Deep Learning Homework 3 Report

Cheng-Liang Chi

March 31, 2025

Contents

1 Introduction

In this lab, we explore the use of **MaskGIT** [1], a masked generative image transformer, for the **image inpainting** task. Image inpainting aims to restore missing or corrupted regions in an image in a visually plausible manner. To achieve this, we implement the second stage of the MaskGIT pipeline, which involves constructing a **Multi-Head Self-Attention** module [2], training a **Bidirectional Transformer** using **Masked Visual Token Modeling (MVTM)**, and performing **iterative decoding** to complete masked images.

The core idea behind MaskGIT is to leverage a bidirectional Transformer architecture to address the inefficiencies of traditional autoregressive models, enhancing both generation quality and computational efficiency. Initially, images are encoded using a pretrained VQGAN encoder [3] to generate discrete latent tokens. During training, random subsets of these tokens are masked, and the model learns to predict these masked tokens. During inference, iterative decoding is used, employing various mask scheduling strategies to progressively refine the reconstruction based on confidence scores.

This report details our implementation steps, key design choices, analyses of mask scheduling strategies, and evaluation of inpainting quality using the **FID score** [4]. Our experiments aim to illustrate MaskGIT's effectiveness in generating semantically consistent and visually high-quality image reconstructions.

2 Implementation Details

This section provides a detailed explanation of our implementation, which is structured into three parts: Multi-Head Self-Attention, Masked Visual Token Modeling (MVTM), and Iterative Decoding for Inpainting.

2.1 Multi-Head Self-Attention

In the Multi-Head Self-Attention implementation, the input tensor is first linearly projected into query (Q), key (K), and value (V) vectors. Each of these vectors is reshaped into multiple heads to allow parallel attention computation. We employ scaled dot-product attention independently on each head, which calculates attention scores by the dot product of Q and K, scaled by the square root of the dimension of the keys. After computing the softmax scores, they are applied to the values (V). The outputs from different heads are concatenated and passed through a final linear layer to produce the attention output.

Specific implementation details and dimension calculations can be referred to in the provided source code.

```
class MultiHeadAttention(nn.Module):
    def __init__(self, dim=768, num_heads=16, attn_drop=0.1):
        super(MultiHeadAttention, self).__init__()
        self.head_num = num_heads
        self.head_dim = dim // num_heads
        self.scale = self.head_dim**-0.5

        self.to_qkv = nn.Linear(dim, 3 * self.head_num * self.head_dim,
        ↪  bias=False)

```

```
        self.head = nn.Sequential(
            nn.Linear(dim, dim),
            nn.Dropout(attn_drop),
        )

    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.
        """
        qkv = self.to_qkv(x)
        q, k, v = qkv.reshape(
            3,
            x.shape[0],
            self.head_num,
            x.shape[1],
            self.head_dim,
        )

        attn = (q @ k.transpose(-2, -1)) * self.scale
```

2.2 Masked Visual Token Modeling (MVTM)

The MVTM module begins by encoding input images into latent tokens using a pretrained VQGAN encoder. We randomly mask a subset of these tokens based on a Bernoulli distribution, replacing masked tokens with a dedicated mask token identifier. The masked sequence is then passed through a bidirectional Transformer, which predicts the original tokens at masked positions. The training objective is to minimize the cross-entropy loss between the predicted tokens and the ground-truth latent tokens.

Specific details regarding mask token selection, masking ratios, and Transformer configurations are described explicitly in the source code implementation.

```
    @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)

```

```
    def forward(self, x):
        _, z_indices = self.encode_to_z(x)
        mask = torch.bernoulli(
            0.5 * torch.ones(z_indices.shape, device=z_indices.device)
        ).bool()
        mask_token = (
            self.mask_token_id
            * torch.ones(z_indices.shape, device=z_indices.device).long()
        )
        masked_tokens = mask * mask_token + (~mask) * z_indices
        logits = self.transformer(masked_tokens)
        return logits, z_indices
```

2.3 Iterative Decoding for Inpainting

During inference, iterative decoding reconstructs masked regions of latent tokens through successive refinement. Initially, all masked positions are filled with the mask token. At each decoding iteration, the Transformer predicts token probabilities, and tokens with the highest confidence scores are fixed for subsequent iterations, while lower confidence tokens are remasked. Various mask scheduling strategies, including cosine, linear, and square, determine how the masking ratio decreases throughout the iterations. This iterative decoding continues until no masked tokens remain.

