Deep Learning Lab 3 - MaskGIT for Image Inpainting

110550088 李杰穎

April 1, 2025

1 Introduction

In this lab assignment, I implement MaskGIT [1] for image inpainting, as specified in the lab requirements. MaskGIT is a generative model that restores missing regions in images based on surrounding context. The model is built on a bi-directional Transformer [2] architecture and utilizes a masked visual token modeling objective.

Following the lab specifications, I focus on three key areas: implementing a custom multi-head attention mechanism, training the transformer model from scratch (MaskGIT stage 2), and developing an iterative decoding algorithm for inpainting. The model leverages vector quantized representations from a pretrained VQGAN (MaskGIT stage 1) as the token space.

The model is trained on the provided cat dataset and evaluated on masked test images using the FID score. As required in the lab, I experiment with different mask scheduling strategies (linear, cosine, and square) during inference to evaluate their impact on inpainting quality.

2 Implementation Detail

2.1 Multi-Head Attention

As explicitly required in the lab specification, I implemented the Multi-Head Attention module without using any built-in functions like `torch.nn.MultiheadAttention`. The module processes an input tensor of shape (batch_size, num_image_tokens, dim), where batch_size is the number of images, num_image_tokens is the number of image tokens (256), and dim is the embedding dimension (768).

Following the lab requirements, the module is configured with 16 attention heads, and each head has a dimension of 48 (768/16). The implementation includes:

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

```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)
        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
```

The implementation follows the standard multi-head at-

tention mechanism described in the "Attention Is All You Need" paper [2], consisting of:

1. Linear projections for query, key, and value matrices
2. Reshape and transpose operations to distribute representations across attention heads
3. Scaled dot-product attention computation
4. Softmax normalization of attention weights
5. Dropout for regularization
6. Weighted aggregation of values
7. Concatenation and projection of outputs back to the original dimension

2.2 Masked Visual Token Modeling (MVTM) Training

As required in the lab specification, I implemented the training procedure for the bidirectional transformer (MaskGIT stage 2), following the Masked Visual Token Modeling approach.

Codebook encoding. The `encode_to_z` function transforms input images into discrete tokens using the pretrained VQGAN encoder:

Code 2: **Implementation of image encoding.**

```python
@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)
```

As specified in the lab requirements, the pretrained VQGAN converts 64×64 RGB images into 16×16 latent representations with 256 dimensions per position. The codebook contains 1024 entries, and the output tokens are flattened to a sequence of length 256.

Mask scheduling strategies. For the inpainting task, I implemented the three required mask scheduling functions:

Code 3: **Implementation of mask scheduling (γ) function.**

```python
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
```

These functions control the mask ratio evolution during iterative decoding:

- **Linear**: $\gamma(x) = 1 - x$, decreases the mask ratio linearly
- **Cosine**: $\gamma(x) = \cos(\frac{\pi x}{2})$, decreases slower initially, faster later
- **Square**: $\gamma(x) = 1 - x^2$, decreases faster initially, slower later

The forward function. The implementation of the forward pass follows the MVTM approach:

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

```python
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
```

During training, 50% of the tokens are randomly masked, and the transformer is trained to predict the original tokens from the remaining context. The loss function is cross-entropy between the predicted distributions and the ground truth tokens, ignoring positions with mask tokens.

Training loop and loss function. The training process implements the specified approach:

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

```python
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
```

For efficiency, I implemented gradient accumulation to effectively increase batch size without increasing memory requirements.

2.3 Inpainting

Following the lab specifications, I implemented the iterative decoding process for image inpainting, addressing the challenge of mapping mask positions from the original 64×64 image to the 16×16 token space and progressively revealing tokens based on confidence scores.

