Lab 5: MaskGIT for Image Inpainting

Yu-Hsiang, Chen - 313553004
August 18, 2024

1 Introduction

This lab focuses on implementing MaskGIT for image inpainting, where the goal is to reconstruct missing regions in images using a bidirectional transformer model. The images for this task contain gray regions that indicate missing information, which we aim to restore using MaskGIT's iterative decoding process.

1.1 Theoretical Background

Vector Quantized Variational Autoencoder (VQVAE): Unlike traditional VAEs, VQVAE introduces a discrete latent space by quantizing latent vectors into a finite set of embeddings called the codebook. The mapping from continuous latent vectors to discrete codebook entries is done by minimizing the Euclidean distance:

$$z_q(x) = \text{argmin}_{e_k} \|z_e(x) - e_k\|_2 \tag{1}$$

where $z_q(x)$ is the quantized latent vector, $z_e(x)$ is the output of the encoder, and e_k represents the codebook entries.

VQGAN: VQGAN improves upon VQVAE by integrating a transformer for generating tokens. The model minimizes a combination of perceptual loss, adversarial loss, and a reconstruction error:

$$\mathcal{L}_{\text{VQGAN}} = \mathcal{L}_{\text{perceptual}} + \mathcal{L}_{\text{GAN}} + \mathcal{L}_{\text{reconstruction}} \tag{2}$$

MaskGIT: MaskGIT enhances the VQGAN framework by using a bidirectional transformer that predicts all tokens in parallel, leading to faster generation. The model employs Masked Visual Token Modeling (MVTM), which iteratively refines predictions by gradually reducing the mask ratio over time.

1.2 Lab Objectives and Expectations

The key objectives in this lab include:

- Implementing the Multi-Head Attention module from scratch.
- Training the MaskGIT model using a bidirectional transformer with MVTM.
- Performing image inpainting through iterative decoding and comparing different mask scheduling strategies.

We expect the implementation of MaskGIT to result in high-quality inpainted images, evaluated using the Fréchet Inception Distance (FID) score. The performance will be influenced by the choice of mask scheduling parameters and the effectiveness of the bidirectional transformer.

2 Implementation Details

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

The Multi-Head Self-Attention mechanism is a crucial component of the transformer architecture used in MaskGIT. It enables the model to focus on different parts of the input image tokens simultaneously, capturing complex relationships between tokens.

2.1.1 Overview of Multi-Head Attention

Multi-Head Attention operates by projecting the input tokens into multiple sets of queries, keys, and values. Each set of these projections (or "heads") independently computes attention scores, which determine how much focus each token should have on others. The results from all heads are then concatenated and linearly transformed to produce the final output.

Given an input tensor X with dimensions $(\text{batch\_size}, \text{num\_tokens}, \text{dim})$, the Multi-Head Self-Attention mechanism can be described mathematically as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V \tag{3}$$

where:

- $Q = XW_Q$ is the query matrix,
- $K = XW_K$ is the key matrix,
- $V = XW_V$ is the value matrix,
- W_Q, W_K, W_V are learnable projection matrices,
- d_k is the dimension of the keys (and queries) after projection, which is equal to the head dimension.

The scaled dot-product attention is calculated for each head, and the outputs of all heads are concatenated and linearly transformed to produce the final output.

