Lab 03: MaskGIT for Image Inpainting

112550127 Pin-Kuan Chiang

July 29, 2025

1 Introduction

In this project, we aim to perform image inpainting using a combination of VQGAN and a Transformer. We first use pretrained VQGAN to encode images into discrete latent tokens. Then, we train a Transformer model to predict masked tokens within this latent space. During inference, we apply iterative decoding with different mask scheduling strategies to progressively recover the masked image content. Our goal is to explore and compare scheduling methods and hyperparameter configurations to identify the optimal setup that achieves the best FID score.

2 Implementation Details

2.1 Multi-Head Self-Attention

The Multi-Head Attention mechanism enables the model to capture contextual dependencies by attending to multiple representation subspaces in parallel. Our implementation follows the standard Transformer design with separate projections for queries, keys, and values, followed by head-wise attention computation and concatenation.

Initialization. The class defines the overall dimensionality and number of heads. It enforces that the embedding size is divisible by the number of heads, and calculates the per-head dimension and attention scaling factor.

```
class MultiHeadAttention(nn.Module):
    def __init__(self, dim=768, num_heads=16, attn_drop=0.1):
        super(MultiHeadAttention, self).__init__()
        assert dim % num_heads == 0
        self.head_dim = dim // num_heads
        self.scale = 1.0 / math.sqrt(self.head_dim)
```

Linear Projections. Three linear layers are initialized to transform the input into query (Q), key (K), and value (V) representations. A final output projection and attention dropout layer are also defined.

```
        self.q_proj = nn.Linear(dim, dim)
        self.k_proj = nn.Linear(dim, dim)
        self.v_proj = nn.Linear(dim, dim)
        self.attn_drop = nn.Dropout(p=attn_drop)
        self.out_proj = nn.Linear(dim, dim)
```

Input Projections. During the forward pass, the input tensor $x \in \mathbb{R}^{B \times N \times D}$ is projected into Q, K, and V using the corresponding linear layers.

```
    def forward(self, x):
        B, N, D = x.shape
        q = self.q_proj(x)
        k = self.k_proj(x)
        v = self.v_proj(x)
```

Head Separation. Each projected tensor is reshaped to separate the attention heads. The tensors are rearranged to the shape (B, H, N, d_h), where H is the number of heads and d_h is the dimension per head.

```
        q = q.view(B, N, self.num_heads, self.head_dim).permute(0, 2, 1, 3)
        k = k.view(B, N, self.num_heads, self.head_dim).permute(0, 2, 1, 3)
        v = v.view(B, N, self.num_heads, self.head_dim).permute(0, 2, 1, 3)
```

Attention Computation. Scaled dot-product attention is computed for each head independently. The resulting attention weights are normalized via softmax and regularized with dropout.

```
        attn_scores = torch.matmul(q, k.transpose(-2, -1)) * self.scale
        attn = torch.softmax(attn_scores, dim=-1)
        attn = self.attn_drop(attn)
```

Context Aggregation and Output Projection. The attention weights are used to compute a weighted sum over the values. The results from all heads are concatenated and projected back to the original embedding space.

```
        ctx = torch.matmul(attn, v)
        ctx = ctx.permute(0, 2, 1, 3).contiguous().view(B, N, D)
        out = self.out_proj(ctx)
        return out
```

This implementation enables the model to jointly attend to information from different positions and subspaces, enhancing its ability to learn spatially and semantically rich representations.

2.2 Masked Visual Token Modeling

In the second stage, we train a bidirectional transformer to predict the masked discrete visual tokens produced by the pretrained VQGAN. The model learns to reconstruct missing tokens based on the visible context, following the principles of Masked Language Modeling applied to image tokens.

