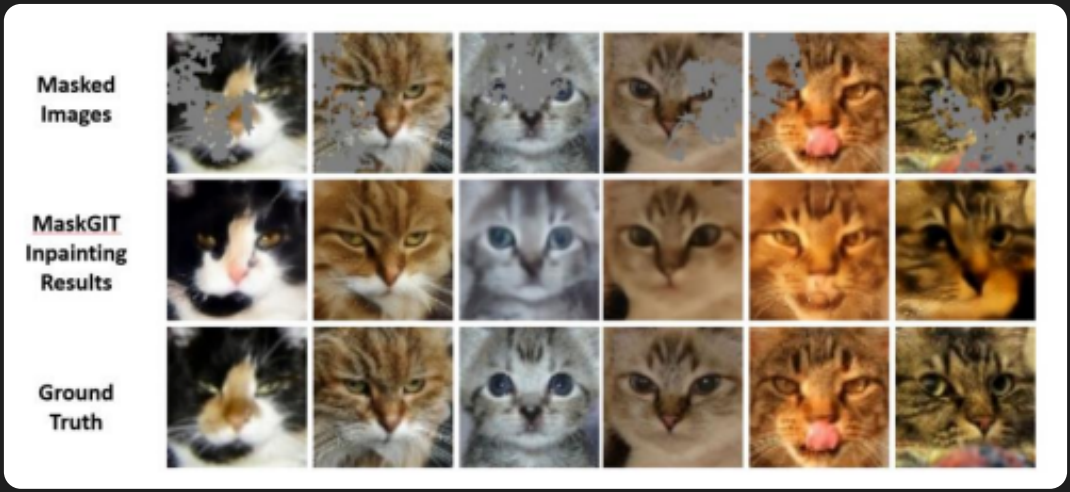
Lab5 MaskGIT for Image Inpainting - Report

313551055 柯柏旭

1. Introduction

這份報告探討了使用MaskGIT模型進行圖像修復的過程與效果。報告中詳細描述了模型的實作細節、及最後 `3. Discussion` 訓練過程中的挑戰與優化策略，以及最終結果的分析。透過調整訓練參數，如迭代次數、學習率及批次大小，作者成功將Fid值從58降至37，顯示了模型性能的顯著提升 。

總的來說，測試時，圖像中會有灰色區域表示缺失的信息，我們使用 MaskGIT 來恢復這些部分。本實驗的重點在於多頭注意力機制、Transformer 訓練和推論修復。此外，我們還可以嘗試不同的遮罩調度參數設定，以比較它們對修復結果的影響。



2. Implementation Details

A. The detail of model (Multi-head self-attention)

完整的 MultiHeadAttention 程式碼:

```
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.d_k = dim // num_heads
        self.d_v = dim // num_heads

        # Query, Key, Value linear layers
        self.query = nn.Linear(dim, dim)
        self.key = nn.Linear(dim, dim)
        self.value = nn.Linear(dim, dim)

        # Dropout layer
        self.attn_drop = nn.Dropout(attn_drop)
        self.proj = nn.Linear(dim, dim)
        self.proj_drop = nn.Dropout(attn_drop)

    def forward(self, x):
        # print(x.size())
        batch_size, num_tokens, dim = x.size()
        # print(batch_size, num_tokens, dim)

        # Ensure the input dimension is correct
        assert dim == self.dim, "Input dimension must match model
dimension"

        # Linear projections for query, key, value
        q = self.query(x).view(batch_size, num_tokens,
self.num_heads, self.d_k).transpose(1, 2)
        k = self.key(x).view(batch_size, num_tokens, self.num_heads,
self.d_k).transpose(1, 2)
        v = self.value(x).view(batch_size, num_tokens,
self.num_heads, self.d_v).transpose(1, 2)

        # Scaled Dot-Product Attention
        attn_scores = torch.matmul(q, k.transpose(-2, -1)) /
math.sqrt(self.d_k)
        attn_probs = torch.softmax(attn_scores, dim=-1)
        attn_probs = self.attn_drop(attn_probs)

        # Weighted sum of values
```

```
        context = torch.matmul(attn_probs, v).transpose(1,
2).contiguous().view(batch_size, num_tokens, dim)
        output = self.proj(context)
        output = self.proj_drop(output)

        return output
```

以下將經過程式片段逐一說明:

```
        self.num_heads = num_heads
        self.dim = dim
        self.d_k = dim // num_heads
        self.d_v = dim // num_heads
```

