Lab5 MaskGit Report ----------多工一 蘇柏叡

1. Introduction

於本Lab中，先是實現了Multi-Head Self-Attention機制，並且透過訓練transformer model 去預測latent token of images。至於在訓練好transformer model後，我們會接著採用iterative decoding方式對inpainting部分進行實作，以增加model對於圖片缺失、模糊區塊修復與填充之能力。最終，我們會將生成的資料和groundtruth以FID value做為評估指標，其中FID Value越低者即代表生成之資料與真實資料有較高的相似性，而在本實驗當中經過多種調試，最佳FID值為32.93(Cosine且sweet spot設10、Total Iteration設為10)且在指定的cosine、linear、square皆小於35，算是成功地達成本實驗設下之要求。

2. Implementation Details

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

TODO1實現了Multi-Head Self-Attention機制。在此機制中，首先對輸入進行三種線性變換，分別生成Key、Query和Value，這三者的維度（head_dim）均為 dim // num_heads，即每個head_dim（皆為768 / 16 = 48）。此外，在Multi-Head Self-Attention中亦設置了scale，這是基於每頭維度開根號再取倒數，目的是在計算時防止數值過大。在通過scaled dot-product attention計算得到注意力權重後，這些權重會被應用於Value。接著，通過轉置和重塑操作將輸出合併，並通過一個線性變換層對其進行最終的整合。此設計保證了輸入和輸出的形狀均為 (Batch_size, num_image_tokens, dim)，以維持資料的一致性和處理的連續性。

```python
#TODO1
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
        assert self.head_dim * num_heads == dim, "dim must be divisible by num_heads"

        self.scale = (self.head_dim ** -0.5)
        # Separate linear transformations for keys, queries, and values
        self.key = nn.Linear(dim, dim)
        self.query = nn.Linear(dim, dim)
        self.value = nn.Linear(dim, dim)

        self.attention_drop = nn.Dropout(attn_drop)
        self.project = nn.Linear(dim, dim)
```

```python
def forward(self, x):
    Batch_size, num_image_tokens, dim = x.shape

    # Perform linear operations and split heads
    keys = self.key(x).reshape(Batch_size, num_image_tokens, self.num_heads, self.head_dim).permute(0, 2, 1, 3)
    queries = self.query(x).reshape(Batch_size, num_image_tokens, self.num_heads, self.head_dim).permute(0, 2, 1, 3)
    values = self.value(x).reshape(Batch_size, num_image_tokens, self.num_heads, self.head_dim).permute(0, 2, 1, 3)

    # Scaled Dot-Product Attention
    attention_scores = torch.matmul(queries, keys.transpose(-2, -1)) * self.scale
    attention = F.softmax(attention_scores, dim=-1)
    attention = self.attention_drop(attention)

    # Apply attention to values
    out = torch.matmul(attention, values)
    out = out.transpose(1, 2).reshape(Batch_size, num_image_tokens, dim)

    # Apply final linear transformation
    return self.project(out)
```

B. The details of your stage2 training

1. MVTM

A. 首先完成encode_to_z function，此function是用來將輸入影像通過 VQGAN 模型進行編碼，並生成對應的 token，並後續的生成模型訓練。

```python
##TODO2 step1-1: input x to vqgan encoder to get the latent and zq value
    @torch.no_grad()
    def encode_to_z(self, x):
        latent, z_q, _ = self.vqgan.encode(x)
        return latent, z_q.reshape(latent.shape[0], -1)
```

B. 為撰寫需要使用的各種mask_scheduling，並且在inpainting時，使用use_gamma進行調用。

```python
def gamma_func(self, mode="cosine"):
    if mode == "linear":
        def mask_func(ratio):
            return 1 - ratio
        return mask_func
    elif mode == "cosine":
        def mask_func(ratio):
            return math.cos(math.pi * ratio / 2)
        return mask_func
    elif mode == "square":
        def mask_func(ratio):
            return 1 - ratio**2
    elif mode == "cubic":
        def mask_func(ratio):
            return 1 - ratio**3
        return mask_func
    else:
        raise NotImplementedError

def use_gamma(self, ratio):
    mask_ratio = self.gamma(ratio)
    return mask_ratio
```