Model Initialization. The model consists of a frozen VQGAN encoder and a learnable bidirectional transformer. The VQGAN is used to discretize images into latent tokens.

```
class MaskGit(nn.Module):
    def __init__(self, configs):
        super().__init__()

        self.vqgan = self.load_vqgan(configs['VQ_Configs'])
        for p in self.vqgan.parameters():
            p.requires_grad = False
        self.vqgan.eval()
```

The VQGAN module is loaded from pretrained weights and set to evaluation mode. Its gradients are disabled to ensure that only the transformer is updated during MVTM training.

Token Configuration. We define parameters such as token vocabulary size and mask token ID. A gamma function is used to dynamically control the masking ratio during training.

```
        self.num_image_tokens     = configs['num_image_tokens']
        self.num_codebook_vectors = configs['num_codebook_vectors']
        self.mask_token_id        = self.num_codebook_vectors
        self.choice_temperature   = configs.get('choice_temperature', 4.5)
        self.gamma                = self.gamma_func(configs.get('gamma_type', 'cosine'))
```

Each image is represented by a sequence of tokens of fixed length (e.g., $16 \times 16 = 256$). The [MASK] token is assigned a new ID beyond the vocabulary size (e.g., 1024).

Transformer Backbone. The transformer is responsible for predicting the masked tokens based on bidirectional context.

```
        self.transformer = BidirectionalTransformer(configs['Transformer_param'])

        self._last_mask  = None
        self._last_ratio = None
```

VQGAN Encoding. This helper function encodes an image into a flattened token sequence.

```
@torch.no_grad()
def encode_to_z(self, x):
    _, codebook_indices, _ = self.vqgan.encode(x)
    z_indices = codebook_indices.view(-1, self.num_image_tokens).long()
    return z_indices
```

Gamma Function. We define a masking schedule function $\gamma(r)$ to control the percentage of masked tokens, which is used in inpainting section.

```python
def gamma_func(self, mode="cosine"):
    if mode == "linear":
        return lambda r: 1.0 - r
    elif mode == "cosine":
        return lambda r: np.cos(np.pi * r / 2.0)
    elif mode == "square":
        return lambda r: 1.0 - (r ** 2)
    else:
        raise NotImplementedError
```

Forward Pass: MVTM Training. During training, we randomly mask tokens in the latent sequence and ask the transformer to recover them.

```python
def forward(self, x):
    device = x.device
    z_indices = self.encode_to_z(x)  # (B, 256)
    B, N = z_indices.shape

    r = torch.rand(B, device=device)
    mask_ratio = torch.from_numpy(self.gamma(r.cpu().numpy())).to(device)
    n_mask = torch.clamp((mask_ratio * N).ceil().long(), min=1, max=N)
```

A random ratio $r \sim \mathcal{U}(0, 1)$ is sampled per image to determine how many tokens should be masked. The total number of masked tokens $n = \lceil \gamma(r) \cdot N \rceil$ is bounded between 1 and N.

```python
    mask = torch.zeros(B, N, dtype=torch.bool, device=device)
    for b in range(B):
        perm = torch.randperm(N, device=device)[: n_mask[b]]
        mask[b, perm] = True
```

Tokens to be masked are randomly selected via permutation. A boolean mask tensor marks the selected positions.

```python
    tokens_in = z_indices.clone()
    tokens_in[mask] = self.mask_token_id
```

The masked tokens are replaced with a special token ID. The unmasked tokens remain unchanged.

```python
    logits = self.transformer(tokens_in)  # (B, N, vocab_size)

    self._last_mask = mask
    self._last_ratio = mask_ratio
    return logits, z_indices, mask
```

The model is trained to predict the correct token for each masked position. Only the masked positions are included in the loss computation. The true target and mask tensor are returned along with the transformer output logits.

2.3 Inference

During inference, we iteratively decode masked image tokens using confidence-based scheduling. At each step, a portion of tokens are predicted and progressively unmasked based on their predicted confidence scores.

