DLP_LAB5

> — 313551133 陳軒宇

1. Introduction (5%)

本次實驗的目的在實作 MaskGIT（Masked Generative Image Transformer）模型，並將其應用於圖像修復任務。MaskGIT 是一種基於 Transformer 的生成模型，能夠有效地處理圖像生成和修復問題。

本次實驗的主要目標包括：

1. 實作 Multi-Head Self-Attention
2. 實作 MaskGIT 模型：從頭開始訓練 Transformer 模型（MaskGIT 的第二階段）
3. 實作 iterative decoding：為圖像修復任務設計並實作迭代解碼過程，逐步填補缺失的圖像區域。
4. 探索不同的 mask scheduling functions：比較不同 mask scheduling functions 設置對修復結果的影響。

使用解析度為 64x64 的圖像數據集，並利用預訓練的 VQGAN（Vector Quantized Generative Adversarial Network）作為 MaskGIT 的第一階段，最後以 FID 評估修復結果。

2. Implementation Details (45%)

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

給定輸入序列 X， Multi-Head Self-Attention 的計算可以表示為：

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

其中，Q、K、V 是輸入 X 的線性變換：

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V$$

最終的 Multi-Head Self-Attention 輸出為：

$$\text{MultiHead}(X) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W_O$$

其中 $\text{head}_i = \text{Attention}(XW_{Q_i}, XW_{K_i}, XW_{V_i})$

```python
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  # 注意力頭的數量
        self.dim = dim  # 輸入的維度
        self.head_dim = dim // num_heads  # 每個頭的維度
        assert self.head_dim * num_heads == dim, "dim must be divisible by nu

        self.qkv = nn.Linear(dim, dim * 3) # (query, key, value) 的維度都是dim
        # 注意力dropout
        self.attn_drop = nn.Dropout(attn_drop)
        # 最終的輸出投影
        self.proj = nn.Linear(dim, dim)

    def forward(self, x):
        ''' Hint: input x tensor shape is (batch_size, num_image_tokens, dim)
            because the bidirectional transformer first will embed each toker
            and then pass to n_layers of encoders consist of Multi-Head Atter
            # of head set 16
            Total d_k , d_v set to 768
            d_k , d_v for one head will be 768//16.
        '''
        # x的形狀: (batch_size, num_tokens, dim)
        B, N, C = x.shape  # B: batch size, N: 序列長度, C: 維度

        # 生成Q、K、V並重塑
        qkv = self.qkv(x)  # shape: (B, N, 3*C)
        qkv = qkv.reshape(B, N, 3, self.num_heads, self.head_dim)  # shape:
        qkv = qkv.permute(2, 0, 3, 1, 4)  # shape: (3, B, num_heads, N, head_
        q, k, v = qkv[0], qkv[1], qkv[2]  # each shape: (B, num_heads, N, hea

        # 計算注意力分數
        attn = (q @ k.transpose(-2, -1))  # 形狀: (B, num_heads, N, N)
        attn = attn * (self.head_dim ** -0.5)  # 縮放注意力分數

        # 應用softmax使分數和為1
        attn = attn.softmax(dim=-1)

        # 應用dropout
        attn = self.attn_drop(attn)

        # 將注意力分數與值相乘
        x = (attn @ v)  # 形狀: (B, num_heads, N, head_dim)
        x = x.transpose(1, 2)  # 形狀: (B, N, num_heads, head_dim)
        x = x.reshape(B, N, C)  # 形狀: (B, N, dim)

        # 最後的線性變換
        x = self.proj(x)

        return x
```

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

在 Stage2 的訓練中，我們實現了 Masked Visual Token Modeling (MVTM) 策略，並設計了相應的前向傳播和損失計算方法。

1. MVTM 策略實現：

   - 在 `MaskGit` 類的 `forward` 方法中實現：

```
    def forward(self, x, ratio):
        z_indices = self.encode_to_z(x) # ground truth
        z_indices = z_indices.view(-1, self.num_image_tokens)
        mask = torch.bernoulli(torch.ones_like(z_indices) * ratio) # a
        z_indices_input = torch.where(mask == 1, torch.tensor(self.mas
        logits = self.transformer(z_indices_input) # transformer predi
        logits = logits[..., :self.mask_token_id]
        gt = torch.zeros(z_indices.shape[0], z_indices.shape[1], self.
        return logits, gt
```

