DLP Lab5

Introduction

In this lab, we are reconstructing masked images with MaskGIT. Our implementation focus is on implementing multi-head attention, transformer training, and inference inpainting. Additionally, we can experiment with different mask scheduling parameters to compare their impact on inpainting results.

Implementation Details

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

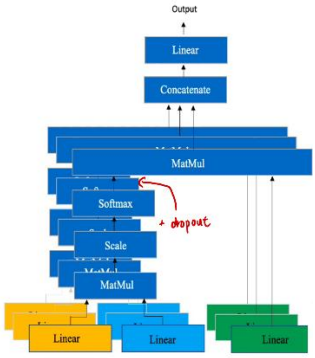
Here I set 3 networks for q, k, and v respectively. I also add the dropout layer since it was passed in the parameters, and I compute the dimension of each head here.

```python
class MultiHeadAttention(nn.Module):
    def __init__(self, dim=768, num_heads=16, attn_drop=0.1):
        super(MultiHeadAttention, self).__init__()
        self.dim = dim
        self.num_heads = num_heads
        self.head_dim = dim // num_heads

        # QKV linear layers, will be split into num_heads later
        self.q_linear = nn.Linear(dim, dim)
        self.k_linear = nn.Linear(dim, dim)
        self.v_linear = nn.Linear(dim, dim)

        # Output linear layer
        self.out_linear = nn.Linear(dim, dim)

        # Dropout layer
        self.attn_drop = nn.Dropout(attn_drop)
```

For the forward process, I follow the description in the transformer slide. The only thing worth mentioning is (1) I reshape and transpose the dimensions to divide heads (2) add dropout after softmax. (details in comment)

```python
    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_image_tokens, _ = x.shape

        # get q, k, v matrices
        q = self.q_linear(x)
        k = self.k_linear(x)
        v = self.v_linear(x)

        # reshape to split by heads and then transpose to get the shape (batch_size, num_heads, num_image_tokens, head_dim)
        q = q.view(batch_size, num_image_tokens, self.num_heads, self.head_dim).transpose(1, 2)
        k = k.view(batch_size, num_image_tokens, self.num_heads, self.head_dim).transpose(1, 2)
        v = v.view(batch_size, num_image_tokens, self.num_heads, self.head_dim).transpose(1, 2)

        # do matrix multiplication for each head
        # q dimension: (batch_size, num_heads, num_image_tokens, head_dim)
        # k dimension: (batch_size, num_heads, head_dim, num_image_tokens)
        # scores dimension: (batch_size, num_heads, num_image_tokens, num_image_tokens)
        # stabilize the gradients by scaling
        scores = torch.matmul(q, k.transpose(-2, -1)) / (self.head_dim ** 0.5)

        attn_weights = torch.nn.functional.softmax(scores, dim=-1)
        attn_weights = self.attn_drop(attn_weights)

        # attn_weights dimension: (batch_size, num_heads, num_image_tokens, num_image_tokens)
        # v dimension: (batch_size, num_heads, num_image_tokens, head_dim)
        attn_output = torch.matmul(attn_weights, v)

        # concatenate heads and apply the final linear layer
        attn_output = attn_output.transpose(1, 2).reshape(batch_size, num_image_tokens, self.dim)
        output = self.out_linear(attn_output)

        return output
```

I was thinking whether I should add dropout right before the final linear layer or add at the place where I add it now. I think it will have stronger effect if I add it right after the softmax layers, so I add it here directly, and the results are good enough.



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

I first encode the image to get the key indeces of z_q and reshape it. Then I generate a ratio for the mask uniformly between 0 and 0.5 for the mask, and generate the random map. I assign value 1024 to the masked token and return the transformer prediction and original unmasked indeces.

```
##TODO2 step1-3:
    def forward(self, x):
        '''
        mask ratio can be adjusted
        '''
        # z_indices=None #ground truth
        _, z_indeces = self.encode_to_z(x)

        # reshape z_indeces to (batch_size, num_image_tokens)
        z_indeces = z_indeces.view(-1, self.num_image_tokens)

        #print(z_indeces.shape)
        ratio = torch.rand(1)
        ratio = (ratio * 0.5).item()
        mask = torch.rand_like(z_indeces, dtype=float) < ratio
        # assign value 1024 to the masked token
        masked_z = torch.where(mask, 1024, z_indeces)
        logits = self.transformer(masked_z)
        return logits, z_indeces
```

We want the transformer to predicit the logits similar to original z_indeces, so I use nn.CrossEntropyLoss for this.

```
    self.loss_fn = nn.CrossEntropyLoss()
```

```
def train_one_epoch(self, data_loader):
    total_loss = 0
    for imgs in tqdm(data_loader):
        self.optim.zero_grad()
        x = imgs.to(args.device)
        logits, z_indeces = self.model(x)
        z_indeces = F.one_hot(z_indeces.view(-1), num_classes=logits.size(-1)).float()
        loss = self.loss_fn(logits.view(-1, logits.size(-1)), z_indeces)
        total_loss += loss.item()
        loss.backward()
        self.optim.step()
```

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

Here I implemented three types of gamma functions and return as the form of functions. The formula is based on the graph in the spec.

