Lab 3: MaskGIT for Image Inpainting

110550095 王皓平

1 Introduction

Image inpainting is a critical task in computer vision that involves reconstructing missing or damaged parts of an image. The MaskGIT (Masked Generative Image Transformer) approach introduces a novel method for efficiently generating high-quality image completions using a bidirectional transformer.

Key contributions of MaskGIT include:

- Employing a bidirectional transformer for faster token generation
- Implementing Masked Visual Token Modeling (MVTM)
- Achieving parallel decoding through iterative mask scheduling

2 Implementation Details

2.1 Multi-Head Self-Attention Module

Multi-Head Self-Attention allows each head to learn different relationships within the sequence, effectively giving the model a richer representation. The procedure involves projecting the sequence into multiple Query/Key/Value sets, computing scaled dot-product attention per head, and then concatenating and projecting the result.

```
def forward(self, x):
    batch_size, seq_len, dim = x.size()
    q = self.q_proj(x)  # (batch_size, seq_len, dim)
    k = self.k_proj(x)  # (batch_size, seq_len, dim)
    v = self.v_proj(x)  # (batch_size, seq_len, dim)
    q = q.view(batch_size, seq_len, self.num_heads, self.head_dim).transpose(1, 2)
    k = k.view(batch_size, seq_len, self.num_heads, self.head_dim).transpose(1, 2)
    v = v.view(batch_size, seq_len, self.num_heads, self.head_dim).transpose(1, 2)

    attn_scores = (q @ k.transpose(-2, -1)) / self.scale
    attn_probs = torch.softmax(attn_scores, dim=-1)
    attn_probs = self.attn_drop(attn_probs)

    context = (attn_probs @ v).transpose(1, 2).contiguous()
    context = context.view(batch_size, seq_len, dim)
    output = self.out_proj(context)  # (batch_size, seq_len, dim)

    return output
```

- **Initialization**:

  We created linear projection layers for query(Q), key(K), and value(V), as well as an output projection layer. Parameters are set with dimension 768, 16 heads, and 48 dimensions per head.

- **Forward Pass Process**:

  First, the input is linearly projected to obtain Q, K, V matrices The tensors are reshaped into multi-head format and transposed for batch matrix multiplication Attention scores are calculated where d_k is the dimension per head Softmax is applied to obtain attention weights Attention weights are applied to values(V) Multi-head outputs are reshaped and merged The final result is obtained through the output projection layer

2.2 Stage 2 Training

```python
def forward(self, x):
    batch_size = x.size(0)
    z_indices=self.encode_to_z(x) #ground truth

    # decide number of token to be masked
    mask_ratio = torch.rand(1).item()
    num_masked = int(self.num_image_tokens * self.gamma(mask_ratio))

    mask = torch.zeros_like(z_indices, dtype=torch.bool)

    for i in range(batch_size):
        mask_indices = torch.randperm(
            self.num_image_tokens,
            device=z_indices.device
        )[:num_masked]
        for idx in mask_indices:
            mask[i, idx] = True

    z_indices_masked = z_indices.clone()
    z_indices_masked[mask] = self.mask_token_id

    #transformer predict the probability of tokens
    logits = self.transformer(z_indices_masked)
    return logits, z_indices
```

```python
def train_one_epoch(self, train_loader, epoch):
    self.model.train()
    total_loss = 0
    with tqdm(total=len(train_loader), desc=f'Train epoch:{epoch}') as pbar:
        for i, imgs in enumerate(train_loader):
            imgs = imgs.to(self.device)
            logits, target = self.model(imgs)  #VQGANTransformer

            loss = F.cross_entropy(
                logits.reshape(-1, logits.size(-1)),
                target.reshape(-1),
            )
            loss = loss / self.args.accum_grad
            loss.backward()

            if (i + 1) % self.args.accum_grad == 0 or (i + 1) == len(train_loader):
                self.optim.step()
                self.optim.zero_grad()

            total_loss += loss.item() * self.args.accum_grad
            pbar.set_postfix(loss=total_loss / (i + 1))
            pbar.update(1)

    avg_loss = total_loss / len(train_loader)
    return avg_loss
```

- **Discrete Token Extraction**:

  Using the pretrained VQGAN encoder and quantizer to convert input images into discrete tokens.