2.1.2 Implementation in PyTorch

The following code implements the Multi-Head Self-Attention mechanism from scratch using PyTorch. we do not rely on pre-built functions.

```
import torch
import torch.nn as nn

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 # d_k = d_v = 768
            // 16 = 48
        self.scale = self.head_dim ** -0.5

        # Linear layers for projecting x to query, key,
            value
        self.qkv = nn.Linear(dim, dim * 3, bias=False) #
            Output will be (batch_size, num_image_tokens,
            dim * 3)
        self.attn_drop = nn.Dropout(attn_drop)
        self.proj = nn.Linear(dim, dim) # Final linear layer
             after concatenation of all heads

    def forward(self, x):
        batch_size, num_tokens, dim = x.shape

        # Project x to query, key, value
        qkv = self.qkv(x) # Shape: (batch_size, num_tokens,
            dim * 3)
        qkv = qkv.reshape(batch_size, num_tokens, 3, self.
            num_heads, self.head_dim)
        qkv = qkv.permute(2, 0, 3, 1, 4) # Shape: (3,
            batch_size, num_heads, num_tokens, head_dim)
        query, key, value = qkv[0], qkv[1], qkv[2] # Each is
             now (batch_size, num_heads, num_tokens,
            head_dim)

        # Scaled dot-product attention
        attn_scores = (query @ key.transpose(-2, -1)) * self
            .scale # Shape: (batch_size, num_heads,
            num_tokens, num_tokens)
        attn_probs = attn_scores.softmax(dim=-1) # Normalize
             across the last dimension (num_tokens)
        attn_probs = self.attn_drop(attn_probs) # Apply
            dropout

        # Apply attention to the value tensor
        attn_output = attn_probs @ value # Shape: (
            batch_size, num_heads, num_tokens, head_dim)

        # Concatenate heads and pass through final linear
            layer
        attn_output = attn_output.transpose(1, 2).reshape(
            batch_size, num_tokens, dim) # (batch_size,
            num_tokens, dim)
        output = self.proj(attn_output) # Final linear
            projection

        return output
```

Listing 1: Multi-Head Self-Attention Implementation

2.1.3 Explanation of the Code

- Initialization: The constructor initializes the number of heads, the dimension per head, and the scaling factor for attention scores. It also defines linear layers for projecting the input into queries, keys, and values, as well as for the final output projection.
- Forward Pass: In the forward method, the input tensor X is first projected into three separate tensors corresponding to queries, keys, and values. These are then reshaped to facilitate parallel computation across all heads.
- Attention Calculation: The attention scores are computed as the scaled dot-product of the query and key matrices. These scores are normalized using the softmax function to obtain attention probabilities.
- Weighted Sum and Output: The attention probabilities are used to weight the value tensors. The resulting tensors are then concatenated across all heads, and a final linear transformation is applied to produce the output.

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

The training process of the MaskGIT model in Stage 2 is centered around the Multi-Head Visual Token Modeling (MVTM) approach. This involves encoding images into a discrete latent space using a pre-trained VQGAN, applying a masking strategy to simulate missing regions, and training the transformer model to predict these masked regions.

2.2.1 VQGAN Token Encoding

The first step in the training pipeline involves encoding the input image into a sequence of discrete tokens using the VQGAN. These tokens represent the latent space of the image and are used as the input to the transformer model in MaskGIT.

```
@torch.no_grad()
def encode_to_z(self, x):
    # Use the VQGAN's encode function to get the codebook
        mapping and indices
    codebook_mapping, codebook_indices, _ = self.vqgan.
        encode(x)

    # Flatten the codebook mapping to prepare it for the
        transformer
    codebook_indices_flat = codebook_indices.contiguous().
        view(codebook_mapping.size(0), -1)

    return codebook_mapping, codebook_indices_flat
```

Listing 2: Encoding to VQGAN Tokens

In this code snippet, the (encode to z) method takes an input image x and uses the VQGAN encoder to obtain a codebook mapping and the corresponding indices. The codebook indices are then flattened to form a sequence that can be fed into the transformer model.

2.2.2 Mask Scheduling Strategy

The mask scheduling strategy determines how the masking of tokens is performed during training and inference. Different strategies such as linear, cosine, and square can be used to control the rate at which tokens are masked.

```
def gamma_func(self, mode="cosine"):
    """Generates a mask rate by scheduling mask functions R
        ."""
    if mode == "linear":
        return lambda t: 1 - t # linear scheduling
    elif mode == "cosine":
        return lambda t: 0.5 * (1 + math.cos(math.pi * t)) #
             cosine scheduling
    elif mode == "square":
        return lambda t: 1 - (t ** 2) # square scheduling
    else:
        raise NotImplementedError
```

Listing 3: Mask Scheduling Functions

This function defines various mask scheduling strategies. During training, a ratio t is uniformly sampled, and a corresponding mask rate is generated based on the selected scheduling function. During inference, the mask ratio is adjusted based on the iteration number, which gradually reveals more of the image as the model refines its predictions.