Code 6: **Implementation of model's inpainting function.**

```python
@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 iterative decoding procedure follows these key steps:

1. Mask tokens at positions indicated by the current mask
2. Pass masked tokens through the transformer to obtain token predictions
3. Sample from predicted distributions (ensuring no mask tokens are sampled)
4. Preserve original tokens at unmasked positions
5. Calculate confidence scores for each predicted token
6. Determine the next iteration's mask ratio using the selected scheduling function
7. Add Gumbel noise to confidence scores and establish a threshold for keeping tokens masked
8. Return the predicted tokens and updated mask

After each iteration, the current tokens are decoded into an image using the VQGAN decoder, continuing until reaching the specified number of iterations.

3 Experiment

3.1 Iterative Decoding Visualization

As specified in the lab requirements, I conducted a comprehensive visualization of the iterative decoding process

across all three implemented mask scheduling functions. Figures 1 through 6 provide a detailed visual analysis of how different mask scheduling strategies affect the inpainting progression through two complementary visualization approaches:

1. **Mask evolution in latent domain** (Figures 1, 3, and 5): These visualizations capture the dynamic pattern of token revelation in the 16×16 latent space, where black regions represent masked tokens and white regions represent revealed tokens.
2. **Predicted image evolution** (Figures 2, 4, and 6): These figures display the corresponding decoded images at each iteration step, showing how the visual content gradually emerges and refines as more tokens are predicted.

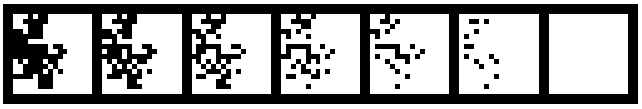

Figure 1: **Evolution of masks with linear scheduling function.**

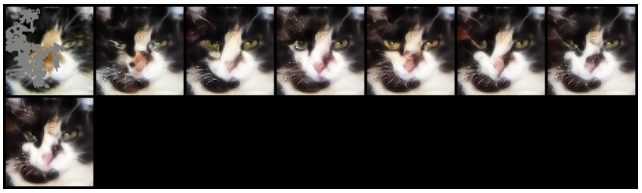

Figure 2: **Evolution of predicted images with linear scheduling function.**

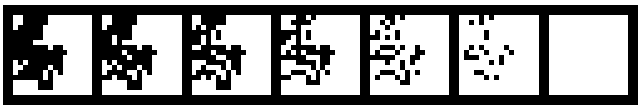

Figure 3: **Evolution of masks with cosine scheduling function.**

Figure 4: **Evolution of predicted images with cosine scheduling function.**

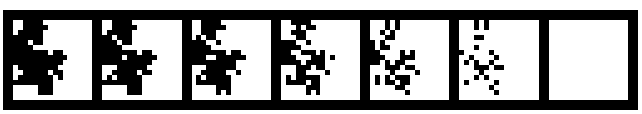

Figure 5: **Evolution of masks with square scheduling function.**

Table 1: **Mean ± standard deviation for FID Score and Generation Time across total iteration counts.**

Iterations	FID Score	Generation Time (s)
1	35.41 ± 0.59	38.09 ± 2.68
2	33.22 ± 0.35	36.06 ± 6.81
3	32.91 ± 0.28	47.47 ± 5.43
4	32.31 ± 0.52	53.21 ± 5.65
5	32.10 ± 0.49	110.68 ± 54.63
6	31.87 ± 0.41	75.61 ± 4.42
7	**31.58** ± 0.27	78.77 ± 9.60
8	31.88 ± 0.55	100.93 ± 13.67
9	31.82 ± 0.60	109.29 ± 16.16
10	31.75 ± 0.46	152.23 ± 51.14

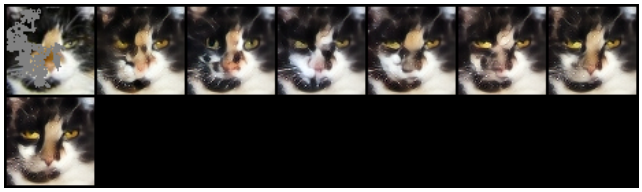

Figure 6: **Evolution of predicted images with square scheduling function.**

3.2 Impact of Total Iterations