Stepwise Inpainting (Single Iteration).

```python
def inpainting(self, z_indices, mask, ratio, orig_mask_count):
    tokens_in = z_indices.clone()
    tokens_in[mask] = self.mask_token_id
    logits = self.transformer(tokens_in)
    logits[..., self.mask_token_id] = -1e9
```

The model receives the current token sequence with masked positions replaced by a special `[MASK]` token. To prevent the model from predicting a mask token, its logit is suppressed.

```python
    probs = torch.softmax(logits, dim=-1)
    z_prob, z_pred = probs.max(dim=-1)
```

Predicted token probabilities are extracted via softmax, and the most likely token is chosen for each position.

```
    tau = self.choice_temperature * (1.0 - ratio).clamp(min=0.0)
    g = self._gumbel(z_prob.shape, device=device)
    confidence = torch.log(z_prob + 1e-9) + tau * g
    confidence = confidence.masked_fill(~mask, -float('inf'))
```

We apply Gumbel sampling to model stochastic confidence scores. Only currently masked positions are considered, and others are excluded with $-\infty$.

```
    remain_mask = int(math.floor(self.gamma(ratio.item()) * orig_mask_count))
    current_mask_count = int(mask.sum().item())
    n_unmask = max(0, current_mask_count - remain_mask)
```

The masking schedule $\gamma(t/T)$ determines how many tokens should remain masked. The rest will be selected to be unmasked.

```
    keep_mask = torch.zeros_like(mask)
    for b in range(B):
        if n_unmask <= 0:
            continue
        topk_idx = torch.topk(confidence[b], k=n_unmask, largest=True).indices
        keep_mask[b, topk_idx] = True
```

Based on confidence scores, the top-k tokens are selected to be unmasked in this iteration.

```
    z_next = z_indices.clone()
    z_next[mask] = z_pred[mask]
    return z_next, keep_mask
```

The predicted tokens are adopted into the sequence for the next step. The updated sequence and the mask of newly revealed tokens are returned.

Full Iterative Decoding Loop.

```
def inpainting(self, image, mask_b, i):
    maska = torch.zeros(self.total_iter, 3, 16, 16)
    imga = torch.zeros(self.total_iter+1, 3, 64, 64)
    mean = ...
    ori = (image[0]*std)+mean
    imga[0] = ori
```

We initialize image and mask storage. The first image is the input masked image and stored as the initial state.

```
    z_indices = self.model.encode_to_z(image)
    z_indices_predict = z_indices.clone()
    mask_bc = mask_b.to(device=self.device)
    orig_mask = mask_bc.clone()
    orig_mask_count = int(orig_mask.sum().item())
```

The image is encoded into a sequence of discrete tokens. The original mask is saved to determine how many tokens to reveal at each step.

```
    for step in range(self.total_iter):
        ratio = float(step+1) / float(self.total_iter)
        _mask_bc = mask_bc.clone()
        z_indices_predict, keep_mask = self.model.inpainting(
            z_indices_predict, _mask_bc, ratio, orig_mask_count
        )
        mask_bc = _mask_bc & (~keep_mask)
```

For each decoding step, we update the ratio t/T and compute the new prediction. Tokens marked by `keep_mask` are considered unmasked.

```
        mask_i = mask_bc.view(1, 16, 16)
        mask_image = torch.ones(3, 16, 16)
        mask_coords = torch.nonzero(mask_i[0])
        if mask_coords.numel() > 0:
            mask_image[:, mask_coords[:, 0], mask_coords[:, 1]] = 0
        maska[step] = mask_image
```

The binary mask is visualized for analysis. Black squares represent masked latent tokens.

```
        z_q = self.model.vqgan.codebook.embedding(z_indices_predict).view(1, 16, 16, 256)
        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 token sequence is converted back into an image using the VQGAN decoder. The resulting image is stored per step.