C. 首先獲取 quantized tokens，並利用伯努利分布為每個 token 生成一個遮罩，其中每個元素的遮蓋機率設定為 0.5，這意味著每個 token 有 50% 的概率被選中進行遮蓋，並將遮罩選中的 tokens 替換為特定的遮罩標識符（mask token ID)。這些部分遮蓋的 tokens 隨後被送入 Transformer 模型進行進一步處理。最終，模型輸出 logit 分數以及原始的未遮蓋 tokens，以供訓練使用。

```
##TODO2 step1-3:
    def forward(self, x):
        # Step 1: Encode input to latent space and quantize
        _, z_indices = self.encode_to_z(x)  # Assume this returns quantized indices directly
        # Step 2: Create mask - 50% chance for each token to be masked
        mask = torch.bernoulli(torch.full(z_indices.shape, 0.5, device=z_indices.device)).bool()
        # Step 3: Apply mask - replace masked tokens with the mask token ID
        masked_indices = torch.where(mask, torch.full_like(z_indices, self.mask_token_id), z_indices)
        # Step 4: Pass masked indices through the transformer
        logits = self.transformer(masked_indices)
        # Return both the logits and the original indices (ground truth for training)
        return logits, z_indices
```

2. (forward&loss)

對於每個batch的數據，執行前向傳播以計算交叉熵損失來評估模型對於MVTM token遮罩後的預測能力，並進行反向傳播來計算梯度，並在指定的間隔（args.accum_grad set as 10）上執行優化器步驟來更新模型的權重。

```
    def train_one_epoch(self, train_loader, epoch, args):
        self.model.train()
        total_loss = 0
        total_batches = len(train_loader)
        progress = tqdm(enumerate(train_loader, start=1), total=total_batches, desc=f"Training Epoch {epoch}")

        for i, data in progress:
            data = data.to(args.device)
            self.optimizer.zero_grad()
            logits, targets = self.model(data)
            loss = F.cross_entropy(logits.view(-1, logits.size(-1)), targets.view(-1))
            loss.backward()
            if i % args.accum_grad == 0:
                self.optimizer.step()
                self.optimizer.zero_grad()

            total_loss += loss.item()
            running_avg_loss = total_loss / i
            progress.set_description(f"Training Epoch {epoch} - Step {i}/{total_batches}: Running Avg Loss = {running_avg_loss:.4f}")

        avg_loss = total_loss / total_batches
        return avg_loss
```

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

在 inpainting 函數中masked_indices和 non_masked_indices被送入 Transformer 模型進行處理，模型輸出預測每個 token 的 logits。這些 logits 被用來計算每個可能 token 的機率分佈，並通過 softmax 函數進行normalize。接下來，透過Gumbel分布進行抽樣，這個隨機性的引入旨在模擬實際操作中的不確定性，並嘗試找到最優的遮罩策略。透過對置信度分數的排序和cutoff的計算，確定哪些token將在下一次迭代中被保留。使用由遮罩比例驅動的動態調整溫度參數，來確定哪些遮罩的tokens將在下一次迭代過程中被展示。這個過程涉及到對概率進行排序並根據 Gumbel 抽樣修改的結果來動態調整遮罩，從而漸進地優化模型對隱藏部分的預測精度。

```python
@torch.no_grad()
def inpainting(self, z_indices, mask, mask_num, ratio):
    masked_indices = mask.nonzero(as_tuple=True)
    non_masked_indices = (~mask).nonzero(as_tuple=True)

    # create a new tensor to store the indices with mask
    z_indices_with_mask = z_indices.clone()
    z_indices_with_mask[masked_indices] = self.mask_token_id
    z_indices_with_mask[non_masked_indices] = z_indices[non_masked_indices]

    # run the transformer model to get the logits
    logits = self.transformer(z_indices_with_mask)
    probs = torch.softmax(logits, dim=-1)
    # make sure the predict token is not mask token
    z_indices_predict = torch.distributions.categorical.Categorical(logits=logits).sample()
    while torch.any(z_indices_predict == self.mask_token_id):
        z_indices_predict = torch.distributions.categorical.Categorical(logits=logits).sample()

    z_indices_predict = mask * z_indices_predict + (~mask) * z_indices
    # to get prob from predict z_indices
    z_indices_predict_prob = probs.gather(-1, z_indices_predict.unsqueeze(-1)).squeeze(-1)
    z_indices_predict_prob = torch.where(mask, z_indices_predict_prob, torch.zeros_like(z_indices_predict_prob) + torch.inf)
```

```python
    mask_ratio = MaskGit.use_gamma(self, ratio)
    mask_len = torch.floor(mask_num * mask_ratio).long()
    '''
    First, create a Gumbel distribution and sample from it. Then calculate
    the adjusted temperature and use it to compute the confidence scores.
    Sort these scores and find the cutoff value. Finally, update the mask
    based on this cutoff value. This process involves thresholding the confidence scores
    to determine which elements should be masked.
    '''
    g_samples = torch.distributions.Gumbel(0, 1).sample(z_indices_predict_prob.shape).to(z_indices_predict_prob.device)
    temp_adjusted = self.choice_temperature * (1 - mask_ratio)
    confidence_scores = z_indices_predict_prob + temp_adjusted * g_samples

    _, sorted_indices = torch.sort(confidence_scores, dim=-1)
    cutoff_value = confidence_scores.gather(1, sorted_indices[:, mask_len].unsqueeze(-1))

    new_mask = (confidence_scores < cutoff_value)
    return z_indices_predict, new_mask
```