- `num_heads` 是 Multi head attention 的頭數，表示注意力機制會平行計算多少個不同的注意力分數。
- `dim` 是輸入向量的維度，d_k 和 d_v 是每個注意力頭的查詢向量（Query）和鍵值向量（Key/Value）的維度，通常等於 dim 除以 num_heads。

```
        self.query = nn.Linear(dim, dim)
        self.key = nn.Linear(dim, dim)
        self.value = nn.Linear(dim, dim)
```

- `query`, `key`, `value` 是用於計算查詢、鍵和值的線性層（全連接層），它們將輸入的向量投影到新的空間中，這些新空間將用來計算注意力分數。

```
        self.attn_drop = nn.Dropout(attn_drop)
        self.proj = nn.Linear(dim, dim)
        self.proj_drop = nn.Dropout(attn_drop)
```

- `attn_drop` 是應用於注意力權重的 dropout 層，用來隨機將部分注意力權重置為零，防止過擬合。
- `proj` 是將最終的多頭注意力輸出投影回原始維度的線性層，proj_drop 是最後的 dropout 層。

```
    def forward(self, x):
        batch_size, num_tokens, dim = x.size()
        assert dim == self.dim, "Input dimension must match model
dimension"
```

- forward 函數定義了這個模組在前向傳播中的具體計算。x 是輸入的張量，形狀為 [batch_size, num_tokens, dim]，其中 batch_size 是批次大小，num_tokens 是序列長度，dim 是每個 token 的向量維度。

```
        q = self.query(x).view(batch_size, num_tokens,
self.num_heads, self.d_k).transpose(1, 2)
        k = self.key(x).view(batch_size, num_tokens, self.num_heads,
self.d_k).transpose(1, 2)
        v = self.value(x).view(batch_size, num_tokens,
self.num_heads, self.d_v).transpose(1, 2)
```

- 輸入 x 經過線性層 query, key, value 之後，會得到對應的 q, k, v 張量。
- view 函數將這些張量重新整形，使每個頭擁有單獨的向量維度，然後 transpose 交換維度順序以便進行多頭並行計算。

```
        attn_scores = torch.matmul(q, k.transpose(-2, -1)) /
math.sqrt(self.d_k)
        attn_probs = torch.softmax(attn_scores, dim=-1)
        attn_probs = self.attn_drop(attn_probs)

        context = torch.matmul(attn_probs, v).transpose(1,
2).contiguous().view(batch_size, num_tokens, dim)
```

- `attn_scores` 是通過 q 和 k 的矩陣相乘來計算的，然後除以 sqrt(d_k) 進行縮放。
- `attn_probs` 是通過 softmax 對 attn_scores 進行歸一化得到的，表示每個 token 的注意力分佈。
- 最後的 `context` 是通過將 attn_probs 和 v 矩陣相乘得到的，這樣得到了加權和的結果。

```
        output = self.proj(context)
        output = self.proj_drop(output)

        return output
```

- 最終的 `context` 經過 proj 投影層，再經過 proj_drop dropout 層，得到最終的輸出。

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

`VQGAN_Transformer.py`

```
class MaskGit(nn.Module):
    def __init__(self, configs, batch_size):
        super().__init__()
        self.vqgan = self.load_vqgan(configs['VQ_Configs'])

        self.num_image_tokens = configs['num_image_tokens']
        self.mask_token_id = configs['num_codebook_vectors']
        self.choice_temperature = configs['choice_temperature']
        self.gamma = self.gamma_func(configs['gamma_type'])
        self.transformer =
BidirectionalTransformer(configs['Transformer_param'])
        self.batch_size = batch_size  # 保存 batch_size

    def load_transformer_checkpoint(self, load_ckpt_path):
        self.transformer.load_state_dict(torch.load(load_ckpt_path))


    @staticmethod
    def load_vqgan(configs):
        cfg = yaml.safe_load(open(configs['VQ_config_path'], 'r'))
        model = VQGAN(cfg['model_param'])
        model.load_state_dict(torch.load(configs['VQ_CKPT_path']),
strict=True)
        model = model.eval()
        return model
```

