Deep Learning Lab 3 - MaskGIT for Image Inpainting

110550088 李杰穎

April 1, 2025

1 Introduction

In this lab assignment, we will implement MaskGIT[1] for image inpainting. MaskGIT is a generative model that can generate images from a given mask. The model is trained to predict the masked pixels based on the unmasked pixels. The model is based on the bi-directional Transformer[2] architecture and uses a masked language modeling objective.

The model is trained on a cat datasets, which contains images of cats with random masks applied. The model is trained to predict the masked pixels based on the unmasked pixels. The model is evaluated on a test set of images with random masks applied.

I experiment with different mask scheduling strategies when inferencing the model, and evaluate the performance of the model by the FID score.

2 Implementation Detail

2.1 Multi-Head Attention

As described in specification, the Multi-Head Attention module is implemented as a PyTorch module. The input to the module is a tensor of shape (batch_size, num_image_tokens, dim), where batch_size is the number of images in a batch, num_image_tokens is the number of image tokens, and dim is the dimension of the tokens. The output of the module is a tensor of the same shape.

The module consists of four linear layers: one for the query, one for the key, one for the value, and one for the output. The input tensor is passed through the query, key, and value linear layers to obtain the query, key, and value tensors. The query, key, and value tensors are then reshaped to (batch_size, num_image_tokens, num_heads, head_dim) and transposed to (batch_size, num_heads, num_image_tokens, head_dim).

The attention scores are computed by taking the dot product of the query and key tensors, scaling the scores by the square root of the head dimension, and applying the softmax function. The attention scores are then multiplied by the value tensor to obtain the attention output. The attention output is reshaped back to (batch_size, num_image_tokens, dim) and passed through the output linear layer to obtain the final output.

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

        self.linear_q = nn.Linear(dim, dim)
        self.linear_k = nn.Linear(dim, dim)
        self.linear_v = nn.Linear(dim, dim)
        self.linear_out = nn.Linear(dim, dim)

        self.attn_drop = attn_drop
        self.dropout = nn.Dropout(p=attn_drop)

        self.softmax = nn.Softmax(dim=-1)
        self.scale = 1. / math.sqrt(self.d_k)

    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, dim = x.size()
        assert dim == self.dim, f"Expected input dimension {self.dim}, but got {dim}"
        query = self.linear_q(x).view(
            batch_size, num_image_tokens, self.num_heads, self.head_dim).transpose(1, 2)
        key = self.linear_k(x).view(
            batch_size, num_image_tokens, self.num_heads, self.head_dim).transpose(1, 2)
        value = self.linear_v(x).view(
            batch_size, num_image_tokens, self.num_heads, self.head_dim).transpose(1, 2)
        # Reshape to (batch_size, num_heads, num_image_tokens, head_dim)
        #      key^T (batch_size, num_heads, head_dim, num_image_tokens)
        # Compute attention scores
        attn_scores = torch.matmul(query, key.transpose(-2, -1)) * self.scale
        attn_scores = self.softmax(attn_scores)
        # (batch_size, num_heads, num_image_tokens, num_image_tokens)
```

Code 1: **Implementation of Multi-Head Attention.**

```
        attn_scores = self.dropout(attn_scores)
        # Compute attention output
        attn_output = torch.matmul(attn_scores, value)
        attn_output = attn_output.transpose(1, 2).contiguous().view(
            batch_size, num_image_tokens, self.dim)
        attn_output = self.linear_out(attn_output)
        return attn_output
```

2.2 Masked Visual Token Modeling (MVTM) Training

The MVTM training process follows a similar approach to masked language modeling, but applied to image tokens. The implementation involves several key components:

Codebook encoding. The `encode_to_z` function takes an input image and passes it through the VQGAN encoder to obtain tokens. The function returns both the codebook mapping and the reshaped codebook indices. This encoding process transforms input images of shape $(B, 3, 64, 64)$ into a sequence of discrete tokens of shape $(B, 256)$, where 256 is the number of tokens (16×16) in the latent space.

Code 2: **Implementation of encode_to_z function.**

```
@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)
```

Mask scheduling. The mask scheduling is controlled by the `gamma_func`, which determines how the mask ratio evolves during the inference process. Three different scheduling functions are implemented:

- **Linear**: $\gamma(x) = 1-x$, a straight-line decrease in mask ratio
- **Cosine**: $\gamma(x) = \cos(\frac{\pi x}{2})$, which decreases slower initially and faster later
- **Square**: $\gamma(x) = 1 - x^2$, which decreases slower initially and faster later

These functions map a ratio $x \in [0, 1)$ to a masking ratio $\gamma(x) \in (0, 1]$, controlling how many tokens remain masked at each decoding step.

Code 3: **Implementation of gamma function for mask scheduling.**

