Lab 3 MaskGIT for Image Inpainting

314551022 廖洺玄

Introduction

在這個 Lab 中，我們的主要任務是訓練 VQGAN 的 transformer，並應用於 MaskGIT 雙向 decoder 中。我們用了三種不同的 mask schedulers，分別在不同 mask ratio 漸進，探索生成品質與收斂穩定性之間的關係。在這個 lab 我也利用 LLM 來教我 MaskGIT 原論文中提到的雙向解碼過程，包括如何從 token logit 中挑選 high-confidence token 進行替換、如何保留 uncertain 部分繼續 mask、以及如何透過 softmax logits 計算 top-k uncertainty 並據此調整 mask ratio。在實作與調整 decoder 時，我遇到了一些 debug，例如 tensor 維度不一、token index 解碼順序錯誤等，我也透過 LLM decode my code。我調整 mask function 的方式是去改 yaml 的設定。

Implementation Details

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

MyScaleDotProductAttention

在這裡我 implement 了一個手刻的 scale dot product funtion 用來給 multi-head self-attention 做查詢 distance 然後將 distance 轉乘 attention score 讓 model 可以做到對某個部分 attention。他會首先做 q 和 k^T 的 multiplication，然後做 scale 去避免發生 gradient vanishing 的問題，接著做 SoftMax 讓每個 query 的 key value 可以變成機率分布的狀態，然後丟棄 10%的 attention 防止 overfitting，最後在做架構圖上最後一個 component，我們計算過來的結果和 v 做 multiplication。

```python
def MyScaleDotProductAttention(q, k, v, attn_drop=0.1):
    '''
    q, k, v: (batch_size, num_image_tokens, dim)
    '''
    d_k = q.size(-1)
    attention_scores = torch.matmul(q, k.transpose(-2, -1))
    scale = attention_scores / math.sqrt(d_k) # average the dot product,
    softmax = torch.softmax(scale, dim=-1) # to make the key distributio
    attn = nn.Dropout(attn_drop)(softmax)
    output = torch.matmul(attn, v)
    return output
```

Multi-Head Self-Attention

在這裡用了一個 Multi-Head Attention module 用來讓 bert 可以 parallel 地關注不同 subspace 的資訊。他會首先透過 qkv_proj 這個 linear layer 同時產生 query、key、value 三個 matrix，然後將結果 reshape 成 16 個 attention head 的格式讓每個 head 專注於不同的 representation subspace，接著 q、k、v 餵進我手刻的 scaled dot-product 計算 attention score，最後透過 transpose 和 reshape 將 multi-head 的結果 concat 起來並通過 output projection。這個 muti head attention 讓 model 可以同時捕捉 token 之間在不同語義層面的關聯性，提升 MaskGIT 在 bidirectional context modeling 的能力

```python
class MultiHeadAttention(nn.Module):
    def __init__(self, dim=768, num_heads=16, attn_drop=0.1):
        super(MultiHeadAttention, self).__init__()
        # my implementation of Multi-Head Attention
        self.num_heads = num_heads
        self.dim = dim
        self.d_k = dim // num_heads
        self.d_v = dim // num_heads
        self.qkv_proj = nn.Linear(dim, dim * 3)
        self.attn_drop = attn_drop
        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 token to dim dimension,
            and then pass to n_layers of encoders consist of Multi-Head Attention and MLP.
            # of head set 16
            Total d_k , d_v set to 768
            d_k , d_v for one head will be 768//16.
        '''
        batch_size, num_tokens, _ = x.size()
        qkv = self.qkv_proj(x).view(batch_size, num_tokens, self.num_heads, 3 * self.d_k).transpose(1, 2)
        q, k, v = qkv.chunk(3, dim=-1)  # q, k, v = q.contiguous(), k.contiguous(), v.contiguous()  # split i
        attn = MyScaleDotProductAttention(q, k, v, self.attn_drop)
        attn = attn.transpose(1, 2).contiguous().view(batch_size, num_tokens, self.dim)
        output = self.proj(attn)
        return output
```

