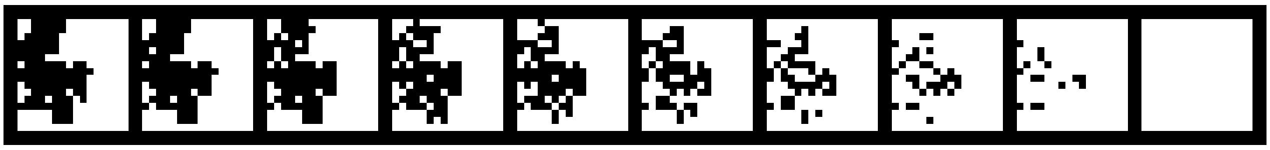
4. Experiment Score

Cosine:

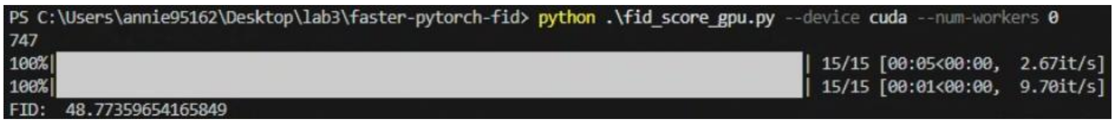


```
PS C:\Users\annie95162\Desktop\lab3\faster-pytorch-fid> python .\fid_score_gpu.py --device cuda --num-workers 0
747
100%|                                                                    | 15/15 [00:02<00:00,  6.94it/s]
100%|                                                                    | 15/15 [00:01<00:00,  9.73it/s]
FID:  49.64757074219608
```

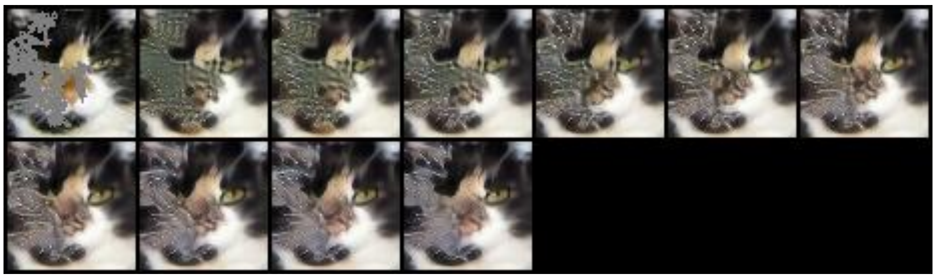


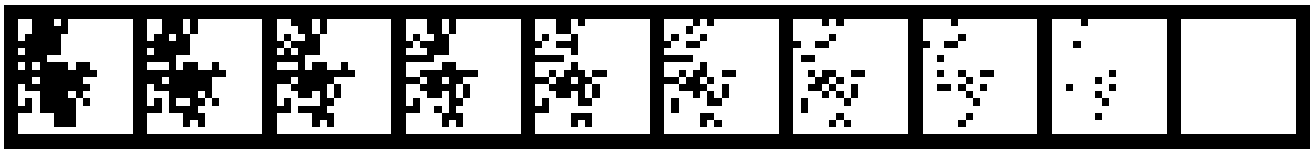


Linear:

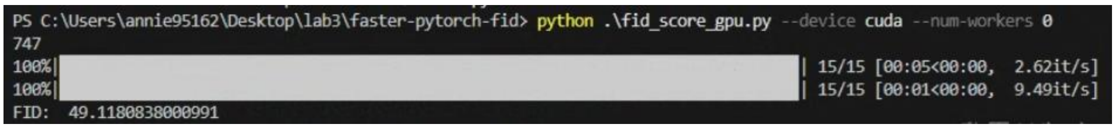


```
PS C:\Users\annie95162\Desktop\lab3\faster-pytorch-fid> python .\fid_score_gpu.py --device cuda --num-workers 0
747
100%|                                                                    | 15/15 [00:05<00:00,  2.67it/s]
100%|                                                                    | 15/15 [00:01<00:00,  9.70it/s]
FID:  48.77359654165849
```

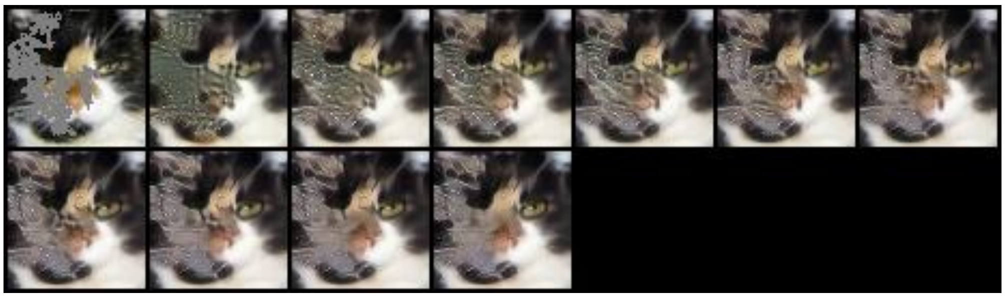


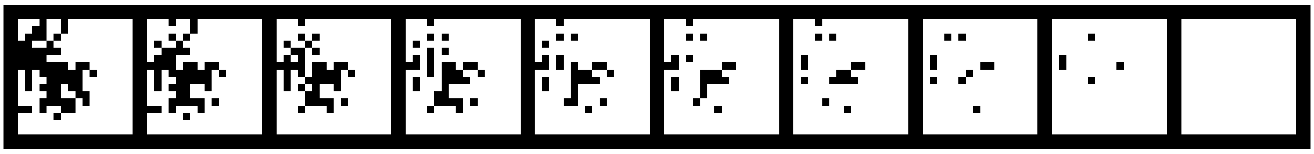


Square:



```
PS C:\Users\annie95162\Desktop\lab3\faster-pytorch-fid> python .\fid_score_gpu.py --device cuda --num-workers 0
747
100%|                                                                    | 15/15 [00:05<00:00,  2.62it/s]
100%|                                                                    | 15/15 [00:01<00:00,  9.49it/s]
FID:  49.1180838000991
```





Best: linear

Command: --learning-rate 0.0001 –epoch 100 --batch-size 0 --num-workers 4 --sweet-spot 10 --total-iter 10

MaskGit					
Masked					