NYCU DL Lab5
MaskGIT for Image Inpainting

313552049 鄭博涵

1. Introduction

在這次實驗中, 我們的目標是訓練 MaskGIT 中的雙向 Transformer，並使用它來解決 Inpainting 的任務, 在實作方面，我們需要完成三個部分, 分別是 Multi-head Attention Layer 的實作、雙向 Transformer 的 train strategy，以及最後用於圖像修補的 Iteration Decoding, 最後, 我們比較不同的遮罩率增減方式(cosine, linear, square) 對圖像修補結果的影響，並使用 FID 作為評估指標

最終我的 FID 是 38.9

2. Implementation Details

A. The details of your model

(Multi-Head Self-Attention)

```python
#TODO1
class MultiHeadAttention(nn.Module):
    def __init__(self, dim=768, num_heads=16, attn_drop=0.1):
        super(MultiHeadAttention, self).__init__()
        self.n_head = num_heads
        self.dim = dim
        # key, query, value dimension for all heads
        self.key = nn.Linear(dim, dim)
        self.query = nn.Linear(dim, dim)
        self.value = nn.Linear(dim, dim)
        # regularization
        self.attn_drop = nn.Dropout(attn_drop)
        # output projection
        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.
        '''
        B, T, D = x.size()
        k = self.key(x).view(B, T, self.n_head, D // self.n_head).transpose(1, 2) # (B, nh, T, d_k)
        q = self.query(x).view(B, T, self.n_head, D // self.n_head).transpose(1, 2) # (B, nh, T, d_q)
        v = self.value(x).view(B, T, self.n_head, D // self.n_head).transpose(1, 2) # (B, nh, T, d_v)
        att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1))) # (B, nh, T, T)
        att = nn.functional.softmax(att, dim=-1)
        att = self.attn_drop(att)
        y = att @ v # (B, nh, T, T) x (B, nh, T, hs) -> (B, nh, T, hs)
        ## [ B x T x D ]
        y = y.transpose(1, 2).contiguous().view(B, T, D) # re-assemble all head outputs side by side
        return self.proj(y)
```

__init__(): 首先在模組初始化時會設置一些參數, 其中 num_head 是指將注意力機制分成了多少個平行的注意力頭, 每個頭會獨立地學習不同部分的特徵, attn_drop 則是注意力機制的 drop 參數

然後將 key, query, value 三個向量分別設置了 nn.Linear, 用來將輸入的特徵轉換到合適的維度, 最後的輸出通過一個 nn.Linear 進行投影

Forward(): 首先 B, T, D 指的是 batch_size, num_image_tokens, dim

再來通過線性轉換層計算出 key, query, value 張量，然後將其分成多個 head，每個 head 的維度為 D // num_head

然後使用 query, key 來計算注意力分數，這個分數矩陣表示序列中不同位置之間的關聯性，再對注意力分數進行 softmax 操作，使其正規化為概率分佈

y = att @ v：將注意力分數與 value 相乘，得到最終的注意力加權表示

y = y.transpose(1, 2).contiguous().view(B, T, D): 將所有頭的輸出重新排列，並且將輸出形狀設為(Batch_size, num_image_tokens, dim)的張量，最後再 return self.proj(y): 表示通過投影層進行線性轉換

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

```python
# TODO2 step1: design the MaskGIT model
class MaskGit(nn.Module):
    def __init__(self, configs):
        super().__init__()
        self.vqgan = self.load_vqgan(configs["VQ_Configs"]) # 初始化VQGAN model, 將其存儲在self.vqgan (下面會定義load_vqgan)

        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"])

    def load_transformer_checkpoint(self, load_ckpt_path):  # 在inpainting時會用到, 用來加載已訓練的transformer model 的權重
        self.transformer.load_state_dict(torch.load(load_ckpt_path))

    @staticmethod # @staticmethod: 表示load_vqgan方法是一個靜態方法 -> 與class的實例無關
    def load_vqgan(configs):
        # 打開yaml配置文件 並以read mode 讀取文件, yaml.safe_load: 將該yaml文件的內容加載為python字典cfg, cfg包含了VQGAN model的各種配置參數
        cfg = yaml.safe_load(open(configs['VQ_config_path'], 'r'))
        model = VQGAN(cfg['model_param'])   # 初始化了一個 VQGAN model的實例
        model.load_state_dict(torch.load(configs['VQ_CKPT_path'], weights_only=True), strict=True) # 加載以訓練的權重, torch.load: 從該路徑加載權重
        model = model.eval()
        return model
```

