MaskGIT for Image Inpainting
Lab Report # 5

By
312581020
許瀚丰

Deep Learning
Spring 2024
Date Submitted: May 21, 2024

Contents

Listings

List of Tables

1 Introduction

1.1 Problem Statement

在本次實驗中，我們需要完成MaskGIT中對於Bidirectional Transformer的訓練，並利用其來解決Inpainting的任務。而在實作上，我們需要完成三個部分，分別Multi-head Attention Layer，對於Bidirectional Trasnformer的Training Strategy與最後用於Inpainting的Iterative Decoding。最後再比較不同的Decoding方式對於Inpainting後的結果是否有影響，並以FID作為衡量指標。

2 Implementation Details

2.1 The details of your model

2.1.1 Multi-Head Self-Attention

對於Multi-Head Self-Attention的部分，我們先計算出每個QKV的大小(768 / 16 = 48)，我們只需要使用一個Linear Layer，設定其輸入與輸出的維度是(768, 3 * num_heads * head_dim)，3代表之後要將輸出reshape成同樣大小並分配給QKV。而在Forward時，將其透過reshape與permute讓輸出的維度是(3, batch_size, num_heads, n, head_dim)，並將第一個dim分別分配給QKV即可之後就如原始計算Attention的作法，只是是針對不同的head各自計算。最後再透過一個Linear來將每個head計算出的結果做合併，如程式碼1所示。

```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.head_dim = dim // num_heads

        self.scale = self.head_dim ** -0.5
        self.to_qkv = nn.Linear(dim, 3 * self.num_heads * self.head_dim, bias=False)
        self.attn_drop = nn.Dropout(attn_drop)
        self.proj = nn.Linear(dim, dim)

    def forward(self, x):
```

```
        b, n, c = x.shape
        qkv = self.to_qkv(x)
        q, k, v = qkv.reshape(b, n, 3, self.num_heads, self.head_dim).permute(2, 0, 3, 1, 4)

        attn = (q @ k.transpose(-2, -1)) * self.scale
        attn = attn.softmax(dim=-1)
        attn = self.attn_drop(attn)

        o = (attn @ v).transpose(1, 2).reshape(b, n, c)
        o = self.proj(o)
        return o
```

Listing 1: Multi-Head Self-Attention

2.2 The details of your stage2 training

2.2.1 Basic Function

在實作中，有幾個function需要實作，分別為如何從VQGAN中獲得影像產生的token，與在inpainting時所使用的mask scheduling。

encode_to_z: 此function是用來將影像透過VQGAN產生成token，如程式碼2所示。

```
@torch.no_grad()
def encode_to_z(self, x):
    codebook_mapping, codebook_indices, _ = self.vqgan.encode(x)
    return codebook_mapping, codebook_indices.reshape(codebook_mapping.shape[0], -1)
```

Listing 2: encode_to_z

gamma_func: 此function會回傳要使用哪種mask scheduling，如程式碼3所示。

```
def gamma_func(self, mode="cosine"):
    if mode == "linear":
        return lambda r: 1 - r
    elif mode == "cosine":
        return lambda r: np.cos(r * np.pi / 2)
    elif mode == "square":
        return lambda r: 1 - r ** 2
    elif mode == "sqrt":
        return lambda r: 1 - np.sqrt(r)
```

```python
    elif mode == "constant":
        return lambda r: 1
    else:
        raise NotImplementedError
```

Listing 3: mask scheduling

2.2.2 MVTM

在MVTM的實作上，首先對於影像輸入x，我先將將其經過VQGAN獲得其token。接著sample一組binary mask(利用bernoulli來sample)，接著只需要將原始的token中binary mask是True的位置更改為mask_token，接著就將其經過Transformer後回傳模型預測的結果與原始的tokens即可，如程式碼4所示。

```python
def forward(self, x):
    # x: (b, c, h, w)
    _, z_indices = self.encode_to_z(x)

    mask_token = torch.ones(z_indices.shape, device=z_indices.device).long() * self.mask_token_id
    mask = torch.bernoulli(0.5 * torch.ones(z_indices.shape, device=z_indices.device)).bool()

    new_indices = mask * mask_token + (~mask) * z_indices
    logits = self.transformer(new_indices)
    z_indices=z_indices # ground truth
    logits = logits  # transformer predict the probability of tokens
    return logits, z_indices
```

