NYCU
Deep Learning Lab-3
MaskGIT for Image Inpainting

學號 : 314551113

姓名 : 劉哲良

Content

Figure Content

1. Introduction

在本次作業中，我們需要實作 MaskGIT 的核心程式碼，如 Multi-Head Self-Attention Module，模擬 Bidirectional Transformer 預測 token 的過程，以及最後的 decode 將圖片逐漸地修補完成。以下報告將會介紹模型內部實作細節，以及討論在各個參數實驗之下訓練出來的結果，結果的好壞將用 FID 分數來進行比較。

2. Implementation Details

2.1 Details of Multi-Head Self-Attention

```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
        self.dim = dim
        self.attn_drop =attn_drop

        self.head_dim = dim // num_heads

        # Weight matrix of Q,K,V
        self.W_Q = nn.Linear(dim,dim)
        self.W_K = nn.Linear(dim,dim)
        self.W_V = nn.Linear(dim,dim)

        self.dropout = nn.Dropout(attn_drop)

        self.output = nn.Linear(dim,dim)
    def forward(self, x):

        batch_size,num_image_tokens,dim = x.shape

        Q = self._split_head(self.W_Q(x))
        K = self._split_head(self.W_K(x))
        V = self._split_head(self.W_V(x))

        scale = math.sqrt(self.head_dim)        # matrix multiplication
        attention_scores = (Q @ K.transpose(-2,-1)) / scale

        attention_prob = attention_scores.softmax(dim=-1)
        attention_drop = self.dropout(attention_prob)

        # Concate
        attention_weight = (attention_drop @ V)

        output = attention_weight.permute(0,2,1,3).reshape(batch_size,num_image_tokens,dim)

        output =self.output(output)

        return output

    def _split_head(self,x):
        batch_size,num_image_tokens,dim = x.shape
        # Input size (batch_size, num_image_tokens, dim)
        # first divide dim into (num_head,head_dim) => (batch_size,num_image_tokens,num_head,head_dim)

        # Then transform it into (batch_size,num_head,num_image_tokens,head_dim)
        return x.view(batch_size,num_image_tokens,self.num_heads,self.head_dim).permute(0,2,1,3)
```

在這裡我們使用 Linear Layer 來計算 Query, Key, Value，並且將計算出來的結果透過 _split_head function 做 reshape 和 permute，轉成 multi-head 的形式，接下來就是正常的 attention score 計算過程。

在計算完 attention score 後，再還原成原本的 shape，最後使用一層 Linear Layer 來將 multi-head 的結果 concate 在一起。

Fig. 1 Multi-Head Self-Attention

2.2 Details of Stage 2 Training

2.2.1 Basic Function

使用 encode_to_z function 將 Input Data 給 VQGAN 的 Encoder 做 encode，輸出會得到對應的 codebook mapping 以及各個 token 對應的 codebook index，我們將 codebook index 做 flatten，以方便作為 Transformer 的輸入。

```
@torch.no_grad()
def encode_to_z(self, x):
    codebook_mapping, codebook_indices, q_loss = self.vqgan.encode(x)

    # Flatten codebook_indices from (batch,16,16) into (batch,256) for transfomer
    return codebook_mapping, codebook_indices.view(codebook_mapping.shape[0], -1)
```

Fig. 2 encode_to_z function

Gamma function 為在 inpainting 過程時決定 inference 時 mask 數量的 function，根據 current step / total step 作為參數進行調整。

```
def gamma_func(self, mode="cosine"):

    if mode == "linear":
        return  lambda gamma : 1-gamma
    elif mode == "cosine":
        return lambda gamma: np.cos(gamma*np.pi/2)
    elif mode == "square":
        return lambda gamma: 1 - gamma**2
    else:
        raise NotImplementedError
```

Fig. 3 gamma function

2.2.2 MVTM

```
def forward(self, x):
        z_indices=None #ground truth
        logits = None  #transformer predict the probability of tokens

        #Ground Truth
        _ , z_indices = self.encode_to_z(x)

        # mask id : self.mask_token_id
        # Normal distribution for percentage of masking
        mask_ratio = np.random.uniform(0.1, 0.9)
        # True : mask
        mask = torch.bernoulli(mask_ratio* torch.ones(z_indices.shape, device=z_indices.device)).bool()

        masked_indices = torch.where(mask, self.mask_token_id, z_indices)
        logits = self.transformer(masked_indices)

        z_indices=z_indices # ground truth
        logits = logits  # Probabilities that predicted by transformer
        return logits, z_indices
```

Fig. 4 MVTM

MVTM 的實作方法，我們先將 Input 做 encode 得到 codebook 的 ground truth，training 使用常態分佈取 10% ~ 90%的 mask 比率，並且產生對應的 mask 以訓練 Transfomer。