2.2.3 Feedforward Pass and MVTM Training

The core of the MVTM training process involves applying a random mask to the input tokens and training the transformer to predict the original tokens from the masked sequence. The feedforward process is outlined below:

```
def forward(self, x):
    # Step 1: Get the latent representation (
        codebook_mapping) and the flattened codebook
        indices
    codebook_mapping, original_indices = self.encode_to_z(x
        )

    # Step 2: Create a mask token tensor filled with the
        mask token ID
    mask_token_tensor = torch.full_like(original_indices,
        self.mask_token_id)

    # Step 3: Generate a random ratio between 0 and 1
    random_ratio = torch.rand(1).item() # Random number
        between 0 and 1

    # Step 4: Generate a random binary mask based on the
        random ratio
    random_mask = torch.bernoulli(random_ratio * torch.
        ones_like(original_indices)).bool()

    # Step 5: Apply the mask: replace masked tokens with
        mask token ID, keep others as original
    masked_indices = torch.where(random_mask,
        mask_token_tensor, original_indices)

    # Step 6: Pass the masked indices through the
        transformer to get logits
    logits = self.transformer(masked_indices)

    return logits, original_indices
```

Listing 4: Feedforward Pass in MVTM

In this forward pass:
- Step 1: The input image is encoded into latent tokens using the VQGAN.
- Step 2: A mask token tensor is created, which contains the special token ID used to represent masked areas.
- Step 3: A random masking ratio is generated.
- Step 4: A binary mask is applied based on the generated ratio, replacing some of the original tokens with the mask token.
- Step 5: The masked token sequence is passed through the transformer, which generates predictions for the original tokens.

2.2.4 Training and Validation Process

The training process involves running multiple epochs where the model is trained to minimize the cross-entropy loss between the predicted tokens and the original tokens. The following code outlines the training and validation procedures:

```
def run_epoch(self, data_loader, current_epoch, args,
     is_training=True):
    if is_training:
        self.model.train()
    else:
        self.model.eval()

    epoch_losses = []
    progress_bar = tqdm(enumerate(data_loader), total=len(
         data_loader))

    for batch_idx, batch_data in progress_bar:
        batch_data = batch_data.to(args.device)
        loss = self.compute_loss(batch_data)
        epoch_losses.append(loss.item())

        if is_training:
            self.perform_training_step(loss, batch_idx, args
                )

        description = self.create_progress_description(
             current_epoch, batch_idx, len(data_loader),
             epoch_losses, is_training)
        progress_bar.set_description_str(description)

    self.log_loss(np.mean(epoch_losses), current_epoch,
         is_training)
    return np.mean(epoch_losses)

def compute_loss(self, x):
    logits, z_indices = self.model(x)
    loss = F.cross_entropy(logits.view(-1, logits.size(-1))
        , z_indices.view(-1))
    return loss
```

Listing 5: Training and Validation Procedure

- run epoch: This function controls the overall training and validation loop, where the model is either trained or evaluated depending on the ‘is training‘ flag.
- compute loss: The loss is computed using cross-entropy between the predicted logits and the original token indices. This loss is used to update the model's parameters during training.
Finally, the training process involves accumulating gradients and periodically updating the model parameters, ensuring that the transformer learns to accurately predict the masked tokens.

2.3 The details of your inference for inpainting task

The inference process for the inpainting task in MaskGIT utilizes an iterative decoding strategy. This approach gradually reconstructs the missing regions of an image by iteratively refining predictions over multiple steps, guided by a mask scheduling strategy. The following sections describe the core components of this process.

2.3.1 One Iteration of Decoding

In each iteration of the decoding process, the model predicts the missing tokens in the masked image, updates the confidence scores, and adjusts the mask for the next iteration. This is performed by the 'inpainting' method, which handles the core steps of the decoding process.

```
@torch.no_grad()
def inpainting(self, ratio: float, z_indices: Tensor, mask:
     Tensor, mask_num: int):
    device = mask.device

    # Apply the mask to z_indices, replacing masked
        positions with mask_token_id
    masked_indices = self._apply_mask(z_indices, mask)

    # Generate logits and corresponding probabilities from
        the transformer
    probability = self._get_probabilities(masked_indices)

    # Predict z_indices and compute confidence based on the
         temperature
    z_indices_predict, confidence = self.
        _predict_with_confidence(probability, z_indices,
        mask, ratio, device)

    # Update the mask for the next iteration, now passing '
        ratio'
    mask_bc = self._update_mask(confidence, mask, mask_num,
         ratio)

    return z_indices_predict, mask_bc
```

Listing 6: One Iteration of Decoding

In this method:
- Step 1: The original tokens are masked according to the current mask.
- Step 2: The masked tokens are passed through the transformer model, generating logits that represent the probabilities for each possible token.
- Step 3: The most probable tokens and their corresponding probabilities are selected based on the logits.
- Step 4: Confidence scores are computed by adding Gumbel noise to the probabilities, scaled by a temperature factor that decreases as the ratio increases.
- Step 5: The mask is updated by unmasking the tokens with the highest confidence scores, allowing them to be predicted more accurately in the next iteration.