```
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":
        def linear_func(x):
            return 1 - x
        return linear_func
    elif mode == "cosine":
        def cosine_func(x):
            return np.cos(np.pi * x / 2)
        return cosine_func
    elif mode == "square":
        def square_func(x):
            return 1 - x ** 2
        return square_func
    else:
        raise NotImplementedError
```

The foward function. The `forward` function implements the core MVTM training procedure:

1. Encode the input image to obtain tokens (`z_indices`)
2. Create a tensor of mask tokens with the same shape as `z_indices`
3. Generate a random mask by sampling from a Bernoulli distribution with $p = 0.5$
4. Replace the masked positions in `z_indices` with mask tokens
5. Pass the masked tokens through the transformer to predict the probabilities
6. Return the predicted logits and the original tokens

During training, the model learns to predict the original tokens from the masked sequence. The loss function is cross-entropy between the predicted token probabilities and the ground truth tokens, ignoring the mask token ID.

Code 4: **Implementation of forward function.**

```
def forward(self, x):
    _, z_indices = self.encode_to_z(x)  # ground truth
    mask_tokens = torch.full_like(
        z_indices, self.mask_token_id)  # mask token
    mask = torch.bernoulli(torch.full(
        z_indices.shape, 0.5)).bool()  # mask ratio

    new_z_indices = z_indices.clone()
    new_z_indices[mask] = mask_tokens[mask]

    # transformer predict the probability of tokens
```

```
    logits = self.transformer(new_z_indices)
    return logits, z_indices
```

Training loop and loss function. The training loop consists of the following steps:

Code 5: **Implementation of the training loop.**

```
def train_one_epoch(self, train_dataloader, epoch, args):
    losses = []
    pbar = tqdm(train_dataloader, desc=f"Epoch {epoch}/{args.epochs}", unit="batch")
    self.model.train()
    for batch_idx, (images) in enumerate(pbar):
        x = images.to(args.device)
        logits, z_indices = self.model.forward(x)
        loss = F.cross_entropy(logits.view(-1, logits.size(-1)), z_indices.view(-1), ignore_index=self.model.mask_token_id)
        loss.backward()
        losses.append(loss.item())
        if (batch_idx + 1) % args.accum_grad == 0:
            self.optim.step()
            self.optim.zero_grad()
        pbar.set_postfix(loss=loss.item(), lr=self.optim.param_groups[0]['lr'])

        # Log batch loss to wandb
        if args.use_wandb and batch_idx % args.wandb_log_interval == 0:
            wandb.log({
                "batch": batch_idx + epoch * len(train_dataloader),
                "train_batch_loss": loss.item(),
                "learning_rate": self.optim.param_groups[0]['lr']
            })

    avg_loss = np.mean(losses)
    print(f"Epoch {epoch}/{args.epochs}, Average Loss: {avg_loss:.4f}")
    if self.scheduler is not None:
        self.scheduler.step()
    return avg_loss
```

The training process for the bidirectional transformer in MaskGIT follows an efficient approach. For each batch of images, we first obtain the token representations using the `encode_to_z` function. Then the model applies random masking with a probability of 0.5 and processes these masked tokens through the transformer to predict the original tokens.

The loss function is implemented as cross-entropy loss between the predicted logits and ground truth tokens:

$$\mathcal{L}_{\text{CE}} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{c=1}^{C} y_{i,c}\log(p_{i,c}) \tag{1}$$

where N is the number of unmasked tokens, C is the number of classes (codebook size), $y_{i,c}$ is a binary indicator if class c is the correct classification for token i, and $p_{i,c}$ is the predicted probability that token i belongs to class c.

Importantly, the loss ignores positions where the ground truth is the mask token ID, ensuring that the model only learns to predict the actual content tokens.

The implementation uses gradient accumulation through the `accum_grad` parameter, which allows for effective training with larger batch sizes by accumulating gradients over multiple forward and backward passes before performing an optimizer step.

Noted that I also integrate wandb for logging the training process.

Validation loop. The validation loop is similar to the training loop. It evaluates the model's performance on a validation dataset and computes the average loss.

Code 6: **Implementation of validation loop.**