`MaskGit` 初始化時，它會載入一個預先訓練好的 VQGAN 模型，並配置 Transformer 模型的參數，如圖像 token 數量、遮罩 token ID 和選擇溫度等。模型中的 Transformer 是雙向的，用來處理圖像的 token 進行生成或修復。這段程式碼還包含了載入 VQGAN 和 Transformer 的檢查點的功能，以便從保存的模型狀態繼續訓練或進行推論。

```
##TODO2 step1-1: input x fed to vqgan encoder to get the latent and zq
    @torch.no_grad()
    def encode_to_z(self, x):
        # 使用 VQGAN 的編碼器生成潛在向量
        z, z_indices, q_loss = self.vqgan.encode(x)
        return z, z_indices
```

```
    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 iteration number: t/T.
        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).

        """
        def linear_gamma(ratio):
            return 1 - ratio

        def cosine_gamma(ratio):
            return np.cos(np.pi * ratio / 2)

        def square_gamma(ratio):
            return 1 - ratio ** 2

        if mode == "linear":
            return linear_gamma
        elif mode == "cosine":
            return cosine_gamma
        elif mode == "square":
```

```
            return square_gamma
        else:
            raise NotImplementedError(f"Gamma function mode '{mode}'
is not implemented.")
```

這段程式碼 `gamma_func` 是用來根據給定的模式來生成遮罩比例（mask rate）的函數。它接受一個模式參數（例如 “linear”、“cosine” 或 “square”），並根據不同的模式返回相應的遮罩比例計算函數。

```
##TODO2 step1-3:
    def forward(self, x):
        # Ground truth: encode the input image to z_indices
        z, z_indices = self.encode_to_z(x)

        # 重塑 z_indices 为 (batch_size, num_image_tokens)
        z_indices = z_indices.view(self.batch_size, -1)

        r = math.floor(self.gamma(np.random.uniform()) *
z_indices.shape[1])
        sample = torch.rand(z_indices.shape,
device=z_indices.device).topk(r, dim=1).indices
        mask = torch.zeros(z_indices.shape, dtype=torch.bool,
device=z_indices.device)
        mask.scatter_(dim=1, index=sample, value=True)

        masked_indices = self.mask_token_id *
torch.ones_like(z_indices, device=z_indices.device)
        # 加上隨機生成的 mask
        a_indices = mask * z_indices + (~mask) * masked_indices

        # Get logits from the transformer
        logits = self.transformer(a_indices)

        # print(f"logits shape: {logits.shape}")
        # print(f"z_indices shape: {z_indices.shape}")

        return logits, z_indices
```

`Forward` 的部分還蠻重要的，因為有一些 `MVTM` 的實作也包含在裡面，這個流程的主要目的是通過隨機遮罩部分圖像信息，並利用 Transformer 來預測這些被遮罩的信息，從而訓練模型在缺失部分信息的情況下仍能重建圖像。詳細過程如下：

1. **編碼 Ground Truth**：首先，將輸入圖片 x 使用 encode_to_z 函數編碼，獲得潛在向量 z 和對應的離散索引 z_indices。z_indices 是對應於輸入圖片的編碼表示，用來指代圖像中每個區域的編碼索引。
2. **重塑 z_indices**：然後，將 z_indices 重塑為 (batch_size, num_image_tokens) 的形狀，這樣可以確保每個批次的圖像在接下來的處理步驟中能夠正確處理。
3. **計算遮罩比例**：根據 gamma 函數計算遮罩比例。gamma 函數會根據隨機生成的一個 ratio 值來計算一個遮罩比例，這個比例代表了要遮罩的圖像區域的數量。
4. **生成隨機遮罩**：生成一個隨機的遮罩矩陣。這裡使用 torch.rand 函數生成隨機值矩陣，並使用 topk 函數選擇要遮罩的 r 個位置。然後，初始化一個全為 False 的遮罩矩陣 mask，並使用 scatter_ 函數將選中的 r 個位置設為 True，這些位置代表需要被遮罩的區域。
5. **應用遮罩**：使用 mask 將對應位置的 z_indices 替換為一個特殊的遮罩標記 mask_token_id，其餘未被遮罩的位置保留原本的 z_indices。
6. **Transformer 處理**：將遮罩後的 z_indices 傳入 Transformer，並獲得對應的 logits。這些 logits 代表了每個位置的編碼結果，它們將用於訓練模型，讓模型學會在給定部分信息的情況下預測整個圖像。
7. **返回結果**：最後，該函數返回 Transformer 的 logits 和未被遮罩的原始 z_indices，後者作為 ground truth 用於計算損失。