- **Masking Strategy**:

  Using random mask ratio during training, rather than following a specific iterative mask schedule. Randomly selecting positions to mask and replacing these positions with a special mask token

- **Forward Pass and Loss Calculation**:

  Inputting the masked sequence into the Transformer for prediction. Calculating cross-entropy loss only at masked positions so that the model learns to predict masked tokens based on context

This training method, known as Masked Visual Token Modeling (MVTM), is very similar to masked language modeling in BERT but applied to the visual token domain. It allows the model to learn bidirectional contextual relationships between tokens, which is particularly important for image inpainting tasks.

2.3 Inference for Image Inpainting

```python
def inpainting(self,image,mask_b,i): #MakGIT inference
    ...
    self.model.eval()
    with torch.no_grad():
        z_indices = self.model.encode_to_z(image)
        mask_num = mask_b.sum() #total number of mask token
        z_indices_predict=z_indices.clone()
        mask_bc=mask_b.clone()
        mask_b=mask_b.to(device=self.device)
        mask_bc=mask_bc.to(device=self.device)
        z_indices_predict[mask_b] = self.model.mask_token_id
        ratio = 0
        #iterative decoding for loop design
        for step in range(self.total_iter):
            if step == self.sweet_spot:
                break
            ratio = step / self.total_iter
            z_indices_predict, mask_bc = self.model.inpainting(
                z_indices_predict,
                mask_bc,
                ratio
            )
            remaining_masks = (z_indices_predict == self.model.mask_token_id)

            if step == self.sweet_spot - 1 and remaining_masks.sum() > 0:
                logits = self.model.transformer(z_indices_predict)
                logits = torch.softmax(logits, dim=-1)
                logits[:, :, self.model.mask_token_id] = -float('inf')
                _, best_tokens = torch.max(logits, dim=-1)
                z_indices_predict[remaining_masks] = best_tokens[remaining_masks]
                mask_bc.fill_(False)

            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)
            max_idx = self.model.vqgan.codebook.embedding.weight.size(0) - 1
            safe_indices = torch.clamp(z_indices_predict, 0, max_idx)
            z_q = self.model.vqgan.codebook.embedding(safe_indices).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 #get decoded image
    ...
```

```python
def inpainting(self, z_indices_predict=None, mask_bc=None, ratio=0.0, mask_num=None):
    if z_indices_predict is None:
        z_indices_predict = torch.full(
            (1, self.num_image_tokens),
            self.mask_token_id,
            dtype=torch.long,
            device=self.transformer.device
        )
    if mask_bc is None:
        mask_bc = torch.ones_like(z_indices_predict, dtype=torch.bool)

    device = z_indices_predict.device
    z_indices_predict = z_indices_predict.to(device)
    mask_bc = mask_bc.to(device)

    try:
        logits = self.transformer(z_indices_predict)
        logits = torch.softmax(logits, dim=-1)

        #FIND MAX probability for each token value
        z_indices_predict_prob, z_indices_predict_candidate = torch.max(logits, dim=-

        g = -torch.log(-torch.log(torch.rand_like(z_indices_predict_prob)))
        temperature = self.choice_temperature * (1 - ratio)
        confidence = z_indices_predict_prob + temperature * g

        confidence = torch.where(
            mask_bc,
            confidence,
            torch.tensor(float('-inf'), device=confidence.device)
        )
        _, sorted_indices = torch.sort(confidence, descending=True)
        num_to_keep = int(mask_num * (1 - self.gamma(ratio)))

        new_z_indices_predict = z_indices_predict.clone()
        new_mask_bc = mask_bc.clone()

        unmask_count = 0
        for i in range(sorted_indices.size(1)):
            idx = sorted_indices[0, i].item()
            if 0 <= idx < self.num_image_tokens and mask_bc[0, idx]:
                new_z_indices_predict[0, idx] = z_indices_predict_candidate[0, idx]
                new_mask_bc[0, idx] = False
                unmask_count += 1
                if unmask_count >= num_to_keep:
                    break
        return new_z_indices_predict, new_mask_bc
```

The inference process for image inpainting tasks uses an iterative decoding strategy, gradually reducing the mask ratio according to the mask scheduling function until a complete image is generated.