Listing 4: MVTM

2.2.3 Forward and Loss

在Training的部分，其實可以將其視為一個簡單的分類的問題，我們所要預測的就是在MVTM中token被mask的位置原始的值，因此在Training時，對於影像資料x，我們先將其進行MVTM，將著就是將模型預測的結果與原始的token做Cross Entropy，以此來更新模型，如程式碼5所示。

```python
def train_one_epoch(self, train_loader, epoch, args):
    losses = []
```

```
    progress = tqdm(enumerate(train_loader))
    self.model.train()
    for i, x in progress:
        x = x.to(args.device)
        logits, z_indices = self.model(x)
        loss = F.cross_entropy(logits.reshape(-1, logits.size(-1)), z_indices.reshape
    (-1))
        loss.backward()
        losses.append(loss.item())
        if i % args.accum_grad == 0:
            self.optim.step()
            self.optim.zero_grad()
        progress.set_description_str(f"epoch: {epoch} / {args.epochs}, iter: {i} / {
    len(train_loader)}, loss: {np.mean(losses)}")
    self.writer.add_scalar("loss/train", np.mean(losses), epoch)
    return np.mean(losses)
```

Listing 5: Training

2.3 The details of your inference for inpainting task

在Inpainting的部分，對於每個iter，分別會輸入此iteration的token、此時的mask、一開始被mask的數量、目前的ratio與要使用何種mask fucnction。而在一開始，會先將token 經由輸入的mask將某些部分mask起來，並由transformer預測被mask起來的位置是何種數值的機率，之後會根據此機率去sample去一組token，並會透過ratio與mask function來決定此iter要留下的數值數量(此部分指的是原始mask為True的部分)，透過排序的方式去把confident較小的值繼續mask起來，最後就只需要回傳一開始sample的token與新的mask即可，如程式碼6所示。

```
@torch.no_grad()
def inpainting(self, z_indices, mask, mask_num, ratio, mask_func):
    z_indices_with_mask = mask * self.mask_token_id + (~mask) * z_indices
    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

    # 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)

    mask_ratio = self.gamma_func(mask_func)(ratio)

    mask_len = torch.floor(mask_num * mask_ratio).long()

    g = torch.distributions.gumbel.Gumbel(0, 1).sample(z_indices_predict_prob.shape).to(z_indices_predict_prob.device)
    temperature = self.choice_temperature * (1 - mask_ratio)
    confidence = z_indices_predict_prob + temperature * g
    sorted_confidence = torch.sort(confidence, dim=-1)[0]
    cut_off = sorted_confidence[:, mask_len].unsqueeze(-1)
    new_mask = (confidence < cut_off)
    return z_indices_predict, new_mask
```

Listing 6: Inpainting

3 Experimental results

3.1 The best testing fid

3.1.1 Screenshot

經過測試，當total-iter與sweet-spot皆為10，並使用cosine作為mask function時可以產生最好的FID，為**26.4074**，測試結果如表1。

FID


Table 1: The best testing FID

3.1.2 Predicted image, Mask in latent domain with mask scheduling

此時的Predict Image與Mask in latent domain如表2，我們可以觀察到Mask的位置會依照iter而逐漸變少，直至最後完全沒有mask，而Predict Image的部分也確實成功的將mask的部分給去除。