`training_Transformer.py`

```
    def configure_optimizers(self):
        optimizer = torch.optim.AdamW(self.model.parameters(),
lr=self.args.learning_rate, weight_decay=1e-5)
        scheduler = torch.optim.lr_scheduler.StepLR(optimizer,
step_size=10, gamma=0.5)
        return optimizer, scheduler

    def train_one_epoch(self, train_loader, epoch):
        self.model.train()
        running_loss = 0.0
        for batch_idx, data in enumerate(tqdm(train_loader,
ncols=140)):
            images = data.to(device=self.args.device)
```

```
            logits, z_indices = self.model(images)
            # print('================')

            # 計算 cross-entropy loss
            loss = F.cross_entropy(logits.view(-1, logits.size(-1)),
z_indices.view(-1))

            # 梯度累積
            loss = loss / self.args.accum_grad
            loss.backward()

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

            running_loss += loss.item() * self.args.accum_grad

        epoch_loss = running_loss / len(train_loader.dataset)
        print(f"Train Epoch: {epoch} Loss: {epoch_loss:.6f}")
        return epoch_loss

    def eval_one_epoch(self, val_loader, epoch):
        self.model.eval()
        running_loss = 0.0
        with torch.no_grad():
            for batch_idx, data in enumerate(tqdm(val_loader,
ncols=120)):
                images = data.to(device=self.args.device)
                logits, z_indices = self.model(images)

                # 計算 cross-entropy loss
                loss = F.cross_entropy(logits.view(-1,
logits.size(-1)), z_indices.view(-1))

                running_loss += loss.item()

        epoch_loss = running_loss / len(val_loader.dataset)
```

```
        print(f"Val Epoch: {epoch} Loss: {epoch_loss:.6f}")
        return epoch_loss
```

`training_transformer.py` 主要實現了模型訓練過程中的優化器配置、訓練單個 epoch 和驗證單個 epoch 的流程。

首先，`configure_optimizers` 函數定義了優化器和學習率調度器。使用 `AdamW` 作為優化器，它是 Adam 的變體，並且加入了權重衰減來防止過擬合。學習率調度器使用了 StepLR，每過一定步數後將學習率減少一半，這有助於在訓練過程中穩定模型的收斂。

在 `train_one_epoch` 函數中，模型進入訓練模式。對於每個批次的數據，先將圖片數據送入模型進行前向傳播，得到預測的 logits 和對應的索引 z_indices。然後使用 `交叉熵損失函數` 計算 loss。**交叉熵損失衡量的是預測分佈與真實分佈之間的差異，在這裡 logits 代表模型的預測，而 z_indices 代表真實值。將 logits 和 z_indices 展平後，計算 loss。**

接下來，loss 會被縮放（除以 accum_grad）以便進行梯度累積，這可以有效減少顯存的佔用。然後進行反向傳播計算梯度。每當累積到一定批次（accum_grad 批）後，更新模型參數並將梯度清零。最後，將這個批次的 loss 累加到 running_loss 中。

eval_one_epoch 與訓練類似，但模型進入驗證模式，並且在計算梯度時使用 torch.no_grad() 來防止計算圖的構建，從而節省內存。最後，返回這一個 epoch 的平均 loss。

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

`VQGAN_Transformer.py` 的 `inpainting` 函數 (one step decoding)

```
    @torch.no_grad()
    def inpainting(self, masked_z_indices, mask_bc, step, total_iter,
mask_num, gamma_type):

        masked_z_indices[mask_bc] = 1024

        # Step 1: Obtain logits from the transformer
        logits = self.transformer(masked_z_indices)
        # print(logits.size())

        # Step 2: Apply softmax to convert logits into a probability
distribution across the last dimension.
        probs = torch.softmax(logits, dim=-1)

        # Step 3: Find the maximum probability for each token value
```

```
        z_indices_predict_prob, z_indices_predict = torch.max(probs,
dim=-1)
        z_indices_predict_prob[~mask_bc] = float('inf')
        # Step 4: Calculate the current ratio
        # ratio = self.gamma((step + 1) / total_iter)
        ratio_func = self.gamma_func(mode=gamma_type)
        ratio = ratio_func((step + 1) / total_iter)

        # Step 5: Add temperature annealing gumbel noise as
confidence
        gumbel_noise = -torch.log(-
torch.log(torch.rand_like(z_indices_predict_prob)))
        # gumbel_noise, _ = torch.max(gumbel_noise, dim=-1)
        # print(f"z_indices_predict_prob shape:
{z_indices_predict_prob.shape}")
        # print(f"gumbel_noise shape: {gumbel_noise.shape}")
        temperature = self.choice_temperature * (1 - ratio)
        confidence = z_indices_predict_prob + temperature *
gumbel_noise

        # print(confidence)

        # Step 6: Sort the confidence for ranking
        sorted_confidence, sorted_indices = torch.sort(confidence)

        z_indices_predict[~mask_bc] = masked_z_indices[~mask_bc]
        # print(ratio*mask_num)
        mask_bc[:, sorted_indices[:, math.floor(ratio*mask_num):]] =
False

        # Return the updated predictions and mask
        return z_indices_predict, mask_bc
```