- **Initial Mask Processing**:

  Converting the mask image to latent representation level mask Initializing token sequence, preserving original tokens in unmasked regions and setting masked regions to special mask token

- **Mask Scheduling Functions**:

  Implementing three mask scheduling strategies: cosine, linear, and square These functions control the rate of mask ratio decrease in each iteration

- **Iterative Decoding Process**:

  In each iteration, getting predictions for the current sequence through the Transformer Ranking predictions by confidence and keeping the highest confidence predictions Determining the number of predictions to keep based on the current iteration's mask ratio Updating the token sequence and recording history states

- **Image Reconstruction**:

  Handling any remaining mask tokens in the final step Converting the final token sequence back to an image Generating the final inpainted image through the VQGAN decoder

3 Discussion

Class-conditional Image Editing with Bounding Boxes

Beyond standard inpainting, one particularly promising extension of MaskGIT is class-conditional image editing using bounding boxes as mask inputs. This approach offers several advantages over traditional inpainting methods:

- **Semantic Control**: Rathe'r than simply filling empty regions, this method allows controlling the semantic category of the generated content within specified bounding boxes.
- **Structure Preservation**: The original pose, proportion, and position of objects can be maintained while only changing their class attributes.
- **Flexible Editing Capabilities**: Objects in an image can be "transformed" into other categories without complete redrawing.
- **Multi-object Editing**: Multiple bounding boxes can be marked simultaneously, enabling editing of multiple objects in a single operation.

This approach expands MaskGIT's applications to creative fields like film production, advertising customization, concept design, and virtual try-on systems, where specific objects need to be replaced while maintaining overall scene coherence.

4 Experiments and Results

4.1 Implementation Details

Model Training

For training the MaskGIT transformer model, I used the following hyperparameters:

- Epochs: 50
- Learning Rate: 1e-4
- Batch Size: 10
- Optimizer: AdamW with weight decay of 0.01

The training was performed on the dataset consisting of 12,000 training images with a resolution of 64×64. During training, the mask ratio was randomly sampled for each batch to ensure the model learns to predict tokens under various masking conditions.

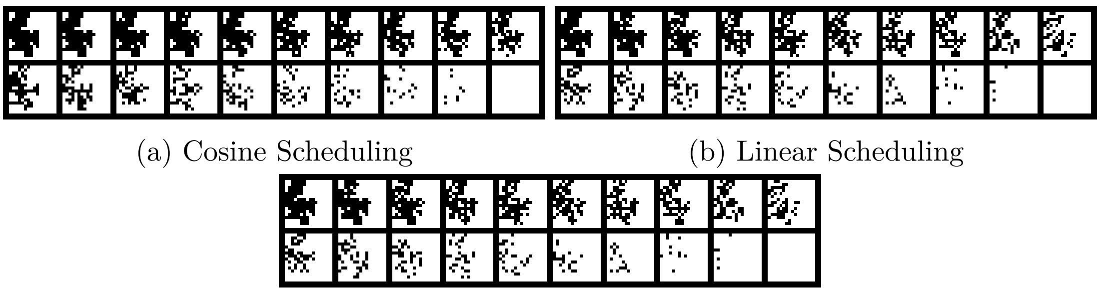
Inference Settings

For the image inpainting task, I used the following configuration:

- Batch Size: 1 (processing one image at a time)
- Total Iterations: 20 (for iterative decoding)
- Temperature: 4.5 (as specified in the MaskGIT configuration)

Three different mask scheduling functions (cosine, linear, and square) were implemented and compared for the inpainting process. For each function, the ratio parameter increased gradually from 0 to 1 across the 20 iterations.

4.2 Iterative Decoding Visualization



(a) Cosine Scheduling (b) Linear Scheduling

(c) Square Scheduling

Figure 1: Iterative Decoding Process

(a) Predicted image of cosine

(b) Predicted image of linear

(c) Predicted image of square

Figure 2: Iterative Decoding Process

- **Cosine Scheduling**: Shows a pattern where mask removal starts slowly, accelerates in the middle stages, and then slows down again at the end. This matches the expected behavior of a cosine function.
- **Linear Scheduling**: Displays a more uniform rate of mask removal throughout the process, with approximately the same number of tokens revealed in each step.
- **Square Scheduling**: Exhibits slower mask removal at the beginning but accelerates significantly toward the end, consistent with a squared function behavior.

4.3 Mask Scheduling Comparison

To evaluate the performance of different configurations, we calculated FID (Fréchet Inception Distance) scores, which measure the similarity between generated images and real images.

Scheduling Function	Average FID	Score
Cosine	38.56	20
Linear	37.61	20
Square	37.52	20

Table 1: FID Scores for Different Mask Scheduling Functions

All three methods received the same score of 20 in your scoring system, suggesting they all performed well for the inpainting task. These results align with the MaskGIT paper's findings that different mask scheduling strategies can affect the quality of generated images, with cosine scheduling often providing a good balance between quality and generation speed.