得到 transformer 對於 mask 的預測結果後，對於其預測結果以及 VQGAN encoder 產出的 Ground Truth 進行 loss 的計算，我們能將 token 的預測視為 multi-class classification 的問題，因此 loss 的選擇使用 Cross Entropy，並藉由 loss 來更新 model weight。

```python
def train_one_epoch(self,train_loader,epoch,args):
    self.model.train()
    total_loss = 0.0
    num_batches = len(train_loader)
    progress_bar = tqdm(train_loader, desc=f"Epoch [{epoch}|{args.epochs}]", total=num_batches)
    for step,images in enumerate(progress_bar):

        images = images.to(args.device)

        # Forward ,get logits and true tokens
        logits, z_indices = self.model(images)  # logits: (batch_size, 256, 1024), z_indices: (batch_size, 256)

        # logits: (batch_size * 256,1024) , z_indice : (batch_size*256,1)
        loss = F.cross_entropy(logits.reshape(-1, logits.size(-1)), z_indices.reshape(-1))

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

        total_loss += loss.item()
        progress_bar.set_postfix({"loss": f"{total_loss / (step + 1):.4f}"})

    avg_loss = total_loss / num_batches

    self.train_losses.append(avg_loss)

    if self.scheduler_type == "ReduceLROnPlateau":
        self.scheduler.step(avg_loss)  # Step based on training loss
    else:
        self.scheduler.step()  # Step for LinearLR + CosineAnnealing

    return avg_loss
```

Fig. 5 Forward/Loss Transformer

2.3 Details of Inference for Inpainting Task

在每次 inference inpainting 時，我們對於每個 iteration，先將輸入的 tokens 產生 masked_tokens，並給 Transformer 做預測，再將預測結果作機率的轉換並得到每個 tokens 對應機率最大的 codebook index，再將 masked_tokens mask 的部分替換成預測的結果。

接下來在進行 confidence 的計算，在這裡我們將 unmask 的 token confidence 設為 INF，以避免被作為 mask 的目標。之後，找出 confidence 最小的 tokens，並將其作 mask，作為下一次 iteration 的 predict 目標，並產生新的 mask 回傳以進行下一次 iteration。

```python
@torch.no_grad()
def inpainting(self,z_indices,mask ,mask_num, ratio):

    # Generate masked token sequence
    # True : mask, False : unmask
    masked_indices = torch.where(mask, self.mask_token_id, z_indices)

    # Predict token probabilities using transformer
    logits = self.transformer(masked_indices)  # Shape: (batch_size, seq_len, num_codebook_vectors)

    probs = F.softmax(logits, dim=-1) # (batch_size, seq_len, num_codebook_vectors)

    # find max prob of each token
    # (batch_size, seq_len)
    z_indices_predict_prob, z_indices_predict = probs.max(dim= -1)
    # mask the maked part using predicted value
    z_indices_predict = torch.where(mask,z_indices_predict,z_indices)

    gumble = torch.distributions.Gumbel(0, 1).sample(z_indices_predict_prob.shape).to(z_indices_predict.device)  # gur
    temperature = self.choice_temperature * (1 - ratio)
    confidence = z_indices_predict_prob + temperature * gumble

    # The number of mask of next iteration
    num_mask = math.floor(self.gamma(ratio) * mask_num)
    # Make sure we dont modify those unmask token
    confidence[~mask] = torch.inf
    # Select those  low confidence token as masked token
    _, idx = confidence.topk(num_mask, dim=-1, largest=False) #update indices to mask only smallest n token
    mask_bc = torch.zeros(z_indices.shape, dtype=torch.bool, device= z_indices_predict.device)
    mask_bc = mask_bc.scatter_(dim= 1, index= idx, value= True)
    return z_indices_predict, mask_bc
```

Fig. 6 Inpainting

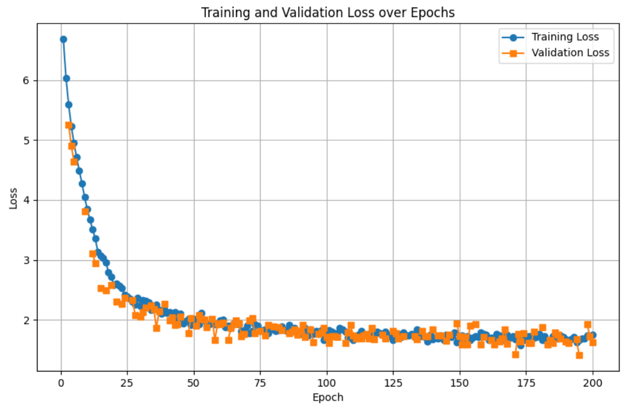
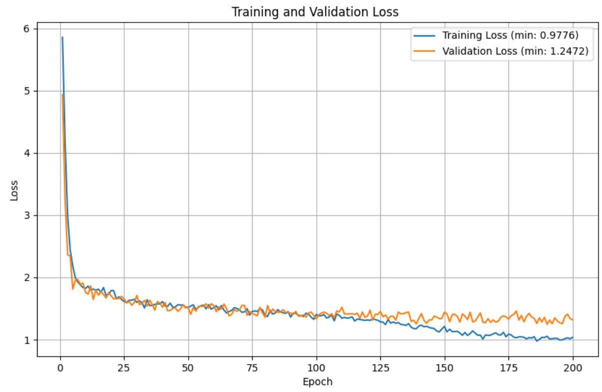
3. Discussion