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

在這裡我 implement 了 encode_to_z function 用來將 input 透過預訓練好的 VQGAN 進行 VQ 處理。他會首先將 image 轉換成 continuous latent representation，然後透過 codebook mapping 將每個位置對應到最接近的 discrete token index，接著回傳 latent (discrete token indices)和 zq (quantized features)這兩個 representation。這個 quantization 過程在 discrete token space 中進行 MaskGIT 的 Bert training 為了從連續空間轉換到離散空間

```python
##TODO2 step1-1: input x fed to vqgan encoder to get the latent and zq
    @torch.no_grad()
    def encode_to_z(self, x):
        # z_hat = E(x), zq = q(z_hat) = argmin_{z} ||z_hat(i, j) - z(k)||^2 = codebook_mapping
        zq, latent, q_loss = self.vqgan.encode(x) # 從VQGAN encoder會return codebook_mapping, codebook_indices, q_loss
        # zq = codebook_mapping (quantized features)
        # latent = codebook_indices (discrete token indices)
        return latent, zq
```

輸入的 ratio 在 0 到 1 之間的 decoding 進度回傳對應的遮罩比率(mask rate)，用來動態控制每次解碼時需 mask 掉多少 token。

1. linear：直接線性遞減遮罩比例，公式是 1 - ratio，代表解碼剛開始時完全遮罩(1)，慢慢均勻地釋放資訊到完全沒有遮罩(0)。
2. cosine：使用餘弦函數 (cos(pi * ratio) + 1) / 2，初期遮罩比率接近 1，後期加速減少遮罩，比線性更平滑且符合 MaskGIT 論文中凹型(Concave)最佳策略。
3. square：使用平方遞減，1 - ratio^2，在開始時保持較多遮罩，後期加速降低遮罩是另一種凹型函數。

```python
##TODO2 step1-2:
    def gamma_func(self, mode="cosine"):
        """Generates a mask rate by scheduling mask functions R.

        Given a ratio in [0, 1), we generate a masking ratio from (0, 1].
        During training, the input ratio is uniformly sampled;
        during inference, the input ratio is based on the step number divided by the total iterat
        Based on experiements, we find that masking more in training helps.

        ratio:   The uniformly sampled ratio [0, 1) as input.
        Returns: The mask rate (float).

        """
        if mode == "linear":
            return lambda ratio: 1 - ratio
        elif mode == "cosine":
            return lambda ratio: (math.cos(math.pi * ratio) + 1.0) / 2.0
        elif mode == "square":
            return lambda ratio: 1 - ratio ** 2
        else:
            raise NotImplementedError
```

我的 forward 函數先用 encoder 將輸入圖像(B,3,H,W)編碼成離散 token index (B,num_image_tokens)，接著為每個 batch 樣本生成隨機 ratio 並通過 gamma 函數計算出不同的遮罩比例，創建隨機布林遮罩將選中位置的 token 替換為 mask_token_id，最後將遮罩後的 token sequence 輸入 transformer 預測被遮罩位置的原始 token 並返回預測 logits 和遮罩後的 token sequence。

```python
##TODO2 step1-3:
    def forward(self, x):
        # x: input image, shape (B, 3, H, W)
        # encode x to zq and latent
        _, z_indices = self.encode_to_z(x)  # z_indices: shape could be (B, H, W) -> flatten to (B, num_image_tokens)

        # Ensure z_indices is properly flattened to (B, num_image_tokens) and is long type
        if len(z_indices.shape) > 2:
            z_indices = z_indices.flatten(1)
        z_indices = z_indices.long()

        if z_indices.shape[1] != self.num_image_tokens:
            z_indices = z_indices[:, :self.num_image_tokens]
        z_indices = torch.clamp(z_indices, 0, self.mask_token_id - 1)

        # get mask schedule
        ratio = torch.rand(z_indices.shape[0], device=z_indices.device)  # random batch ratios
        # create mask for z_indices
        # Apply gamma function to each ratio and expand to match z_indices shape
        mask_ratios = torch.tensor([self.gamma(r.item()) for r in ratio], device=z_indices.device)  # shape (B,)
        mask_ratios = mask_ratios.unsqueeze(1).expand(-1, z_indices.shape[1])  # shape (B, num_image_tokens)
        # create a boolean mask based on the gamma function
        mask = torch.rand_like(z_indices, dtype=torch.float) < mask_ratios  # mask: shape (B, num_image_tokens), bool
        z_indices[mask] = self.mask_token_id  # replace masked tokens with mask token id
        # z_indices: shape (B, num_image_tokens), with masked tokens replaced by mask_token_id
        # feed z_indices to transformer
        logits = self.transformer(z_indices)  # logits: shape (B, num_image_tokens, num_codebook_vectors)
        # Return raw logits for loss calculation (softmax will be applied in cross_entropy)
        return logits, z_indices
```