Implementation details such as temperature annealing, Gumbel noise application for sampling tokens, and specific mask scheduling functions are available within the provided implementation.

```
    # mask_b: iteration decoding initial mask, where mask_b is true means
    ↪ mask
    def inpainting(self, image, mask_b, i):  # MakGIT inference
        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)
            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
            dec_img_ori = None
            # iterative decoding for loop design
            # Hint: it's better to save original mask and the updated mask by scheduling separately
            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,
                    mask_num,
                    ratio,
                )

                # static method yon can modify or not, make sure your visualization results are correct
                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)
                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  # get decoded image

            assert dec_img_ori is not None
            ##decoded image of the sweet spot only, the test_results folder path will be the --predicted-path for fid score calculation
            vutils.save_image(
                dec_img_ori, os.path.join("test_results", f"image_{i:03d}.png"), nrow=1
            )

            # demo score
            vutils.save_image(
                maska,
                os.path.join("mask_scheduling", f"test_{i}.png"),
```

```
        return logits, z_indices

    @torch.no_grad()
    def inpainting(self, z_indices, mask, mask_num, step):
        """predict"""
        masked_z_indices = mask * self.mask_token_id + (~mask) * z_indices
        logits = self.transformer(masked_z_indices)

        # Apply softmax to convert logits into a probability distribution
        ↪  across the last dimension.
        prob = torch.softmax(logits, dim=-1)
        # Find max probability for each token value & max prob index
        z_indices_predict_prob, z_indices_predict = prob.max(dim=-1)

        """ mask schedule """
        ratio = self.gamma(step)
        mask_len = torch.ceil(mask_num * ratio).long()

        """ sample """
        # predicted probabilities add temperature annealing gumbel noise as
        ↪  confidence
        g = torch.distributions.gumbel.Gumbel(0, 1).sample(
            z_indices_predict_prob.shape
        )  # gumbel noise
        assert g is not None
        g = g.to(z_indices_predict_prob.device)

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

        """ mask """
        # hint: If mask is False, the probability should be set to
        ↪  infinity, so that the tokens are not affected by the
        ↪  transformer's prediction
        confidence[~mask] = torch.inf
        # sort the confidence for the rank
        _, idx = confidence.topk(mask_len, dim=-1, largest=False)
        # define how much the iteration remain predicted tokens by mask
        ↪  scheduling
        mask_bc = torch.zeros(
            z_indices.shape, dtype=torch.bool,
            ↪  device=z_indices_predict_prob.device
        )
        # At the end of the decoding process, add back the
        ↪  original(non-masked) token values

```

3 Discussion

In this section, we discuss key insights gained from our experiments, potential limitations of our approach, and directions for future improvement.

My experiments showed the choice of mask scheduling function significantly influences the quality of inpainting results. The cosine scheduling consistently delivered the best performance, highlighting its effectiveness in gradually and smoothly reducing the mask ratio. Conversely, linear and square schedules showed slightly lower performance, possibly due to abrupt or uneven changes in masking ratios.

Another area worth exploring is the impact of different iterative decoding parameters, such as the number of iterations and the sweet spot. While my current settings provided good results, further experiments could optimize these parameters to balance computational efficiency and inpainting quality.

Lastly, future work might include applying our MaskGIT implementation to higher-resolution datasets or exploring other Transformer architectures to potentially improve inpainting quality and diversity further.

4 Experiment Score

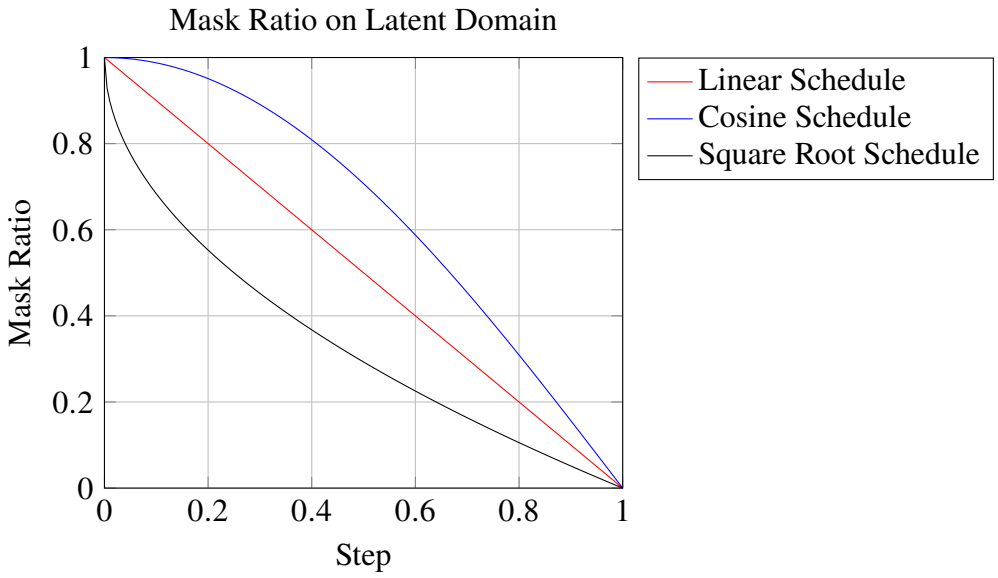
This section presents the experimental results validating our implementation, structured into two main parts: demonstration of iterative decoding and reporting of the best FID scores obtained.