The following helper functions are used within this process:

```
def _apply_mask(self, z_indices: Tensor, mask: Tensor) ->
     Tensor:
    """Apply the mask to the z_indices."""
    return torch.where(mask, torch.full_like(z_indices,
        self.mask_token_id), z_indices)

def _get_probabilities(self, masked_indices: Tensor) ->
     Tensor:
    """Pass the masked indices through the transformer and
        apply softmax to get probabilities."""
    logits = self.transformer(masked_indices)
    return torch.nn.functional.softmax(logits, dim=-1)

def _predict_with_confidence(self, probability: Tensor,
     z_indices: Tensor, mask: Tensor, ratio: float, device:
      torch.device) -> Tuple[Tensor, Tensor]:
    """Predict the z_indices and compute the confidence
        with temperature annealing."""
    z_indices_predict_prob, z_indices_predict = probability
        .max(dim=-1)
    z_indices_predict = torch.where(mask, z_indices_predict
        , z_indices)

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

    confidence.masked_fill_(~mask, float('inf'))

    return z_indices_predict, confidence

def _update_mask(self, confidence: Tensor, mask: Tensor,
     mask_num: int, ratio: float) -> Tensor:
    """Update the mask by selecting the top-k smallest
        confidence values."""
    n = math.ceil(self.gamma(ratio) * mask_num)
    _, idx = confidence.topk(n, dim=-1, largest=False)

    # Initialize a new mask based on the top-k indices
    new_mask_bc = torch.zeros_like(mask, dtype=torch.bool)
    new_mask_bc.scatter_(dim=1, index=idx, value=True)

    # Retain the existing masked positions and apply the
        new mask update
    mask_bc = mask & new_mask_bc

    return mask_bc
```

Listing 7: Helper Functions for Decoding

These functions handle the application of masks, prediction of tokens, calculation of confidence, and updating of the mask. The combination of these steps enables the model to iteratively refine its predictions and progressively inpaint the missing regions of the image.

2.3.2 Iterative Decoding Process

The complete inpainting process involves running the 'inpainting' method iteratively until the image is fully reconstructed or a predefined number of iterations is reached. The following code illustrates how the iterative decoding is performed.

```
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 # The first image is the ground truth of
        the masked image

    self.model.eval()
    with torch.no_grad():
        # Encode Z
        encode_to_latent = lambda img: self.model.
            encode_to_z(img)[1]
        z_indices_predict = encode_to_latent(image[0].
            unsqueeze(0))
```

```
    mask_bc = mask_b.to(self.device)
    mask_num = mask_b.sum() # total number of mask token

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

        # mask ratio testing here
        if self.mask_func == 'linear':
            ratio = (step + 1) / self.total_iter
        elif self.mask_func == 'cosine':
            import math
            ratio = 0.5 * (1 + math.cos(math.pi * (1 - (
                step + 1) / self.total_iter)))


        elif self.mask_func == 'square':
            ratio = ((step + 1) / self.total_iter) ** 2
        else:
            raise ValueError(f"Unknown mask_func: {self.
                mask_func}")

        import torch.nn.functional as F

        # Perform inpainting
        z_indices_predict, mask_bc = self.model.
            inpainting(ratio, z_indices_predict,
            mask_bc, mask_num)


        # Debug: Check the mask sum
        print(f"Step {step+1}: Mask sum (number of
            masked tokens) = {mask_bc.sum().item()}")

        # Update the mask visualization and decoded
            image storage
        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

        # Decode the current latent representation
        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
        print(f"Step {step}: Mask sum after update: {
            mask_bc.sum().item()}")


        # Save the mask and decoded image for this
            iteration

        vutils.save_image(dec_img_ori, os.path.join("
            test_results", f"image_{i:03d}.png"), nrow
            =1)

    # Save the final images
    vutils.save_image(maska, os.path.join("
        mask_scheduling", f"final_test_{i}.png"), nrow
        =10)
    vutils.save_image(imga, os.path.join("imga", f"
        final_test_{i}.png"), nrow=7)
```