__init__(): 首先 load_vqgan()會讀取配置文件，並根據其中的參數初始化 VQGAN，其中我想解釋的參數有:

self.num_image_token: 是 model 將處理的圖像 token 數量，

self.mask_token_id: 是遮罩 token 的 id

self.choice_temperature: 是一種溫度參數，較低時會使 model 更自信，較高時則會增加預測的隨機性

再來是 Encoder_to_z:

```
##TODO2 step1-1: input x fed to vqgan encoder to get the latent and zq
# 將輸入圖像轉換為VQGAN的潛在表示和量化表示
    @torch.no_grad()
    def encode_to_z(self, x):
        codebook_mapping, codebook_indices, _ = self.vqgan.encode(x)
        return codebook_mapping, codebook_indices
```

這邊的功能是利用 VQGAN model 的編碼器來生成圖像的潛在向量(codebook_mapping) 和對應的量化索引(codebook_indices)

接著是 Gamma_function:

```
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).

    """
    if mode == 'Linear':
        return lambda x: 1 - x  # x為輸入的ratio
    elif mode == 'cosine':
        return lambda x: np.cos(x * np.pi / 2)
    elif mode == 'square':
        return lambda x: 1 - x ** 2
    else:
        raise NotImplementedError
```

gamma_func 的主要功能是生成遮罩率(ratio)，這個遮罩率決定了圖像中的 token 有多大比例被 mask，其中有 Linear, cosine, square 這三種不同的遮罩率排成策略

MVTM: 全名是 Multi-Variate Transformer Masking，在 MaskGIT 中指的是一種遮罩策略，用來隨機選擇圖像中的一部分 token 進行遮罩，然後用 transformer model 來預測這些被 mask 的 token

```python
##TODO2 step1-3:
    """ 將輸入圖像進行編碼、遮罩處理，然後通過 Transformer 模型進行預測，最終返回預測的 logits 和對應的 ground truth """
    def forward(self, x, ratio):
        _, z_indices = self.encode_to_z(x)  # 編碼輸入圖片，獲取對應的量化索引
        z_indices = z_indices.view(-1, self.num_image_tokens)   # 將 z_indices 形狀調整為 (batch_size, num_image_tokens)

        # 創建遮罩，並將 z_indices 中被遮罩的位置設為 mask_token_id
        mask = torch.rand_like(z_indices, dtype=torch.float) < ratio  # 隨機遮罩，probability of masking is `ratio`
        z_indices_input = z_indices.masked_fill(mask, self.mask_token_id)  # 用 masked_fill 替換被遮罩的元素

        logits = self.transformer(z_indices_input)  # 使用 Transformer 預測被遮罩位置的 token
        logits = logits[..., :self.mask_token_id]  # 只保留與 mask_token_id 前的 logits

        # 使用 one-hot 編碼生成 ground truth
        ground_truth = F.one_hot(z_indices, num_classes=self.mask_token_id).float()

        return logits, ground_truth
```

在 forward 當中，透過 mask = torch.rand_like(z_indices, dtype=torch.float) < ratio 來實現 MVTM 的核心概念，即通過隨機遮罩來模擬圖像信息缺失的情況

而 z_indices_input = z_indices.masked_fill(mask,self.mask_token_id) 則是被選中的遮罩部分會被填充為一個特殊的 mask_token_id，然後利用 Transformer model 根據上下文來預測被遮罩的信息

(logits = self.transformer(z_indices_input) 的部分)