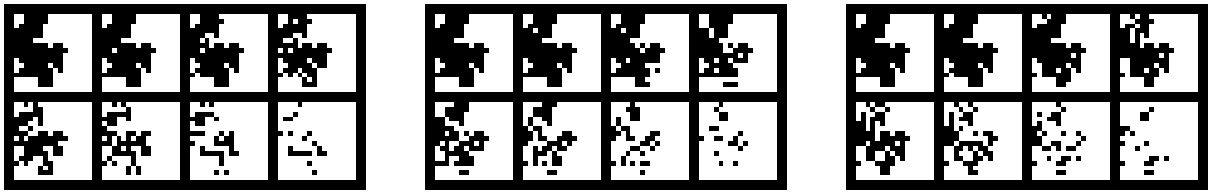
4.1 Iterative Decoding

In this part, we demonstrate the iterative decoding process for different mask scheduling strategies.

4.1.1 Mask in Latent Domain

Present visualizations of the mask distribution over iterations using different scheduling strategies (cosine, linear, square).





(a) Cosine Schedule (b) Linear Schedule (c) Square Root Schedule

Figure 1: Mask in Latent Domain with Different Schedules. Three of this schedule are generated with the same image.

4.1.2 Predicted Images

The intermediate predicted images at different decoding iterations are shown in Figures 2a, 2b, and 2c. The images illustrate the model's ability to progressively fill in the masked regions, demonstrating the effectiveness of the iterative decoding process.

(a) Cosine Schedule (b) Linear Schedule (c) Square Root Schedule

Figure 2: Predicted Images at Different Iterations. The images show the model's iterative refinement process.

The predicted images for the linear schedule are shown in Figure 2b. The images demonstrate the iterative refinement process, where the model progressively fills in the masked regions.

4.2 Best FID Score

4.2.1 Results

The FID scores obtained from the experiments are summarized in Table 1. The scores indicate the quality of the generated images, with lower scores indicating better performance.

Table 1: FID Scores for Different Mask Scheduling Strategies

Mask Scheduling Strategy	FID Score
Cosine Schedule	38.02
Linear Schedule	39.12
Square Root Schedule	39.30

4.2.2 Experimental Settings

The experiments were conducted with the following settings:

- **Learning Rate:** 1×10^{-4}
- **Batch Size:** 32
- **Epochs:** 200
- **Sweet Spot:** 8
- **Total Iterations:** 8

And the results shown in Table 1 are using the lowest loss on the validation set.

References

[1] H. Chang, H. Zhang, L. Jiang, C. Liu, and W. T. Freeman, "Maskgit: Masked generative image transformer," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022, pp. 11 315–11 325. [Online]. Available: https://arxiv.org/abs/2202.04200.

[2] A. Vaswani, N. Shazeer, N. Parmar, *et al.*, "Attention is all you need," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017, pp. 5998–6008. [Online]. Available: https://arxiv.org/abs/1706.03762.

[3] P. Esser, R. Rombach, and B. Ommer, "Taming transformers for high-resolution image synthesis," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2021, pp. 12 873–12 883. [Online]. Available: https://arxiv.org/abs/2012.09841.

[4] M. Heusel, H. Ramsauer, T. Unterthiner, B. Nessler, and S. Hochreiter, "Gans trained by a two time-scale update rule converge to a local nash equilibrium," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017, pp. 6626–6637. [Online]. Available: https://arxiv.org/abs/1706.08500.

[5] Hank891008, *Deep-learning*, 2023. [Online]. Available: https://github.com/hank891008/Deep-Learning.