```
def eval_one_epoch(self, val_dataloader, epoch, args):
    self.model.eval()
    losses = []
    with torch.no_grad():
        for batch_idx, (images) in enumerate(val_dataloader):
            x = images.to(self.device)
            logits, z_indices = self.model.forward(x)
            loss = F.cross_entropy(logits.view(-1, logits.size(-1)), z_indices.view(-1), ignore_index=self.model.mask_token_id)
            losses.append(loss.item())
    avg_loss = np.mean(losses)
    print(f"Validation Loss: {avg_loss:.4f}")
    return avg_loss
```

The model checkpoints are saved based on both training and validation performance, with the best models preserved for later use in the inpainting task.

2.3 Inpainting

The inpainting process in MaskGIT follows an iterative decoding strategy. The implementation consists of several key components:

Code 7: **Implementation of inpainting function.**

```
def inpainting(self, image, mask_b, i):
    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))
        #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)
        ratio = 0

        for step in range(self.total_iter):
            if step == self.sweet_spot:
                break

            ratio = (step + 1) / self.total_iter
            z_indices_predict, mask_bc = self.model.inpainting(z_indices_predict, mask_bc, mask_num, ratio, self.mask_func)

            # Visualization steps for mask and decoded image
            mask_i = mask_bc.view(1, 16, 16)
            mask_image = torch.ones(3, 16, 16)
            indices = torch.nonzero(mask_i, as_tuple=False)
            mask_image[:, indices[:, 1], indices[:, 2]] = 0
            maska[step] = mask_image

            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
```

The main `inpainting` function contains the iterative decoding process. It first initializes tensors to store the masks and decoded images at each iteration. Then, it encodes the image to tokens using the VQGAN encoder and counts the number of masked tokens. For each iteration, it calculates the current ratio based on the step number and total iterations, then calls the model's inpainting method to update the predicted tokens and mask.

Code 8: **Implementation of model's inpainting function.**

```
@torch.no_grad()
def inpainting(self, z_indices, mask, mask_num, ratio, mask_func):
    masked_z_indices = z_indices.clone()
    masked_z_indices[mask] = self.mask_token_id

    logits = self.transformer(masked_z_indices) # B x num_image_tokens x num_codebook_vectors
    probs = logits.softmax(dim=-1)
    z_indices_predict = torch.distributions.Categorical(logits=logits).sample()

    while torch.any(z_indices_predict == self.mask_token_id):
        z_indices_predict = torch.distributions.Categorical(logits=logits).sample()

    z_indices_predict[~mask] = z_indices[~mask]
    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.full_like(z_indices_predict_prob, float('inf')))

    mask_ratio = self.gamma_func(mask_func)(ratio)
    print(f"mask ratio: {mask_ratio}")
    mask_len = int(mask_num * mask_ratio)

    g = torch.distributions.Gumbel(0, 1).sample(z_indices_predict_prob.shape).to(z_indices_predict_prob.device)
    temperature = self.choice_temperature * (1 - ratio)
    confidence = z_indices_predict_prob + temperature * g
    sorted_confidence, _ = torch.sort(confidence, dim=-1)
    threshold = sorted_confidence[:, mask_len].unsqueeze(-1)
    mask_bc = confidence < threshold

    return z_indices_predict, mask_bc
```

The model's `inpainting` method implements the token prediction and confidence calculation for each iteration:

1. First, it creates a masked version of the tokens, replacing masked positions with the mask token ID
2. It passes these tokens through the transformer to predict probabilities and samples from this distribution
3. The original tokens are kept for unmasked positions
4. It calculates the mask ratio using the selected gamma function (linear, cosine, or square)
5. To determine which tokens to unmask, it adds Gumbel noise to the prediction probabilities to obtain confidence scores
6. It sorts the confidence scores and finds a threshold such that the number of tokens with confidence below the threshold equals the desired mask length
7. Tokens with confidence below this threshold remain masked in the next iteration

The temperature parameter controls the amount of noise added to the prediction probabilities, which decreases as the ratio increases. This helps to balance exploration and exploitation during the decoding process.

After each iteration, the current mask and decoded image are saved for visualization. The decoded image at the sweet spot (the optimal iteration) is saved as the final inpainting result.

Code 9: **Saving visualization and final 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)
```

The visualization shows how the mask evolves over iterations and how the image is progressively filled in. The final result is stored in the `test_results` folder, which is then used for FID score calculation.

The mask scheduling function plays a crucial role in the inpainting performance. Different functions (linear, cosine, square) affect how quickly the mask is reduced and can lead to different quality results.

3 Discussion

Sweet spots. After reading the original MaskGIT[1] paper, I found that the definition of sweet spot is the best total iteration number for inpainting, not the best step to terminate the iterative decoding process. Therefore, in the later experiments, I set the `sweet_spot` to -1, which means that the iterative decoding process will run for the total iteration number.

4 Experiment

4.1 Iterative decoding

The iterative decoding process is a key component of the inpainting task. The model generates images by iteratively refining the masked tokens.