Loss:

```python
#TODO2 step1-4: design the transformer training strategy
class TrainTransfromer:
    def __init__(self, args, MaskGit_configs):
        self.args = args
        self.device = args.device
        self.learning_rate = args.learning_rate
        self.model = VQGANTransformer(MaskGit_configs["model_param"]).to(self.device)   # 在設置模型時會用.to(device), 來確保model跟數據是在同一設備上
        self.optimizer, self.scheduler = self.configures_optimizers()   # configures_optimizers() 下面會定義
        self.prepare_training()
        self.writer = SummaryWriter("transformer_checkpoints/logs/")

    @staticmethod   # 靜態方法與實例無關
    def prepare_training(): # exist_ok: 當目錄存在時, 不會拋出錯誤
        os.makedirs("transformer_checkpoints", exist_ok=True)   # 創建 transformer_checkpoints 目錄, 裡面放checkpoint

    def train_one_epoch(self, train_loader, epoch): # 在一個epoch裡所要做的train動作
        self.model.train()
        losses = []

        for x in tqdm(train_loader, ncols=90, leave=False, desc=f"Train Epoch {epoch}"):    # desc: 用來設定進度條前顯示的文字 -> 知道目前在哪個epoch
            x = x.to(self.device)
            self.optimizer.zero_grad()
            ratio = np.random.rand()
            y_pred, y = self.model(x, ratio)
            loss = F.cross_entropy(y_pred, y)
            loss.backward()
            torch.nn.utils.clip_grad_norm_(self.model.parameters(), max_norm=1.0)   # 梯度裁減
            self.optimizer.step()

            losses.append(loss.detach().item())
        self.scheduler.step()
        return np.mean(losses)

    # 驗證階段
    def eval_one_epoch(self, val_loader, epoch):
        self.model.eval()
        losses = []

        with torch.no_grad():
            for x in tqdm(val_loader, ncols=90, leave=False, desc=f"Val Epoch {epoch}"):    # leave=False: 進度條完成後不會保留在終端上
                x = x.to(self.device)
                ratio = np.random.rand()
                y_pred, y = self.model(x, ratio)
                loss = F.cross_entropy(y_pred, y)
                losses.append(loss.detach().item())
        return np.mean(losses)
```

__init__: 初始化了 model, device, lr, optim 等基本參數, 並設置了 learningrate scheduler 及 log 的保存路徑

Train_one_epoch: 負責執行 model 在一個 train epoch 中的所有步驟, 在每個 batch 中隨機生成 mask ratio 並將其應用於輸入數據, 之後計算 loss 使用 cross_entropy, 並透過 backward 來更新 model weight, 在每次訓練後使用梯度裁減以防止梯度爆炸, 並累積損失以計算平均損失

```python
#TODO2 step1-5:
for epoch in range(args.num_fast_epochs + 1, args.epochs + 1):
    best_val_loss = float('inf')
    train_loss = train_transformer.train_one_epoch(train_loader, epoch)
    val_loss = train_transformer.eval_one_epoch(val_loader, epoch)

    train_transformer.writer.add_scalar('Loss/train', train_loss, epoch)
    train_transformer.writer.add_scalar('Loss/val', val_loss, epoch)

    print(f"Epoch {epoch}/{args.epochs}  |  Train Loss: {train_loss:.4f}   | Val Loss: {val_loss:.4f}  |  lr: {train_transfo

    if val_loss < best_val_loss :
        best_val_loss = val_loss
        torch.save(train_transformer.model.transformer.state_dict(), f"transformer_checkpoints/best_ckpt_{epoch}.pt")
```

進行多個 epoch 的 training, 每個 epoch 都包括 train_one_epoch 和 eval_one_epoch, 每次訓練完成後, 會計算 train_loss 以及 val_loss, 並加入到 Tensorboard 中