在 Train one epoch 這裡我會將 MaskGit 模型設為訓練模式，啟用 dropout 和 batch normalization init loss 和 batch counter 然後把 gradient 清空後就進入 batch loop，會去把 input(batch_size, 3, H, W) load 到 model 上，然後 call 剛剛我們寫的 forword 去拿 encoder 將圖像編碼成離散 token，然後調 mask，最後 apply 到 transformer 上。接下來就去重新 encode 原始圖像得到 ground truth 的 token index，接下來就是用 Cross entropy 計算 loss 用被遮罩位置的預測 logits 和被遮罩位置的真實 token label。然後就 backpropagation、gradient and loss cumulation

```python
    def train_one_epoch(self, train_loader):
        self.model.train()
        total_loss = 0.0
        num_batches = 0

        for batch_idx, x in enumerate(tqdm(train_loader, desc="Training")):
            x = x.to(self.args.device)

            # Forward pass
            logits, z_indices_masked = self.model(x)

            # Get original z_indices (ground truth)
            with torch.no_grad():
                z_indices_gt, _ = self.model.encode_to_z(x)
                # Apply same preprocessing as in forward pass
                original_batch_size = x.shape[0]
                if len(z_indices_gt.shape) > 2:
                    z_indices_gt = z_indices_gt.flatten(1)
                elif len(z_indices_gt.shape) == 1:
                    # If 1D, reshape to proper batch size
                    z_indices_gt = z_indices_gt.view(original_batch_size, -1)
                z_indices_gt = z_indices_gt.long()
                if len(z_indices_gt.shape) > 1 and z_indices_gt.shape[1] != self.model.num_image_to
                    z_indices_gt = z_indices_gt[:, :self.model.num_image_tokens]
                z_indices_gt = torch.clamp(z_indices_gt, 0, self.model.mask_token_id - 1)

            # Calculate cross entropy loss only on masked positions
            mask = (z_indices_masked == self.model.mask_token_id)
            if mask.sum() > 0:
                loss = F.cross_entropy(
                    logits[mask].view(-1, logits.size(-1)),
                      z_indices_gt[mask].view(-1)
                  )
              else:
                  loss = torch.tensor(0.0, device=self.args.device, requires_grad=True)

              # Backward pass
              loss.backward()

              # Gradient accumulation
              if (batch_idx + 1) % self.args.accum_grad == 0:
                  self.optim.step()
                  self.optim.zero_grad()

              total_loss += loss.item()
              num_batches += 1

          # Handle remaining gradients
          if num_batches % self.args.accum_grad != 0:
              self.optim.step()
              self.optim.zero_grad()

          avg_loss = total_loss / num_batches if num_batches > 0 else 0.0
          return avg_loss