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

For one iteration of decoding, I select the best n tokens and add it to the output, where n is (the total number of mask * ratio generated by gamma function). The last two line fill in the original values if the original mask is false.

```python
##TODO3 step1-1: define one iteration decoding
    @torch.no_grad()
    def inpainting(self, z_indeces, mask, ratio, mask_num):
        masked_z = torch.where(mask, 1024, z_indeces)

        logits = self.transformer(masked_z)
        # get softmax probability
        logits = torch.nn.functional.softmax(logits, dim=-1).squeeze(0)

        # find max probability tokens
        z_indeces_predict_prob, z_indeces_predict = torch.max(logits, dim=-1)

        # add gumbel noise to the confidence
        g =  -torch.empty_like(z_indeces_predict_prob).exponential_().log()
        temperature = self.choice_temperature * (1 - ratio)
        confidence = z_indeces_predict_prob  + temperature * g

        # find the threshold of confidence
        new_confidence = torch.where(mask, confidence, float('inf'))
        z_sort = torch.argsort(new_confidence, descending=False)
        n = ratio * mask_num
        confidence_threshold = new_confidence[0, z_sort[0,int(n)]]

        # put the original token values back if its on the mask
        z_indeces_predict = torch.where(mask, z_indeces_predict, z_indeces)
        mask_bc = new_confidence < confidence_threshold
        return z_indeces_predict.squeeze(0), mask_bc
```

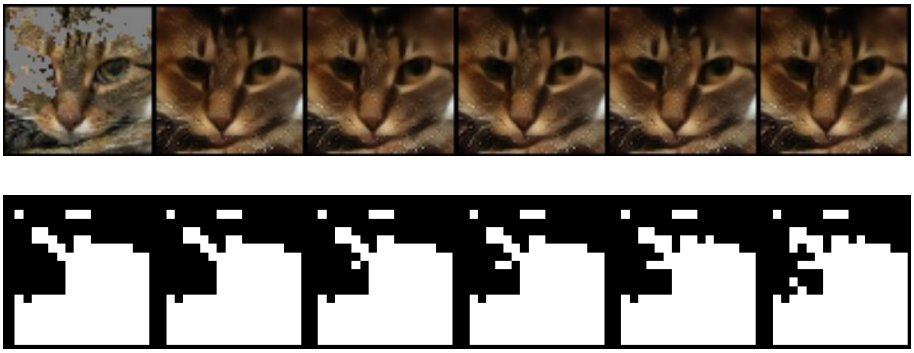
Experimental results

A. The best testing fid

- Screenshot

```
(temp) user@user-ESC8000-G4:~/Shen_Wu/stanley/DLP5/code/faster-pytorch-fid$ python fid
_score_gpu.py --predicted-path ../test_results --device cuda:0
747
100%|                                                    | 15/15 [00:02<00:00,  6.62it/s]
100%|                                                    | 15/15 [00:01<00:00,  8.60it/s]
FID:  39.390187292708816
```

• Predicted image, Mask in latent domain with mask scheduling



• The setting about training strategy, mask scheduling parameters

Batch-size 32

Epochs 50

Lr 1e-4

Betas (0.9, 0.999)

Weight-decay 2e-3

Optimizer adamW

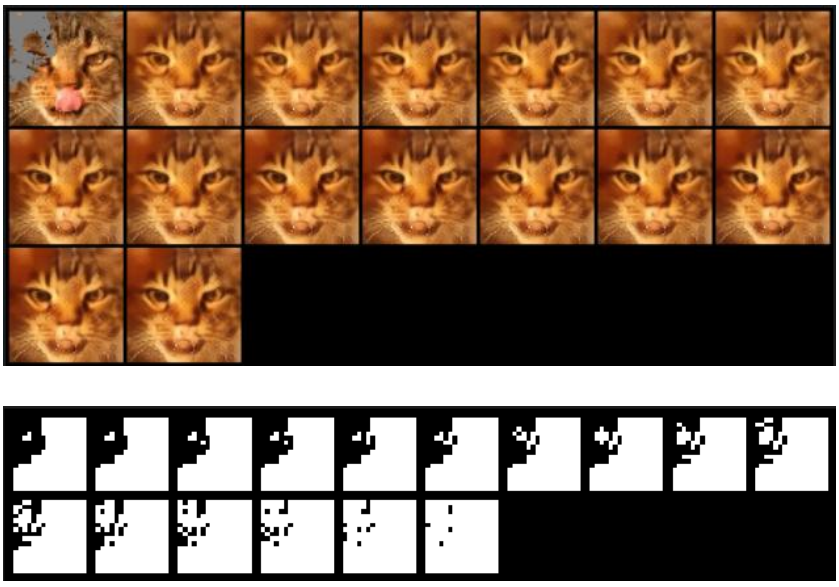
Scheduler milestone [2, 10, 40], gamma 0.3

Mask ratio [0, 0.5] uniformly.