這段程式碼實現了對被遮罩（masked）的圖像進行修補（inpainting）的過程。首先，利用傳送進來的遮罩索引來標記那些需要修補的部分，並將它們設為一個特殊的標記值（1024）。接著，通過 Transformer 模型預測這些遮罩部分的內容。預測結果經過 softmax 操作轉換為概率分佈，然後找到最大概率所對應的預測值。

為了提升預測的穩定性和多樣性，程式碼引入了溫度調整和 Gumbel 噪聲，這些都用來增加不確定性，從而更好地排名預測值。這些排序後的預測值會進一步更新遮罩狀態，確定哪些部分應該被揭示（即修補完成）。最終，程式碼返回了更新後的預測結果和遮罩狀態，繼續進行修補的迭代過程。這樣的修補過程將逐步完成對圖像被遮蔽部分的修復。

`inpainting.py` (total step decoding)

```
    ##TODO3 step1-1: total iteration decoding
    #mask_b: iteration decoding initial mask, where mask_b is true
means mask
    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 to be the ground truth of
masked image

        self.model.eval()
        with torch.no_grad():
            z, z_indices = self.model.encode_to_z(image)  #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)

            # Iterative decoding loop
            for step in range(self.total_iter):
                if step == self.sweet_spot:
                    break
```

```
                z_indices_predict =
z_indices_predict.view(self.batch_size, -1)
                # print(z_indices_predict.size())

                # Perform the iteration of inpainting
                z_indices_predict, mask_bc =
self.model.inpainting(z_indices_predict, mask_bc, step,
self.total_iter, mask_num, self.mask_func)


                # 在每個步驟後進行檢查
                # print(f"Step {step}: unmasked tokens:
{mask_bc.sum().item()}")
                # print(f"Step {step}: z_indices_predict:
{z_indices_predict}")

                # Visualize the current mask
                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 true
                mask_image[:, indices[:, 1], indices[:, 2]] = 0
 #3,16,16
                maska[step] = mask_image

                # Decode the image from latent space
                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 # save decoded image

            # Save the decoded image and mask scheduling results
            vutils.save_image(dec_img_ori,
os.path.join("test_results", f"image_{i:03d}.png"), nrow=1)
```

```
            vutils.save_image(maska, os.path.join("mask_scheduling",
f"test_{i}.png"), nrow=10)
            vutils.save_image(imga, os.path.join("imga",
f"test_{i}.png"), nrow=7)
```

這段程式碼實現了對圖像進行逐步修補的過程，將部分被遮罩的圖像逐步還原為完整的圖像。首先，它會初始化一些變數來儲存所有迭代中的遮罩狀態和解碼後的圖像，並將輸入圖像的標準化版本存入初始狀態中。

接著，程式進入了一個迴圈，逐步執行修補過程。在每一次迭代中，程式會根據目前的遮罩狀態預測圖像中被遮罩部分的像素值，然後更新遮罩狀態以確定哪些部分已經被還原。隨著迭代次數的增加，逐漸揭示出整個圖像的內容。每次迭代後，會將目前的遮罩情況視覺化，並將其儲存起來，同時也將目前的解碼圖像儲存起來。

在迴圈結束後，最終修補完成的圖像會被保存到指定的目錄中，同時也會保存整個修補過程中各個步驟的中間結果。這樣的過程讓模型可以逐步推斷出被遮罩部分的內容，最終生成一個完整的圖像。

3. Discussion

訓練過程的發現與改進

一開始將程式寫完後拿去跑其實 fid 超級高，大概 58 左右。

後來做了以下調整，有逐漸降低 fid 到最後能夠低到37多。

首先，我覺得影響最大的就是 `sweet_spot` ，原本我覺得 iteration 數越多，decode 出來的結果會越好。

然而實際上卻反了過來，模型到後面都會越補越大洞 (如下圖，後面把一隻眼睛補不見了)，因此到最後我只把 `sweet_spot` 設成 2。

然後，transformer 的訓練針對結果產出也很相關，我試過把 learning rate 調大或調小，但後來還是覺得一開始的 1e-4 就很適合了。觀察 loss curve 發現可能問題也不太像是 overfit，valid loss 其實一直很不是穩定，會浮動。但所幸浮動的範圍不是非常大。最後發現 batch size 不能設太大，原本是 32 ，後來改成 8 ，訓練就變得更加穩定了 。我推測原因可能是當 batch size 減小時，每次更新模型參數的頻率增高。這意味著模型能更頻繁地更新參數，可能會更快適應數據中的變化，從而減少 loss 波動。

總而言之，我認為這是一個很有趣的作業，感謝助教批改！