```

我的 eval_one_epoch 會將 model 設為評估模式並關閉梯度計算，對驗證集的每個 batch 執行與訓練相同的前向傳播流程（圖像編碼→遮罩→transformer 預測），獲取 ground truth token 並只在被遮罩位置計算 cross entropy 損失，累積所有 batch 的損失並返回平均驗證損失用於 monitor 模型表現。以及 optimizer and schedular 用 Adam (lr=1e-4, betas=(0.9, 0.999))是因為 Adam 結合 momentum 自適應學習率適合訓練 transformer，lr 設定保守避免訓練 loss epoch curve 不穩定，且只優化 transformer 參數因為 VQGAN 已

經預訓練 frozen；搭配 CosineAnnealingLR scheduler(T_max=50, eta_min=1e-6)平滑的 lr 避免震盪

```python
    def eval_one_epoch(self, val_loader):
        self.model.eval()
        total_loss = 0.0
        num_batches = 0

        with torch.no_grad():
            for x in tqdm(val_loader, desc="Validation"):
                x = x.to(self.args.device)

                # Forward pass
                logits, z_indices_masked = self.model(x)

                # Get original z_indices (ground truth)
                z_indices_gt, _ = self.model.encode_to_z(x)
                # Apply same preprocessing as in forward pass
                original_batch_size = x.shape[0]
                if len(z_indices_gt.shape) > 2:
                    z_indices_gt = z_indices_gt.flatten(1)
                elif len(z_indices_gt.shape) == 1:
                    # If 1D, reshape to proper batch size
                    z_indices_gt = z_indices_gt.view(original_batch_size, -1)
                z_indices_gt = z_indices_gt.long()
                if len(z_indices_gt.shape) > 1 and z_indices_gt.shape[1] != self.model.num_image
                    z_indices_gt = z_indices_gt[:, :self.model.num_image_tokens]
                z_indices_gt = torch.clamp(z_indices_gt, 0, self.model.mask_token_id - 1)

                # Calculate cross entropy loss only on masked positions
                mask = (z_indices_masked == self.model.mask_token_id)
                if mask.sum() > 0:
                    loss = F.cross_entropy(
                        logits[mask].view(-1, logits.size(-1)),
                      z_indices_gt[mask].view(-1)
                    )
                else:
                    loss = torch.tensor(0.0, device=self.args.device)

                total_loss += loss.item()
                num_batches += 1

        avg_loss = total_loss / num_batches if num_batches > 0 else 0.0
        return avg_loss

    def configure_optimizers(self):
        lr = self.args.learning_rate if hasattr(self.args, 'learning_rate') else 1e-4
        optimizer = torch.optim.Adam(self.model.transformer.parameters(), lr=lr, betas=(0.9, 0.999))
        scheduler = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max=50, eta_min=1e-6)
        return optimizer, scheduler
```

Step 5 在寫拿 arg 和 training 的過程跑到 for loop 會開始用 train one epoch function 去 train model 然後驗證跟計算 loss 更新 model weight 印出結果、存 check point 後再跑到下個 loop

```python
#TODO2 step1-5:
    print(f"\nEpoch {epoch}/{args.epochs}")

    # Training phase
    train_loss = train_transformer.train_one_epoch(train_loader)
    print(f"Train Loss: {train_loss:.4f}")

    # Validation phase
    val_loss = train_transformer.eval_one_epoch(val_loader)
    print(f"Val Loss: {val_loss:.4f}")

    # Update learning rate scheduler
    train_transformer.scheduler.step()

    # Save checkpoint
    if epoch % args.save_per_epoch == 0:
        checkpoint = {
            'epoch': epoch,
            'model_state_dict': train_transformer.model.transformer.state_dict(),
            'optimizer_state_dict': train_transformer.optim.state_dict(),
            'scheduler_state_dict': train_transformer.scheduler.state_dict(),
            'train_loss': train_loss,
            'val_loss': val_loss,
        }
        torch.save(checkpoint, f"transformer_checkpoints/epoch_{epoch}.pt")
        print(f"Checkpoint saved at epoch {epoch}")

    # Save latest checkpoint
    latest_checkpoint = {
        'epoch': epoch,
        'model_state_dict': train_transformer.model.transformer.state_dict(),
        'optimizer_state_dict': train_transformer.optim.state_dict(),
        'scheduler_state_dict': train_transformer.scheduler.state_dict(),
        'train_loss': train_loss,
        'val_loss': val_loss,
    }
    torch.save(latest_checkpoint, "transformer_checkpoints/latest.pt")
```