   - 其中 `z_indices` 的 Encode 是透過 `encode_to_z` 方法實現的，來自 `VQGAN` 的 `encode` 方法：

```
     def encode_to_z(self, x):
         _, z_ind, _ = self.vqgan.encode(x)
         return z_ind
```

2. Mask scheduling functions

   - 在 `gamma_func` 方法中實現了三種掩碼調度策略：`linear`，`cosine`，`square`

```
def gamma_func(self, mode="cosine"):
    if mode == "linear":
        return lambda r: 1 - r
    elif mode == "cosine":
        return lambda r: math.cos(math.pi * r / 2)
    elif mode == "square":
        return lambda r: 1 - r ** 2
```

3. Loss Function

   - 使用 Cross Entropy Loss 計算預測結果和真實標籤之間的差異：

```
loss = F.cross_entropy(y_pred, y)
```

4. Optimizer

   - 使用 AdamW 優化器，以及 weight decay 的分組策略：

```
optim_groups = [
    {"params": [param_dict[pn] for pn in sorted(list(decay))], "weigh
    {"params": [param_dict[pn] for pn in sorted(list(no_decay))], "we
]
optimizer = torch.optim.AdamW(optim_groups, lr=self.learning_rate, be
```

5. Learning Rate Scheduler

   - 使用 LambdaLR 調度器實現 warmup 策略

```
scheduler = torch.optim.lr_scheduler.LambdaLR(optimizer, lambda steps
```

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

1. iterative decoding：在 `MaskGIT` 類的 `inpainting` 方法中實現：

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

    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)

        # raise Exception('TODO3 step1-1!')
        ratio = 0
        #iterative decoding for loop design
        #Hint: it's better to save original mask and the updated mask by
        for step in range(self.total_iter):
            if step == self.sweet_spot:
                break
            ratio = (step + 1) / self.total_iter #this should be updated

            z_indices_predict, mask_bc = self.model.inpainting(image, ra

            #static method yon can modify or not, make sure your visuali
            mask_i=mask_bc.view(1, 16, 16)
            mask_image = torch.ones(3, 16, 16)
            indices = torch.nonzero(mask_i, as_tuple=False)#label mask t
            mask_image[:, indices[:, 1], indices[:, 2]] = 0 #3,16,16
            maska[step]=mask_image
            shape=(1,16,16,256)
            z_q = self.model.vqgan.codebook.embedding(z_indices_predict)
            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 #get decoded image

            image = decoded_img # update image
```

2. 單次迭代的修復過程： `MaskGit` 模型中的 `inpainting` 方法中實現：

```
@torch.no_grad()
def inpainting(self, x , ratio, mask_b):
    z_indices = self.encode_to_z(x)
    z_indices_input = torch.where(mask_b == 1, torch.tensor(self.mask_tc
    logits = self.transformer(z_indices_input)
    #Apply softmax to convert logits into a probability distribution acr
    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(ratio)
    #predicted probabilities add temperature annealing gumbel noise as c
    g = -torch.log(-torch.log(torch.rand(1, device=z_indices_predict_prc
    temperature = self.choice_temperature * (1 - ratio)
    confidence = z_indices_predict_prob + temperature * g

    #hint: If mask is False, the probability should be set to infinity,
    #sort the confidence for the rank
    confidence = torch.where(mask_b == 0, torch.tensor(float('inf')).to(
    ratio = 0 if ratio < 1e-8 else ratio
    n = math.ceil(mask_b.sum() * ratio)
    _, idx_to_mask = torch.topk(confidence, n, largest=False)

    #define how much the iteration remain predicted tokens by mask sched
    #At the end of the decoding process, add back the original token val
    mask_bc = torch.zeros_like(mask_b).scatter_(1, idx_to_mask, 1)
    torch.bitwise_and(mask_bc, mask_b, out=mask_bc)