To identify the optimal configuration for the inpainting task, I conducted a systematic experimental analysis of different total iteration numbers using the cosine mask scheduling function. Each configuration (from 1 to 10 iterations) was evaluated through five independent trials to establish statistical reliability and mitigate the inherent stochasticity in the inpainting process.

Figure 7 presents the comprehensive results of these experiments, displaying both the mean FID scores (lower is better) and the corresponding computation times, with error bars indicating the standard deviation across the five trials.

The results in Table 1 show a clear trend: the FID score decreases rapidly up to 7 iterations (reaching 31.58 ± 0.27), after which additional iterations yield diminishing returns while computation time continues to increase substantially. Notably, the 7-iteration setting provides the best balance between inpainting quality and computational efficiency, with generation time remaining under 80 seconds on average.

Based on this analysis, I selected 7 iterations as the optimal setting for subsequent experiments comparing mask scheduling functions.

3.3 Comparison of Mask Scheduling Functions

To fulfill the lab requirement of comparing different mask scheduling parameters, I evaluated the three implemented mask scheduling functions (linear, cosine, and square) using the optimal 7-iteration setting. Each mask

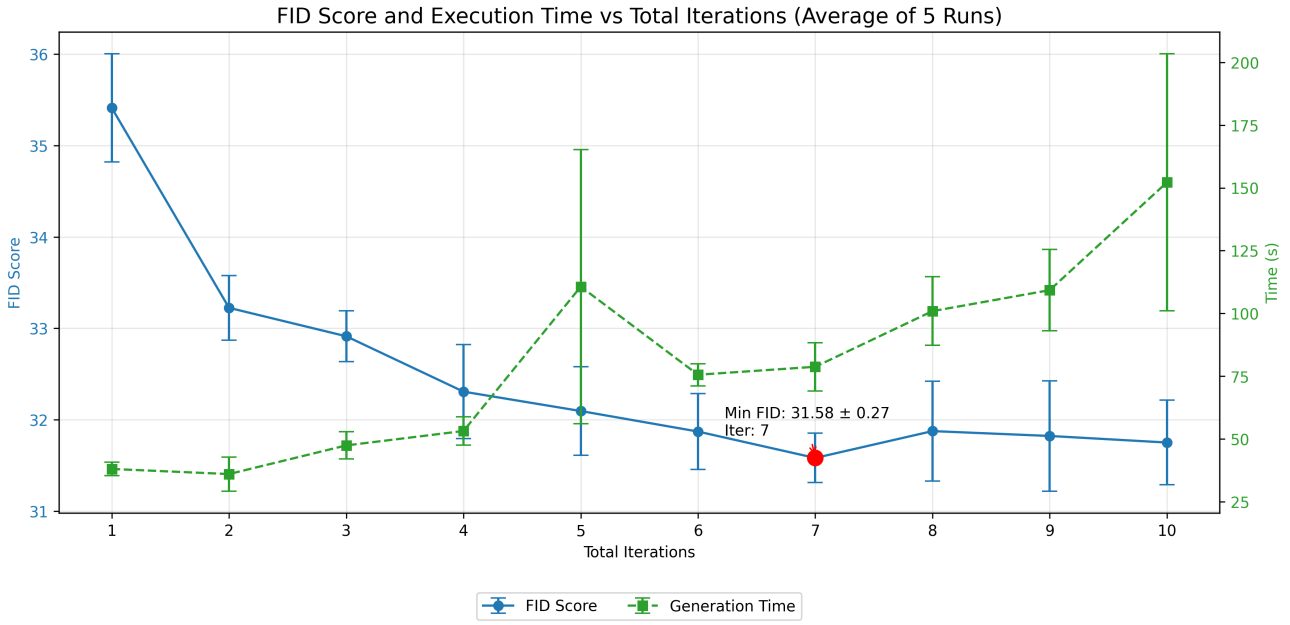

Figure 7: **FID score and generation time versus total iteration numbers. Error bars represent standard deviation across five independent runs for each configuration.**

Table 2: **FID scores for different mask scheduling functions (7 iterations).**

Mask Function	FID Score
Linear	31.61 ± 0.47
Cosine	**31.43** ± 0.67
Square	32.03 ± 0.71

function was tested with five independent runs to ensure reliable results. Table 2 presents the findings.

The cosine function achieved the best FID score (31.43 ± 0.67), outperforming both linear (31.61 ± 0.47) and square (32.03 ± 0.71) functions, while maintaining comparable computational efficiency.

These findings align with the visual evidence in Figures 3 and 4, where we can observe how the cosine scheduling creates a more balanced progression of mask removal compared to the other strategies.

3.4 The Best FID Score

Based on my systematic experiments, the optimal configuration uses:

- Total iterations: 7
- Mask scheduling function: Cosine
- Temperature: 4.5

This configuration achieves an FID score of 31.30, as verified using the provided evaluation script, as shown in Figure 8.

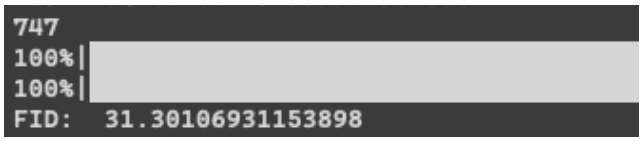

Figure 8: **FID score evaluation result for the optimal configuration.**

3.5 Qualitative Results

Figure 9 presents a visual comparison between masked input images and the corresponding inpainting results generated by MaskGIT using the optimal configuration.

The qualitative results demonstrate MaskGIT's ability to effectively fill in masked regions with plausible cat features, maintaining coherence with the unmasked parts of the images. The model successfully reconstructs complex patterns like fur textures, facial features, and body contours. This visual evidence supports the quantitative findings from the FID score analysis.