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

在這裡我會先做 masking schedule 計算，接下來 run 雙向 transformer 去對當前的 token 預測 logits，無論位置是 mask 還是非 mask，都會得到預測，然後 softmax 成機率分布後加入 temperature 是為了讓分布更平滑或更尖銳，根據 softmax 分布 隨機抽樣一個 token 當作 confidence 分數。num_to_mask = int(mask_ratio * flat_confidence.numel())在這裡計算我們這個 itr 要遮掉多少 pixel 後， sorted_candidates = candidates[torch.argsort(flat_confidence[candidates])]

indices_to_mask = sorted_candidates[:num_to_mask] 從所有 token 中選出信心最低的 token，要把它們再 mask 起來。把要遮起來的位置換回 [MASK] token；其他則保留預測值。

```python
##TODO3 step1-1: define one iteration decoding
@torch.no_grad()
def inpainting(self, step, total_iter):

    # 計算當前步驟的 mask ratio
    ratio = step / total_iter
    mask_ratio = self.gamma(ratio)
    logits = self.transformer(self.z_indices_predict)

    prob_logits = torch.softmax(logits / self.choice_temperature, dim=-1)
    confidence, z_indices_predict = torch.max(prob_logits, dim=-1)

    # 選 confidence 最低的 tokens 進行 mask
    flat_confidence = confidence.view(-1)
    current_mask = self.mask_bc.view(-1)
    candidates = current_mask.nonzero(as_tuple=True)[0]  # 還沒確定的 token 位置

    num_to_mask = int(mask_ratio * flat_confidence.numel())
    sorted_candidates = candidates[torch.argsort(flat_confidence[candidates])]
    indices_to_mask = sorted_candidates[:num_to_mask]

    # 更新maks
    new_mask = self.mask_bc.clone()
    new_mask.view(-1)[...] = False
    new_mask.view(-1)[indices_to_mask] = True

    # 在masked區域填入 mask token；其他位置填入預測 token
    z_indices_predict = torch.where(
        new_mask,  # 需要繼續預測的
        self.mask_token_id,
        z_indices_predict
    )

    # 保留原始圖像中不需要修復的區域
    z_indices_predict = torch.where(
        ~self.original_mask,  # 非 inpaint 區域，保留原圖 token
        self.original_z_indices,
        z_indices_predict
    )

    self.z_indices_predict = z_indices_predict
    self.mask_bc = new_mask
    return z_indices_predict, new_mask
```

Discussion

在這個 Lab 裡面我還額外發現了 Iteration 和 Sweet spot 太大或太小都會造成 FID 下降，太小會造成 MaskGIT 還沒找到足夠 confident 的 token 時就已經停止了，所以解果圖片會有很多 mask pixel，而太大則會造成 overfit，模型過度預測讓 FID score 反而下降。然後我發現 cosine 的收斂速度會比 linear 和 squeare 快很多，cosine 我只需要 18~20、linear 則要到 24~25 而 square 還需要到 30~32 左右。