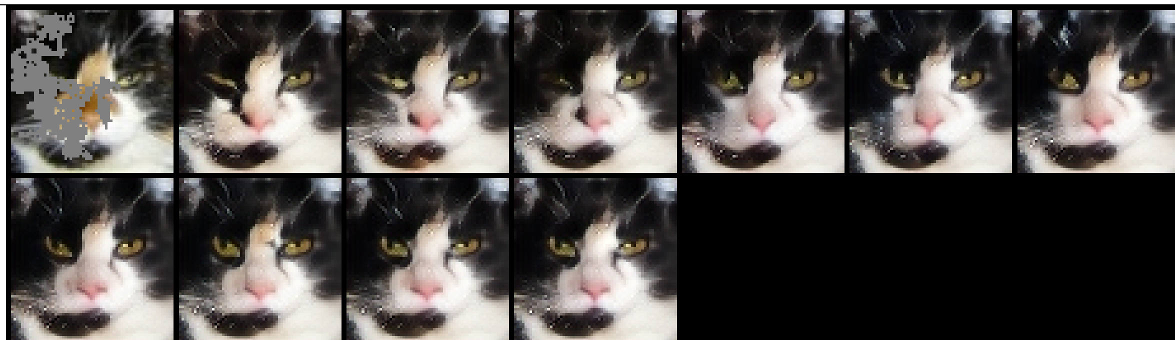
Image	
Mask	

Table 2: Result of Best FID

3.1.3 The setting about training strategy, mask scheduling parameters

經過測試，產生best FID的一些超參數設定如下：

- Learning Rate of training Transformer: $1e^{-4}$
- epoch of training Transformer: 200
- training strategy: mask about 50% image latent in training
- sweet-spot: 10
- total-iter: 10
- mask-func: cosine

3.2 Comparison figures with different mask scheduling parameters setting

不同mask scheduling的測試結果如表3，超參數的設定如section 3.1.3，只有改變mask-func的設定(cosine, linear, square)。從結果中我們可以觀察到Cosine與Square的在mask scheduling的結果較為類似，在前期的mask改變的較少，而到後期則會逐漸增加，而Linear則是每步都改變相同數量。

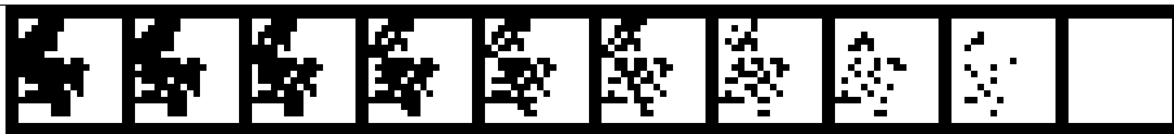
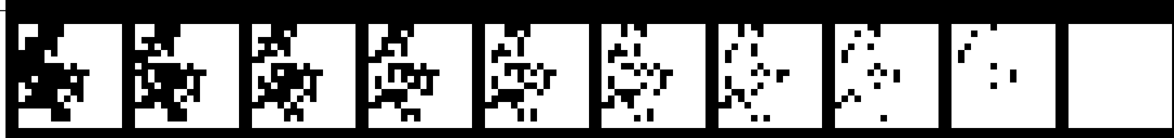
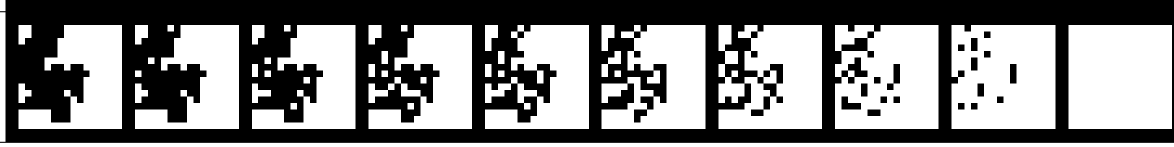
Cosine	
Linear	
Square	

Table 3: Mask scheduling comparison

FID的測試結果如表4，每個configurations皆執行30次inpainting的實驗並計算其FID的平均與標準差，可以觀察到Cosine的結果在平均上是最好的，Square則略差，而Linear則是表現最差。

Configurations	$FID \downarrow$
cosine	26.724 ± 0.267
linear	26.852 ± 0.293
square	26.757 ± 0.331

Table 4: FID comparison on different mask scheduling

4 Discussion

4.1 Details of training Transformer

在訓練Transformer時，我使用固定的Learning Rate為$1e^{-4}$，並訓練了300個epoch，

經過實驗發現約訓練到200 epoch後valid loss就開始上升，可見其已開始overfitting，因此在我所設定的超參數下，最好的model weights應該在epoch為200左右，如表5所示。