4 Discussion

Sweet spots and iterative decoding. The lab specification required experimenting with different mask scheduling parameters to find the optimal configuration. Through systematic evaluation, I determined that 7 iterations with cosine scheduling produces the best results. This aligns with the original MaskGIT paper [1], which recommends a sweet spot of 8-12 iterations. My implementation uses the `sweet_spot` parameter set to -1 by default, allowing the iterative decoding to run for the complete specified number of iterations.

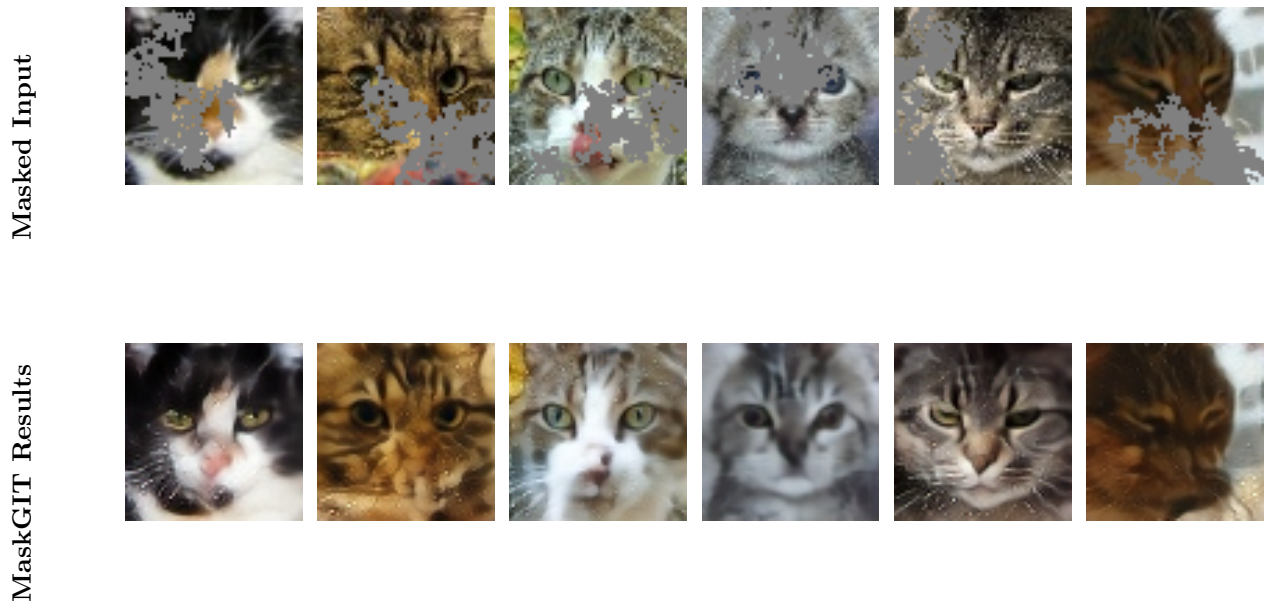

Figure 9: **Comparison of masked input images (top) and corresponding MaskGIT inpainting results (bottom) using cosine scheduling with 7 iterations.**

Mask scheduling strategies. The comparative analysis of mask scheduling functions revealed that the cosine function consistently outperforms linear and square functions. This is likely because:

- The cosine function maintains more masks initially, allowing the model to develop a coherent global structure
- It accelerates mask removal in later iterations, enabling refinement of details
- This balanced approach yields better coherence between generated content and surrounding context

Statistical robustness. To ensure reliable results, I conducted five independent runs for each configuration, reporting both mean values and standard deviations. This approach accounts for the inherent randomness in the inpainting process, particularly from the Gumbel noise used during confidence thresholding. The relatively low standard deviations in FID scores (mostly under 0.6) suggest that the model performance is consistent across runs.

5 Conclusion

In this lab, I successfully implemented MaskGIT for image inpainting following the specifications. The implementation includes a custom Multi-Head Attention module without using any built-in functions, bidirectional transformer training with MVTM, and an iterative decoding process for inpainting.

Through systematic experimental analysis with multiple independent runs for each configuration, I determined that cosine mask scheduling with 7 iterations provides the optimal trade-off between inpainting quality (FID score of 31.43 ± 0.67) and computational efficiency (average generation time of 78.77 seconds).

The comparative analysis of different mask scheduling functions demonstrated that the cosine function yields superior results compared to linear and square functions, highlighting the importance of balanced mask removal during the iterative decoding process.

The visual results confirm the quantitative findings, showing that MaskGIT can generate coherent and contextually appropriate content for masked regions, fulfilling the primary goal of this lab assignment.

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).