Experiment Score

Part1: Prove your code implementation is correct

Show iterative decoding

Cosine

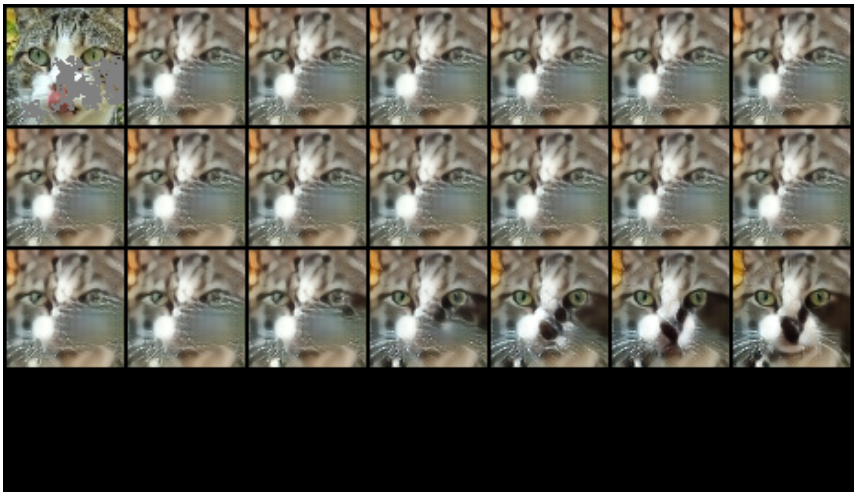
Mask in latent domain

Predicted image

Linear

Mask in latent domain

Predicted image



square

Mask in latent domain

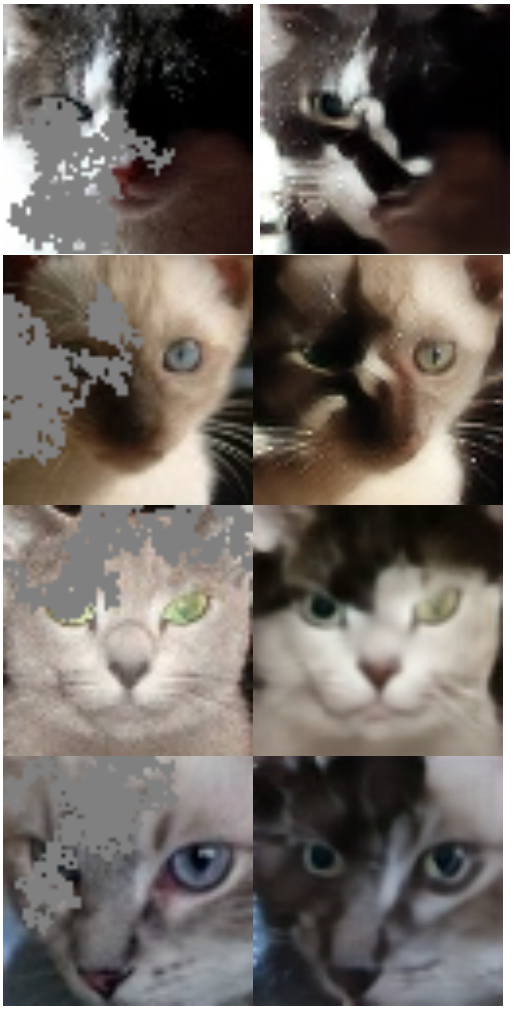
Predicted image

Part2: The Best FID Score

Screenshot

```
Running inpainting...
Using seed: 88414
Calculating FID...
100%|                                                                                | 15/15 [00:01<00:00,  8.07it/s]
100%|                                                                                | 15/15 [00:01<00:00,  8.86it/s]
747
FID:  49.86409195125711
Current FID: 49.86409195125711
FID (49.86409195125711) is below 50. Done.
(maskgit) root@jeff-MS-7D42:~/Lab3# ./run_until_fid.sh
```

Masked Images v.s MaskGIT Inpainting Results



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

目前我有找到最好的 FID 是 49.864 我用我訓練 50 epoch 的 VQGAN model 和 itr: 22、sweet-spot: 20(已經寫好在 default 了)，還有我用的 seed 是 88414，也有寫在 inference 讓他固定調整。