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

inpainting(在 VQGAN_Transformer 裡):

```python
## TODO3 step1-1: define one iteration decoding
    @torch.no_grad()
    def inpainting(self, x, ratio, mask_b): # mask_b: 是boolean變數, 用來標記哪些位置在當前在當前步驟中需要被遮罩或處理
        # 跟forward幾行類似
        _, z_indices = self.encode_to_z(x)
        z_indices_input = torch.where(mask_b == 1, torch.tensor(self.mask_token_id).to(mask_b.device), z_indices)

        logits = self.transformer(z_indices_input)
        logits = F.softmax(logits, dim=-1)   # softmax將logits轉換為概率分佈 -> 才可以用torch.max()之類的方法, 找到每個位置上最有可能的token值

        z_indices_predict_prob, z_indices_predict = torch.max(logits, dim=-1)   # 尋找每個token預測值的最大概率

        ratio = self.gamma(ratio)   # 調整masking ratio
        # torch.log(torch.rand_like(...)): 計算的是隨機數的自然對數，這個自然對數通常是負數或零 -> 取"-"後變成正數
        gumbel_noise = -torch.log(-torch.log(torch.rand_like(z_indices_predict_prob))) # 生成gumbel噪聲 -> 常用於採樣的噪聲, 並引入隨機性

        temperature = self.choice_temperature * (1 - ratio)  # 溫度越高, 隨機性越大, 溫度越低, 結果越確定
        confidence = z_indices_predict_prob + temperature * gumbel_noise    # 計算信心值 -> 有效避免model過度確定某些預測結果

        confidence[mask_b == 0] = float('inf')  # 將未遮罩的部分的信心值設為無限大

        # item(): 將張量 (Tensor) 中的單個元素提取出來, 並將其轉換為純 Python 數值類型, 如 int 或 float
        n = max(1, int(mask_b.sum().item() * ratio))
        _, idx_to_mask = torch.topk(confidence, n, largest=False)

        # 生成新的遮罩 mask_bc, 並對其進行更新, 以便確定哪些位置應該在當前步驟中被替換或保留
        mask_bc = torch.zeros_like(mask_b, dtype = torch.bool)   # 形狀跟mask_b相同
        mask_bc.scatter_(1, idx_to_mask, 1) # 在指定的維度 (第1維) 上進行元素更新
        mask_bc = mask_bc & mask_b  # 只有那些在 mask_b 中已經被遮罩, 且在新的遮罩中也被選中的位置，才會保留 True

        return z_indices_predict, mask_bc
```

使用 torch.where 函數將遮罩位置的 z_indices 替換為特殊的 mask_token_id，以標記需要預測的部分，再將遮罩後的 z_indices_input 傳遞給 Transformer 模型進行預測，並通過 softmax 轉換成概率分佈，然後找出最有可能的 token 值及其對應的最大概率，接著計算 ratio 以及 use Gumbel 噪聲來引入隨機性，並結合 temperature 來調整信心值，避免 model 過於自信

之後使用 torch.topk 函數來選出信心值最低的 token，更新 mask，以確定哪些位置在下一步中需要進行進一步的預測，

最後返回更新後的 z_indices_predict 和 mask_bc，會用於後續的 inpainting 中

inpainting (total iteration decoding):

```python
##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): #MaskGIT 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 = self.model.encode_to_z(image[0].unsqueeze(0))[1] #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)

            start_ratio = 0
            end_ratio = 1
```

初始化的部分:

maska: 用來保存每次迭代過程中生成的 mask

imga: 用來保存每次迭代過程中解碼後的圖像

ori: 對初始輸入圖像進行標準化變換後的結果, 然後用 imag [0] 保存這個基準圖像, 作為遮罩圖像的基線

model 推理準備的部分:

設為評估模式並使用 encoder_to_z 將輸入圖像 編碼為潛在表示, 得到 z_indices, 然後 mask_num 計算了初始遮罩中需要被修補的 token 數量

Iteration encoding:

```
#iterative decoding for loop design
#Hint: it's better to save original mask and the updated mask by scheduling separately
for step in range(self.total_iter):
    ratio = start_ratio + (end_ratio - start_ratio) * (step / (self.total_iter - 1))    # 使用線性插值來計算 ratio

    """ 這邊用到了VQGAN_Transformer裡面的inpainting函數 """
    z_indices_predict, mask_bc = self.model.inpainting(image, ratio, mask_bc)

    if mask_bc.max().item() >= mask_bc.numel():     # check是否有誤
        raise ValueError(f"Invalid index detected in mask_bc at step {step}")

    #static method yon can modify or not, make sure your visualization results are correct
    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
    shape = (1, 16, 16, 256)
    # assert torch.all(z_indices_predict < self.model.vqgan.codebook.embedding.num_embeddings)
    z_q = self.model.vqgan.codebook.embedding(z_indices_predict).view(shape)
    z_q = z_q.permute(0, 3, 1, 2)
    # print(f"z_q shape: {z_q.shape}")  // debugging line
    decoded_img = self.model.vqgan.decode(z_q)
    dec_img_ori = (decoded_img[0] * std) + mean
    image = decoded_img
    imga[step + 1 ] = dec_img_ori #get decoded image

    """ Save the result at the sweet spot  """
    if step == self.sweet_spot:
        sweet_spot_result = dec_img_ori.clone()

# 將甜蜜點的圖片儲存到test_results 以便計算FID
if sweet_spot_result is not None:
    vutils.save_image(sweet_spot_result, os.path.join("./Result/test_results", f"image_{i:03d}.png"), nrow=1)

vutils.save_image(maska, os.path.join("./Result/mask_scheduling", f"test_{i}.png"), nrow=10)    # 創造出2x10的圖
vutils.save_image(imga, os.path.join("./Result/imga", f"test_{i}.png"), nrow=7) # 3x7的圖
```