Bonus 的 Discussion 置於Exp部分

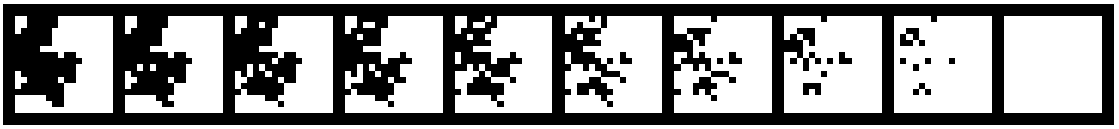
Experiment

1. Prove your code implementation is correct

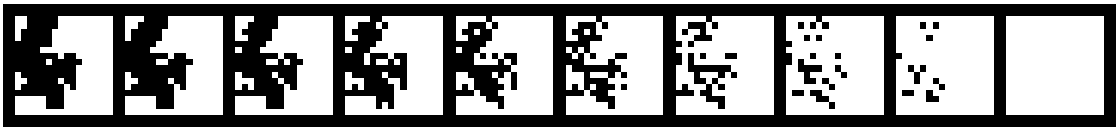
A.Show iterative decoding.

(a)Mask in latent domain (take first picture to represent)

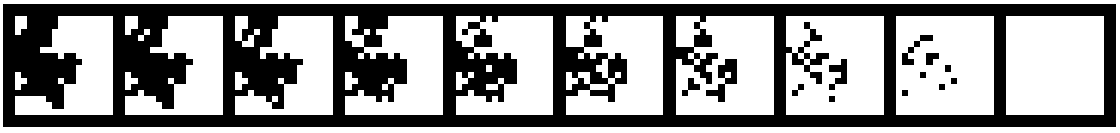
i. cosine



ii. linear



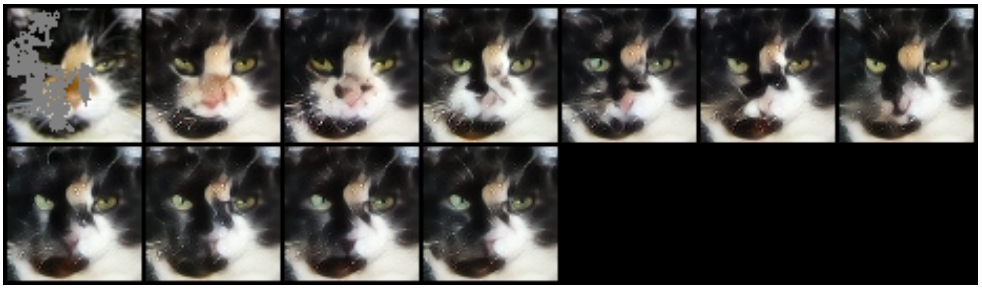
iii. square



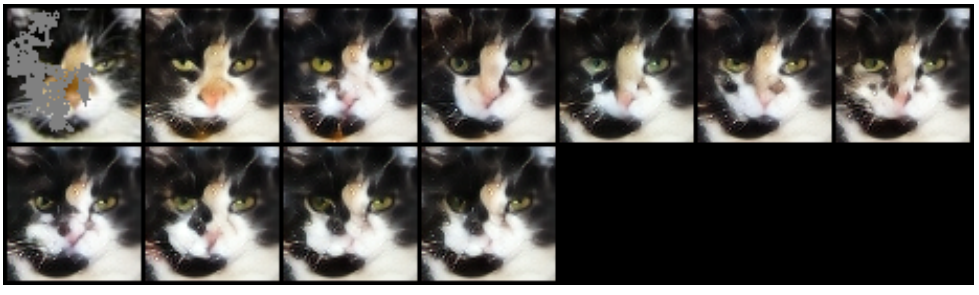
(b)Predicted image (take first picture to represent)

i. cosine

ii. linear



iii. square



2. The Best FID Score

A. Experiment setting and reference

a. LR = 1E-4、Criteria in training: Cross Entropy、Sweet Spot = 10、Iteration = 10

```
parser.add_argument('--sweet-spot', type=int, default=10, help='sweet spot: the best step in total iteration')
parser.add_argument('--total-iter', type=int, default=10, help='total step for mask scheduling')
```

b. 因時間較不充裕，僅設定Sweet Spot = 10(取8、12的平均)、Iteration = 10並且執行四種方式(包含Cubic)，並從中挑選出較佳的組合。