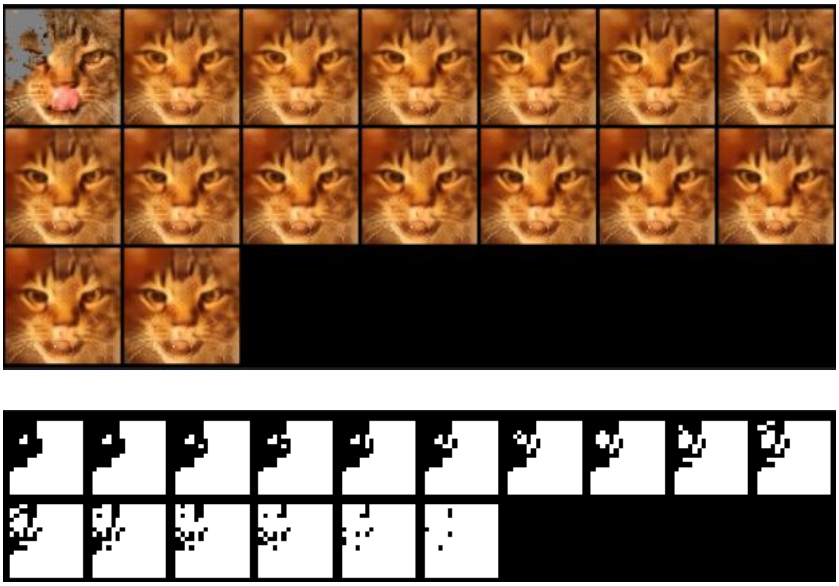
Inpainting sweet spot 5, total iter 15, mask function cosine

B. Comparison figures with different mask scheduling parameters setting

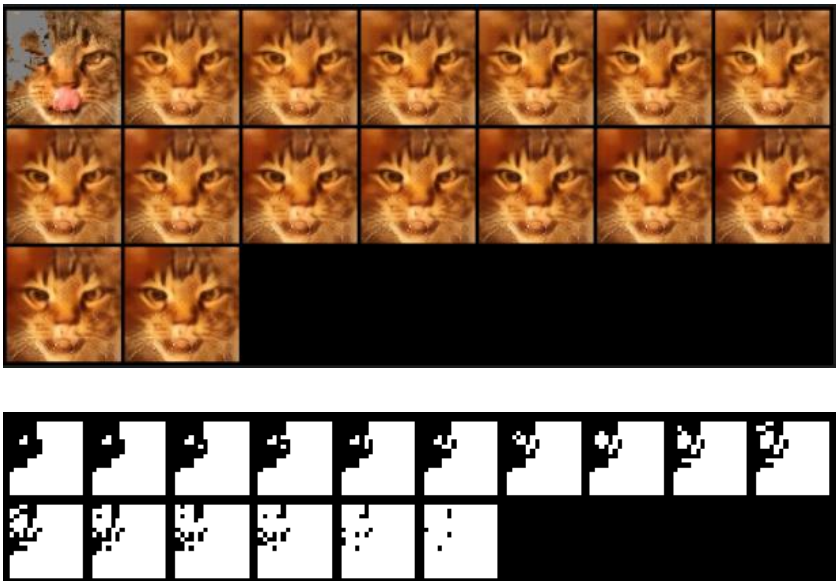
• cosine



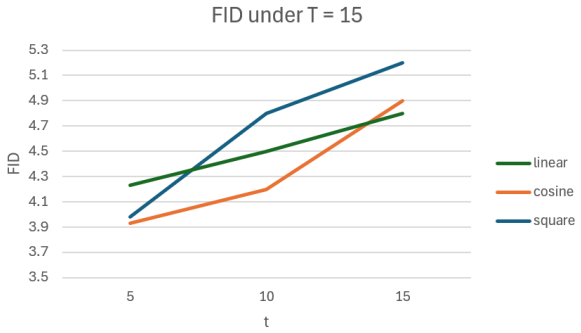
• linear



• square



• graph



Discussion

I would like to discuss the mask scheduling in training stage. I used an random mask for tokens randomly in training stage, but the mask in the testing dataset is dense (regional). This makes the noise uniform distributed on the whole image not good enough. I tried using 8*8 mask with probability 0.8 being mask to simulate the dense make, but it turned out that it performs worse than the original mask scheduling approach.