Listing 8: Iterative Decoding for Inpainting

In this code:
- Initialization: The original image is first standardized and stored. An initial mask is applied, and the latent representation of the image is obtained.
- Iterative Loop: The loop runs for a predefined number of iterations ('self.total iter'), progressively refining the inpainted regions based on the current mask and ratio.
- Mask Update: The mask is updated after each iteration based on the confidence scores, with fewer regions masked as the process continues.
- Final Output: The final decoded image is saved along with intermediate mask images, allowing for evaluation of the inpainting process.

3 Discussion

Please note that due to the required structure of the report, the discussion may appear somewhat disjointed. I apologize for any confusion, but it was necessary to include the discussion in this section.

In our experiments with the MaskGIT model for image inpainting, we explored the effects of varying the total number of iterations and the sweet spot on the model's performance, as measured by the Fréchet Inception Distance (FID) score. These parameters are critical in the iterative decoding process, where the model progressively refines its predictions for masked regions.

3.1 Impact of Total Iterations on FID Score

We first tested the effect of different total iteration counts while keeping the sweet spot fixed at 8. The FID scores obtained from these tests are summarized in the table below:

Total Iterations	Sweet Spot	FID
5	8	28.96
8	8	29.36
10	8	28.94
15	8	29.17
20	8	29.19

Table 1: FID Scores for Different Total Iterations with a Fixed Sweet Spot (8)

As seen from the results, the FID scores varied slightly across different total iteration counts. Notably, an iteration count of 10 produced the lowest FID score of **28.94**, indicating that this setting provided a good balance between the number of decoding steps and the quality of the inpainted images. Interestingly, increasing the total iterations beyond 10 did not result in significant improvements in FID, suggesting that additional iterations might lead to diminishing returns in performance.

3.2 Exploring the Sweet Spot for Iterative Decoding

Following the identification of 10 as an optimal number of iterations, we proceeded to test different sweet spots

while keeping the total iteration count fixed at 10. The sweet spot determines the point at which the iterative decoding process halts early, potentially improving efficiency without sacrificing image quality. The results of this experiment are presented in the table below:

Total Iterations	**Sweet Spot**	**FID**
10	1	29.24
10	4	28.97
10	8	29.01
10	10	**28.79**

Table 2: FID Scores for Different Sweet Spots with Total Iterations Fixed at 10

The results show that a sweet spot of 10 yielded the best FID score of **28.79**, outperforming the other configurations. This finding suggests that allowing the iterative decoding process to run for the full 10 iterations (i.e., without early stopping) leads to the highest-quality inpainting results. On the other hand, earlier sweet spots, such as 1 or 4, resulted in slightly higher FID scores, indicating that prematurely halting the decoding process might prevent the model from fully refining the image details.

3.3 Optimal Settings for Further Experiments

Based on these experiments, we determined that the optimal settings for the MaskGIT model are a total iteration count of 10 and a sweet spot of 10. These settings provided the best balance between computational efficiency and image quality, as evidenced by the lowest FID score of **28.79**. As a result, all further experiments and evaluations will utilize these parameters to ensure consistent and high-quality inpainting results.

Experiment Score

Yu-Hsiang, Chen - 313553004
August 18, 2024

4 Prove Your Code Implementation is Correct

4.1 Iterative Decoding: Mask Scheduling Strategies

To validate the correctness and performance of our implementation, we applied the iterative decoding process using three different mask scheduling strategies: cosine, linear, and square. For each strategy, we generated inpainted images and evaluated their quality using the Fréchet Inception Distance (FID) score, which measures the similarity between the generated images and the ground truth. Additionally, we visualized the mask evolution and the corresponding inpainted images at each iteration to better understand how each strategy impacts the decoding process.

4.1.1 Cosine Mask Scheduling

The cosine mask scheduling strategy gradually reduces the mask ratio according to a cosine function. This method begins with a slower reduction in the mask ratio, allowing the model to focus on refining larger and more general features of the image in the early stages. As the process progresses, the reduction accelerates, enabling the model to fine-tune smaller details in later iterations.