Total iteration numbers. I first evaluate of different total iteration numbers that yields the best inpainting results. In this experiments, I set the `mask_func` to `cosine` and the `sweet_spot` to -1, which means that the iterative decoding process will run for the total iteration number. I test the total iteration number from 1 to 28, and the results are shown in Figure 1.

The results show that initially, the FID score decreases as the total iteration number increases, indicating that the model generates better images with more iterations. However, after the model achieves the lowest FID score with 30.56 at iteration number is 8, the FID doesn't decrease as before. This observation shows that the model is converged when we set the total number of iterations to 8. This is coincided with the observation stated in the original paper, with the sweet spot is between 8 and 12. we can also observe that the generation time increases linearly with the total iteration number, which is expected since more iterations require more computation.

In the later experiments, we will set the total iteration number to 8, which is the sweet spot for inpainting.

Mask scheduling. The mask scheduling function is another important factor that affects the inpainting results. I test three different mask scheduling functions: linear, cosine, and square. The results are shown in **??**.

The choice of mask scheduling function significantly impacts the quality of the generated images.

The three functions (linear, cosine, square) were tested to observe their effects on the inpainting results. The linear function provides a straightforward approach, while the cosine and square functions introduce more complex dynamics in the mask reduction process.

4.2 Transformer training

Hyperparameters. The hyperparameters for the training process are as follows:

- **Batch size**: 40
- **Image size**: 64x64
- **Number of heads**: 16
- **Gradient accumulation steps**: 5
- **Optimizer**: AdamW
- **Learning rate schedule**: CosineAnnealing
- **Learning rate**: 5e-5
- **Weight decay**: 0.01
- **Epochs**: 150
- **Mask ratio**: 0.5
- **Mask token ID**: 1024

References

[1] Huiwen Chang et al. "MaskGIT: Masked Generative Image Transformer". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2022.

[2] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).

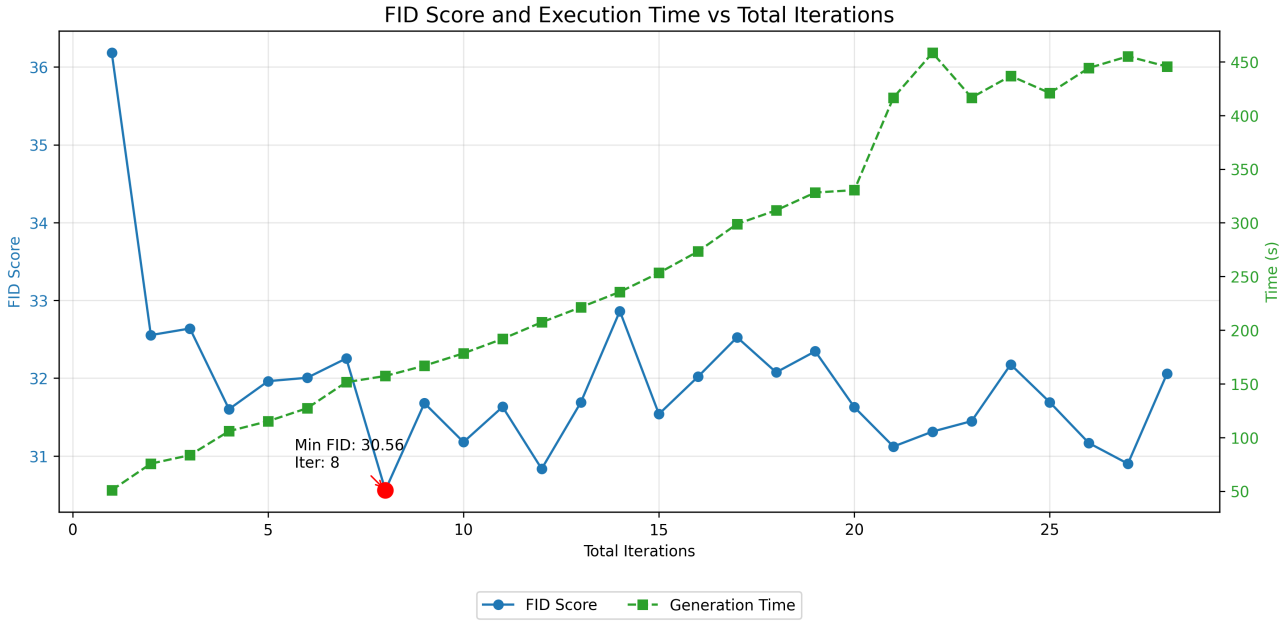

Figure 1: **The FID score and generation time of different total iteration numbers.**