    return z_indices_predict, mask_bc
```

3. Discussion(bonus: 10%)

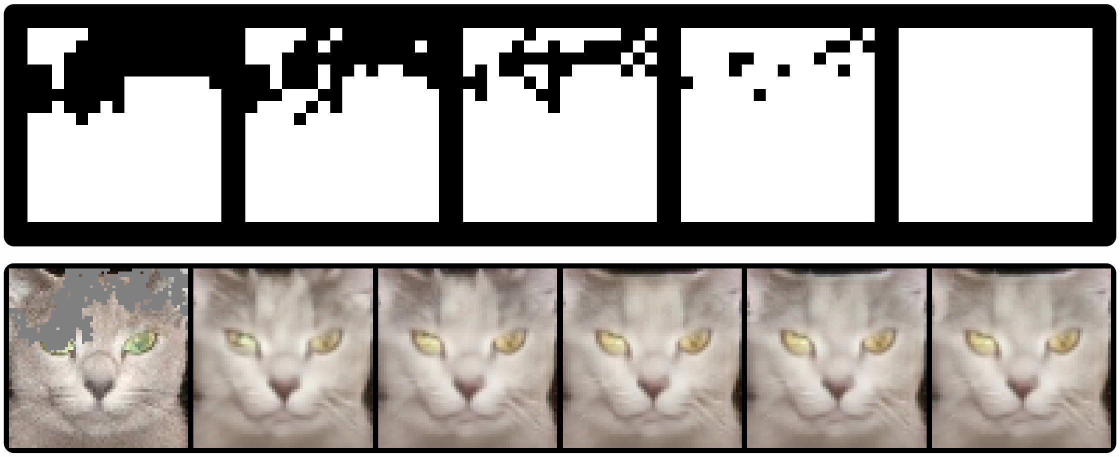
pass

4. Experiment Score (50%)

A. Experimental results (30%)

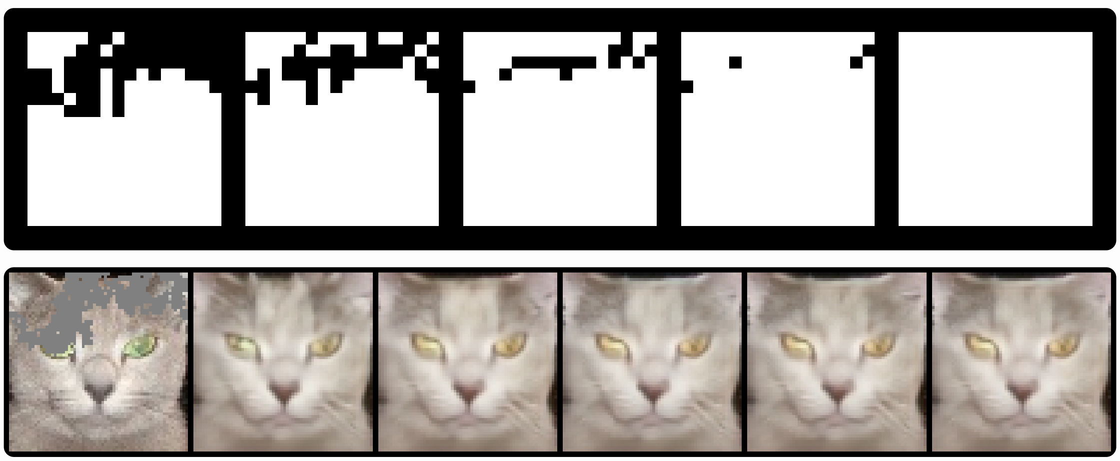
> show iterative decoding with different mask scheduling functions
>
> 1. Mask in latent domain
> 2. Predicted image

cosine



cosine

linear



linear

square

square

B. The Best FID Score(20%)

Screenshot

- FID: 31.340395367051542

```
[14] !python inpainting.py --total-iter 5 --sweet-spot 5 --mask-func cosine --output-path ./output/cosine_5_5 --load-transformer-ckpt-path /content/drive/MyDrive/DLP/Lab5/tran
     !python ./faster-pytorch-fid/fid_score_gpu.py --device cuda --predicted-path ./output/cosine_5_5/test_results --gtcsv-path ./faster-pytorch-fid/test_gt.csv

Inpainting: 747it [02:45,  4.51it/s]
747
100% 15/15 [00:03<00:00,  4.43it/s]
100% 15/15 [00:03<00:00,  4.82it/s]
FID:  31.167982574428237
```

Masked Images v.s MaskGITInpainting Results v.s Ground Truth

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

- Inpainting Parameters
  - mask_func: cosine
  - sweet_spot: 5
  - total_iter: 5
  - choice_temperature: 4.5 (default)
- Transformer Training Parameters
  - batch_size: 16
  - epochs: 50
  - learning_rate: 10^{-4}