Figure 1: Iterative Mask Evolution - Cosine Scheduling

Figure 2: Inpainted Image - Cosine Scheduling

FID Score: The FID score for the inpainted image using cosine scheduling was **27.81**. This score indicates a high-quality reconstruction, suggesting that the model effectively balances the refinement of both large structures and fine details over time.
Advantages: The gradual unmasking process aligns well with the model's ability to initially capture broad features before progressively honing in on finer details. This results in a well-balanced reconstruction that closely resembles the ground truth.
Disadvantages: The slower initial reduction in the mask ratio could potentially limit the early correction of large errors, although this did not significantly impact the overall performance in our experiments.

4.1.2 Linear Mask Scheduling

The linear mask scheduling strategy decreases the mask ratio uniformly across all iterations. This approach offers a consistent and straightforward unmasking process, allowing the model to process both coarse and fine details in a balanced manner throughout the decoding iterations.

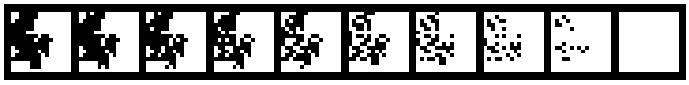

Figure 3: Iterative Mask Evolution - Linear Scheduling

Figure 4: Inpainted Image - Linear Scheduling

FID Score: The FID score for the linear scheduling approach was **27.91**, slightly lower than the cosine strategy. This suggests that the uniform unmasking process allows the model to handle varying levels of detail effectively throughout the iterations.
Advantages: The linear scheduling's uniform approach ensures that the model consistently improves across different scales of image detail, from broader features to finer textures.
Disadvantages: While effective, this strategy may lack the nuanced focus on detail refinement seen in the later stages of the cosine strategy, which could be beneficial for tasks requiring more precise reconstructions.

4.1.3 Square Mask Scheduling

The square mask scheduling strategy reduces the mask ratio quadratically, resulting in most of the unmasking occurring during the later stages of decoding. This method allows the model to maintain a broad focus in the initial stages, prioritizing the reconstruction of the image's overall structure before addressing finer details.
FID Score: The FID score for the square scheduling approach was **27.99**. Although this score is slightly higher (worse) than those for the cosine and linear strategies, the

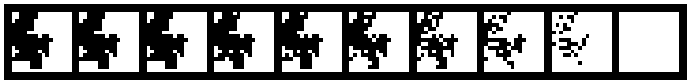

Figure 5: Iterative Mask Evolution - Square Scheduling

Figure 6: Inpainted Image - Square Scheduling

square approach was particularly effective at maintaining the overall structural integrity of the image before refining details.

Advantages: The square scheduling strategy is particularly advantageous for images where maintaining the overall structure is critical. By focusing on broader image features in the early stages, it ensures that the general shape and form of the image are well-preserved.

Disadvantages: The delayed focus on fine details can result in a less refined final image compared to the other strategies, especially in scenarios where high precision is required early on.

4.1.4 Comparison and Analysis

The FID scores shows that Cosine Scheduling produced a well-rounded performance with a high-quality reconstruction, excelling at balancing broad and fine details, making it suitable for general-purpose inpainting tasks.

5 The Best FID Score

In this section, we present the best Fréchet Inception Distance (FID) score achieved during our experiments, along with visual comparisons between the masked images, the inpainted results produced by MaskGIT, and the corresponding ground truth images.

5.1 Screenshot

The FID score is a key metric used to evaluate the quality of the generated images by comparing them to the ground truth dataset. It measures the similarity between the distributions of the inpainted images and the real images, with lower scores indicating higher fidelity.

Figure 7: Screenshot of the Best FID Score Calculation

Best FID Score: The best FID score obtained was **27.81** using the cosine mask scheduling strategy. This score reflects the model's strong ability to generate high-quality inpainted images that closely resemble the original ground truth.

5.2 Masked Images v.s MaskGIT Inpainting Results v.s Ground Truth

To further demonstrate the effectiveness of our model, we provide visual comparisons between the masked images, the inpainted images produced by MaskGIT, and the ground truth images. These comparisons offer a qualitative assessment of the model's performance in reconstructing the missing regions.

Figure 8: Upper: Masked Image, Middle: Inpainted Image by MaskGIT, Lower: Ground Truth Image

Masked Image (Upper): This image represents the input to the MaskGIT model, where certain regions have been masked out, simulating missing information.