> get less concave. Among functions that achieve strong FIDs (*i.e.* cosine, square, and linear), cosine not only has the strongest overall score, but also the earliest sweet spot at a total of 8 to 12 iterations. We hypothesize that such sweet

c. 因為設備+時間因素(平均一個Epoch含驗證需要27mins+)，故Transformer僅有訓練46Epoch。

B. Screenshot

a.訓練過程截圖：

Training Loss大約從2.2開始降，到第46Epoch時則約為1.46，而Val_Loss大約從1.9開始降，到第46Epoch時則約為1.46，且兩者皆有持續下降之趨勢，在未來若設備、時間允許，則是可以持續訓練以求能有更好之效果。

```
Epoch 43/300 - Validation Loss: 1.4739
New best training loss. Saving model...
Training Epoch 44 - Step 3000/3000: Running Avg Loss = 1.4665: 100%|
| 3000/3000 [23:57<00:00,
 2.09it/s]
Epoch 44/300 - Training Loss: 1.4665
Evaluating Epoch 44 - Step 750/750: Running Avg Loss = 1.4631: 100%|
| 750/750 [03:12<00:00,
 3.90it/s]
Epoch 44/300 - Validation Loss: 1.4631
New best training loss. Saving model...
New best validation loss. Saving model...
Training Epoch 45 - Step 3000/3000: Running Avg Loss = 1.4661: 100%|
| 3000/3000 [23:59<00:00,
 2.08it/s]
Epoch 45/300 - Training Loss: 1.4661
Evaluating Epoch 45 - Step 750/750: Running Avg Loss = 1.4602: 100%|
| 750/750 [03:12<00:00,
 3.89it/s]
Epoch 45/300 - Validation Loss: 1.4602
New best training loss. Saving model...
New best validation loss. Saving model...
Training Epoch 46 - Step 3000/3000: Running Avg Loss = 1.4633: 100%|
| 3000/3000 [24:05<00:00,
 2.08it/s]
Epoch 46/300 - Training Loss: 1.4633
Evaluating Epoch 46 - Step 750/750: Running Avg Loss = 1.4620: 100%|
| 750/750 [03:17<00:00,
 3.79it/s]
Epoch 46/300 - Validation Loss: 1.4620
New best training loss. Saving model...
```

b. 設定四種方式皆採用sweet spot = 10、total iter = 10，其中cosine表現較優，

此為採用Cosine方式後所得之FID值為32.93(sweet spot = 10、total iter = 10)

```
Begin Inpainting...
PS C:\Users\蘇柏叡\Desktop\Lab5 code\lab5> cd faster-pytorc
PS C:\Users\蘇柏叡\Desktop\Lab5 code\lab5\faster-pytorch-fi
747
100%|
100%|
FID:  32.934284326534794
```

此為採用Cosine方式後所得之FID值為33.18(sweet spot = 10、total iter = 20)

```
PS C:\Users\蘇柏叡\Desktop\Lab5 code\lab5\faster-pytorch
core_gpu.py"
747
100%|
100%|
FID:  33.184750064628986
PS C:\Users\蘇柏叡\Desktop\Lab5 code\lab5\faster-pytorch
```

c.各方法之FID Value(sweet spot = 10、total iter = 10)

FID \ Method	Cosine	Linear	Square	Cubic
FID value:	32.93	34.01	34.00	34.52

從此表可看出，如同文獻所呈現，Cosine的表現確實是較優於其它者的，另外，也有實作Cubic方式，不過效果亦沒有指定三種來的好。

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

Masked Images

MaskGIT Inpainting Results

Ground Truth

C. Discussion (Include anything I wanna say)

1.比較不同Sweet Spot(以Cosine為示範)將分別以SweetSpot = 8、10、12進行比較以(Sweet Spot,Total Iter)作呈現

	(8,8)	(10,10)	(12,12)	(8,20)	(10,20)	(12,20)
FID	33.82	32.93	32.97	34.23	33.18	34.62

由此表可看出，當sweet spot 設為10時可能會具有較佳之FID值

2. 結論:

本Lab訓練時雖因設備因素而無法訓練更多Epoch以優化生成效果(中止於46Epoch)，且如同先前提到不論是Train、Val的Loss皆有穩定下降趨勢，故尚未達到Overfitting的問題，且此時兩者Loss值落差極小，故也無Under fitting之情況。經過測試，不論是自行增加implement的cubic也好，以及文獻中提到Sweet Spot設在8-12區間較佳...等，皆可以有效地將FID值壓低在35之下，且大多數生成的圖片也能夠有效還原，雖偶爾會有部分是較模糊或是明顯沒生成好的圖片如下圖在眼睛部分有明顯的不合理，不過整體而言算是有達成本次Lab目標(意即FID值40以下)，若是未來設備、時間允許則可以嘗試訓練200、300Epoch並且加入Early Stopping避免Overfitting，應該是有機會可以提升生成之效果與表現。