先計算 ratio, 再使用 one iteration encoding 的 inpainting()

來預測當前被遮罩區域的 token (z_indices_predict), 並更新遮罩 (mask_bc), 然後將每次迭代的遮罩 mask_image 和解碼圖像 dec_img_ori 保存到 maska 和 imaga 中, 用於後續分析

接著將預測出的 token 轉換為對應的嵌入向量 z_q, 並透過 VQGAN model 進行解碼, 得到重建的圖像, 當迭代達到預設的 sweet_spot 時, 保存當前的解碼圖像到 test_results 資料夾裡, 以便後續的 FID 計算

最終保存結果:

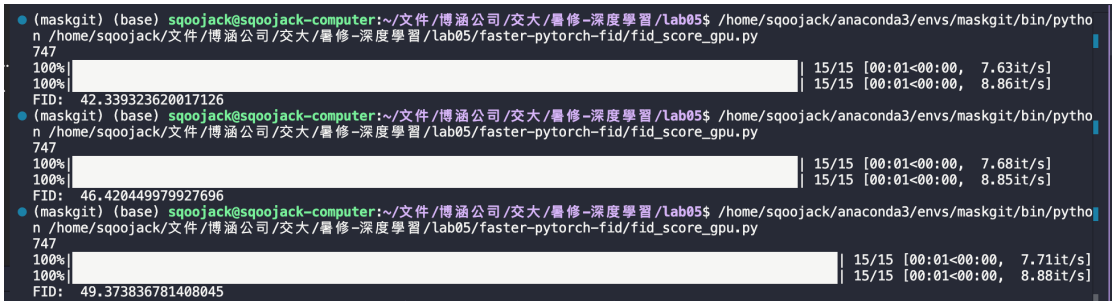
test_results 資料夾: 存放 sweet_spot 的最佳修補圖像

mask_scheduling 資料夾: 保存所有迭代步驟(step)的遮罩圖像

imga 資料夾: 保存所有迭代步驟的解碼圖像

3. Discussion

以下是我將 sweet_spot 分別設為 2, 5, 8, 11 之後所計算出來的 FID, 皆是用 square 方法, seed 為固定, 所以由此可知, 越後面的 step, 所預測出來的效果越差



我推測原因可能有以下幾點:

1. 遮罩範圍的逐步縮小:

在 maskGIT 的修補過程中, 隨著迭代的進行, 遮罩範圍會逐步減少, 表示在後期時需要修補的區域越來越小,

剩下的未修補可能是較難以預測的部分, 而 model 在前幾個 step 中已經生成了相對穩定的圖像, 但後期在修補難以預測的部分時可能會引入一些誤差和不自然的細節, 導致 FID 上升

2. 細節誤差逐步放大:

前幾次迭代中累積的細微誤差可能會逐漸放大, 所以在後期時, 這些誤差可能會更加顯著, 使得最終生成的圖像與真實圖像相比更不自然

3. Model 信心值的變化:

在多次迭代過程中, model 可能對於後期的預測逐漸失去信心, 使得後面的修補步驟更加不可靠, 這時 model 的預測可能會更依賴於隨機性, 所以導致 FID 上升

Reference:

KJLdefeated/NYCU_DLP_2024: 2024 NYCU Deep Learning_ (github.com)