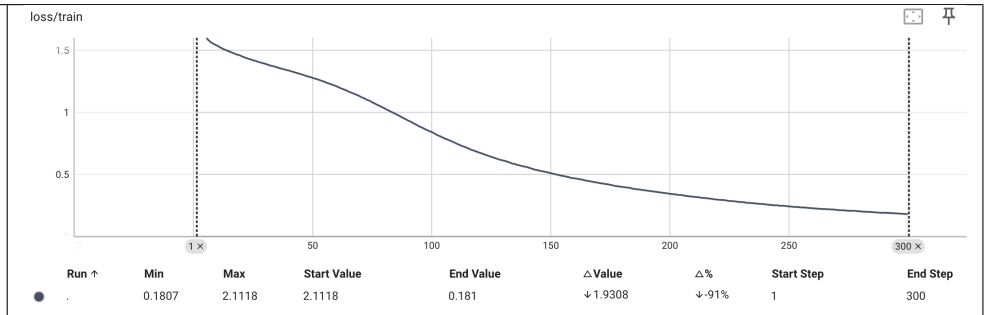
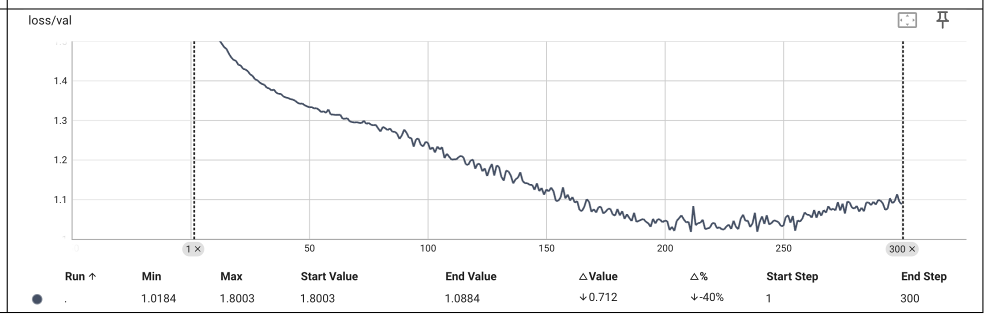
<table>
<tr><td>Training Loss</td><td>
</td></tr>
<tr><td>Valid Loss</td><td>
</td></tr>
</table>

Table 5: Loss curve of training transformer

4.2 Comparison of other mask strategies

在Mask Scheduling的方式上，除了原始的三種方法外，我另外比較了直接將結果一步到位產生(一開始就直接預測所有被mask的token)，與圖形為convex的Square Root，此方式在前期需保留較多mask的預測結果，後期則會逐漸減少。

4.2.1 Comparison on Difference Iteration

我們共比較了三種不同的iteration(T)，分別為8、10、12，我們可以觀察到在不同的iteration中，cosine的結果皆是最好的，而在T為10時與使用cosine時，則有最佳的結果，實驗結果如表6所示。

T	Configurations	$FID \downarrow$
-	Baseline	28.078 ± 0.254
8	Cosine	26.764 ± 0.343
	Linear	26.887 ± 0.310
	Square	26.787 ± 0.309
	Sqrt	27.023 ± 0.300
10	**Cosine**	26.724 ± 0.267
	Linear	26.852 ± 0.293
	Square	26.757 ± 0.331
	Sqrt	26.935 ± 0.305
12	Cosine	26.746 ± 0.368
	Linear	26.840 ± 0.244
	Square	26.807 ± 0.367
	Sqrt	26.905 ± 0.306

Table 6: FID comparison on different iteration

另外，我也嘗試比較了固定iteration，並設定不同的sweet spot($Stop$)，實驗結果如表7。我們可以觀察到最好的結果是發生在sweet spot為10，而total iteration為12，一樣是使用Cosine。另外我們也可以發現，在sweet spot較小時，使用Linear與Sqrt的結果反而比Cosine與Square好，原因是因為前兩者在前期會保留較多預測的結果，而後兩者則是後期才會逐漸增加，因此才會有此現象。

T	$Stop$	Configurations	$FID \downarrow$
-	-	Baseline	28.078 ± 0.254
12	8	Cosine	27.020 ± 0.328
		Linear	26.876 ± 0.281
		Square	27.042 ± 0.229
		Sqrt	26.978 ± 0.264
12	10	**Cosine**	26.803 ± 0.346
		Linear	26.858 ± 0.266
		Square	26.870 ± 0.288
		Sqrt	27.059 ± 0.270

Table 7: FID comparison on different sweet spot