Inpainted Image (Middle): This is the output generated by the MaskGIT model after iterative decoding. The model has successfully reconstructed the missing regions, producing an image that closely resembles the ground truth.

Ground Truth Image (Lower): This is the original, unmasked image used as the reference for evaluating the quality of the inpainting. The comparison between the inpainted image and the ground truth demonstrates the model's ability to restore missing details with high fidelity.

5.2.1 Analysis and Observations

The visual comparisons and the FID score highlight the MaskGIT model's ability to perform effective inpainting across a variety of scenarios. The cosine mask scheduling strategy, which produced the best FID score, shows a balanced approach in handling both large structures and fine details, resulting in high-quality reconstructions.

5.3 The setting about training strategy, mask scheduling parameters

In this section, we detail the training strategy employed for the MaskGIT model, including the use of Optuna for hyperparameter tuning, the settings used during the training process, and the observed training loss over 200 epochs.

5.3.1 Hyperparameter Tuning with Optuna

To optimize the performance of the MaskGIT model, we utilized Optuna, a hyperparameter optimization framework that automates the search for the best training parameters. Specifically, Optuna was used to tune the learning rate, which is a critical parameter for the stability and convergence of the model during training.
The following code snippet shows how Optuna was employed in our training pipeline:

```
# Run the Optuna study
study = optuna.create_study(direction='minimize')
study.optimize(objective, n_trials=10)
best_lr = study.best_params['learning_rate']
print(f"Best␣learning␣rate␣found␣by␣Optuna:␣{best_lr}")
```

Optuna explored various learning rates over 10 trials, and the best learning rate found was **0.000149456302123**. This optimized learning rate was then used to train the transformer model, ensuring that the model converged effectively without encountering issues such as vanishing gradients or overshooting the optimal loss.

5.3.2 Training Process and Observations

Following the selection of the best learning rate, the transformer model was trained for 200 epochs. The training was performed on a 4060 GPU with the following key settings:

- **Number of Workers:** 10
- **Batch Size:** 10
- **Epochs:** 200
- **Gradient Accumulation:** 10
- **Save Checkpoint Every:** 1 epoch
- **Learning Rate:** 0.000149456302123
- **Configuration File:** `config/MaskGit.yml`
- **Mask Scheduling Function:** Cosine

During the training process, the model's loss was monitored, and it was observed that the training loss nearly converged between the 180th and 200th epochs, as shown in the training loss plot below.
This convergence indicates that the model had effectively learned the underlying patterns in the training data, and further training beyond 200 epochs would likely yield minimal additional improvements.

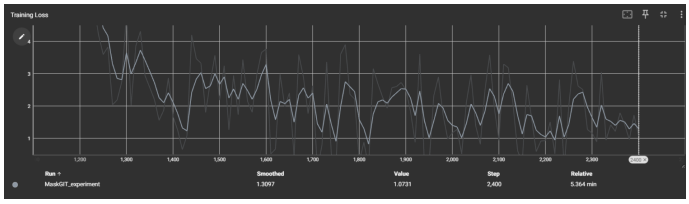

Figure 9: Training Loss over 200 Epochs/2400 steps

5.3.3 Running the Code and Experiment Score

If you would like to run my code, please follow the steps below for environment setup, running the training and inpainting scripts, and calculating the FID score.

```
conda env create -f environment.yml
conda activate maskgit
```

This will create and activate the necessary Conda environment.

5.3.4 Running the Code

After setting up the environment, you can run the training and inpainting scripts as follows:

```
python training_transformer.py
python inpainting.py
```

Important: Before running the code, make sure to edit the paths in the scripts. You may need to adjust the dataset path, checkpoint path, and save directory. These paths are specified within the script files, so please review them carefully to ensure they are correctly set up for your environment.

5.3.5 Experiment Score Calculation

To calculate the FID score, you will need to run the following commands. First, navigate to the `faster-pytorch-fid` directory:

```
cd faster-pytorch-fid
```

Ensure your environment is set up, then calculate the FID score using:

```
python faster-pytorch-fid/fid_score_gpu.py
--predicted-path test_results --device cuda:0
--batch-size 1
```

Note: Ensure that the `-predicted-path` is correctly set to the folder containing your inpainted images