3.1 The Influence of Total_Iter and Sweet Spot parameters

在本次實驗中，我嘗試了不同 total_iter 和 sweet spot 參數的調整以觀察 FID 分數的變化，在 total_iter 和 sweet spot 相同的情況以及使用 cosine gamma function，在使用相同參數的 Transfomer 做預測，以下為實驗結果。

Total_Iter / Sweet Spot	FID
5	29.382404949147883
10	28.24679803225783
15	28.553048822219665
20	28.668546725285438

從實驗結果來看，以及考慮 FID 的誤差範圍，在本次作業中這些參數的設定不會太影響 FID，因此猜測主要影響因此還是在 Transformer 的訓練上。以下在兩種不同版本的 Transformer 下，total_iter 和 sweet spot 都為 10，gamma function 採用 cosine 的實驗結果。

<table>
<tr><th>Lr = 1e-3, batch=32 (min val:1.6151)</th><th>Lr =1e-4, batch=16 (min val:1.2472)</th></tr>
<tr><td>
</td><td>
</td></tr>
</table>

我們可以看到明顯會影響 FID 分數，這也是直覺的實驗結果，因為圖片的修補就是靠 Transformer 的預測結果而決定，loss 較低的模型，自然就會有較好的預測結果。

	FID
Lr = 1e-3, batch=32 (min val:1.6151)	64.0774478411227
Lr =1e-4, batch=16 (min val:1.2472)	28.24679803225783

3.2 The Influence of Gamma Function

以下實驗將實驗在同一個 Transformer 下，改變 gamma function 對於 FID 的影響，

由實驗結果可知，在本次作業中，Gamma Function 幾乎不影響 FID 分數。

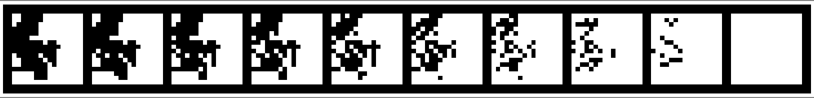
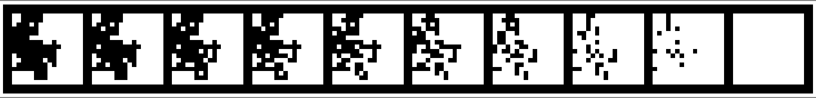
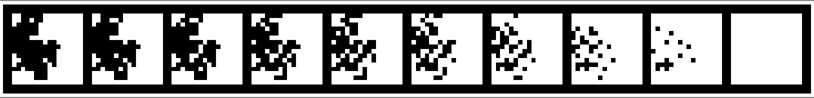
Gamma Function	FID
Linear	28.540681852585692
Cosine	28.330224449801534
Square	28.574741379456356

4. Experiment Score

4.1 Iterative Decoding

4.1.1 Mask in latent domain

Gamma	**Mask Scheduling**

Function	
Linear	
Cosine	
Square	

4.1.2 Predicted image

Gamma Function	Predicted image
Linear	
Cosine	
Square	

4.2 Best FID Score

4.2.1Training Hyperparameters

Epoch : 200

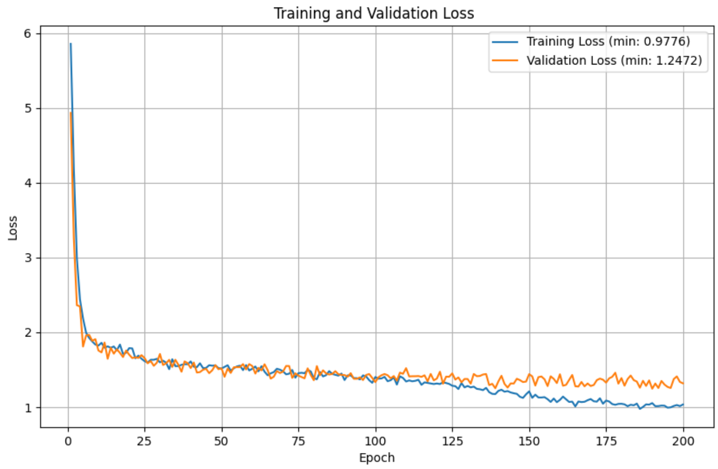
Batch Size : 16

Accum Grad : 5

Learning rate : 1e-4

Optimizer Adam with weight decay : 3e-5

Scheduler : LinearWarmUp and Cosine Annealing



Inpainting Hyperparameters

Total_Iter : 10

Sweet Spot : 10

Gamma Function : Cosine

4.2.2 Screenshot

```
(maskgit) sw710@Mochi:/mnt/e/School/Course/Summer-DLP/Lab3/faster-pytorch-fid$ python fid_score_gpu.py --predicted-path ../test_results/ --device cuda:0
747
100%|| 15/15 [00:01<00:00, 12.91it/s]
100%|| 15/15 [00:00<00:00, 21.27it/s]
FID:   28.624482904866483
```

4.2.3 Masked Images v.s MaskGIT Inpainting Results

Masked Image					
MaskGIT Inpainting Results