```
        if step == self.sweet_spot:
            break
```

To accelerate evaluation, we optionally terminate decoding at a predefined sweet spot iteration.

3 Discussion

During the training of the Transformer model, we observed several interesting behaviors and challenges.

In the early stages of training, the FID score was surprisingly low. However, this appeared to be misleading. At that point, the model had not yet learned to properly distinguish between foreground objects (e.g., animals) and background regions. As a result, it often blended object contours into the background, especially when background colors were dominant or smooth. This led to visually plausible but semantically incorrect reconstructions.

Around epoch 50, the model began to improve significantly. It started to better capture object structure and boundaries, which led to more meaningful inpainting results. However, after epoch 150, we noticed a growing gap between the training and validation loss curves, suggesting that the model was beginning to overfit.

To address this, we applied early stopping and selected the model checkpoint at epoch 150, where the validation performance was still optimal without being overfitted.

We also experimented with different batch sizes: 5, 10, and 20. Among these, batch size 10 consistently produced the best results. A batch size of 20 appeared to make optimization harder and convergence slower, possibly due to reduced gradient noise and poorer generalization. Conversely, batch size 5 led to overly volatile training, causing instability in loss curves.

These observations highlight the importance of both monitoring training dynamics and tuning hyperparameters carefully, especially when working with Transformer-based inpainting models.

4 Experiment Score

4.1 Correctness of Implementation

To verify the correctness of our iterative decoding implementation, we provide visualizations of the masked latent tokens and their corresponding predictions after 20 decoding steps using different masking strategies. Each scheduler (linear, cosine, square) is shown with its mask and predicted image.

(a) Linear Mask Scheduler

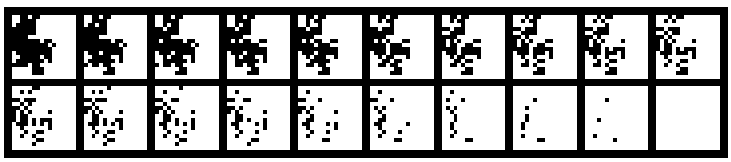

Figure 1: Masked latent tokens with linear mask scheduler

Figure 2: Predicted image after 20 iterations (linear mask scheduler)

(b) Cosine Mask Scheduler

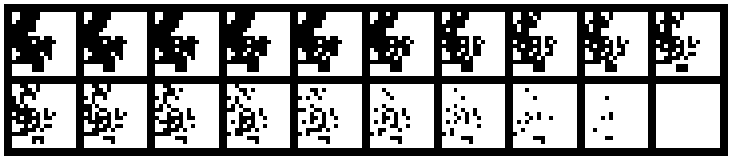

Figure 3: Masked latent tokens with cosine mask scheduler

Figure 4: Predicted image after 20 iterations (cosine mask scheduler)

(c) Square Mask Scheduler

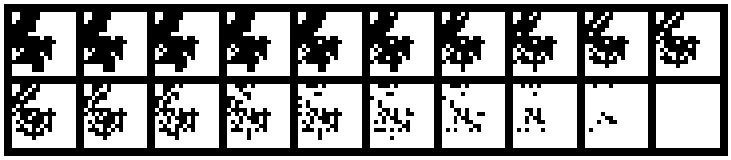

Figure 5: Masked latent tokens with square mask scheduler

Figure 6: Predicted image after 20 iterations (square mask scheduler)

4.2 Best FID Score

We evaluated different combinations of mask functions and sweet spot parameters. The best performance was achieved using **linear masking** with **sweet_spot = 3**, resulting in the lowest FID score of **29.68**.

Figure 7: Screenshot

FID Results (Total Iterations = 7):

Sweet Spot	Linear	Cosine	Square
3	**29.68**	30.42	30.21
5	29.85	30.71	30.15
7	30.24	29.89	30.10

Table 1: FID scores for different sweet_spot values and mask functions (lower is better)

Visual Results: Below is a row-wise comparison of ground truth images (top) and corresponding inpainting predictions (bottom) under the best setting.

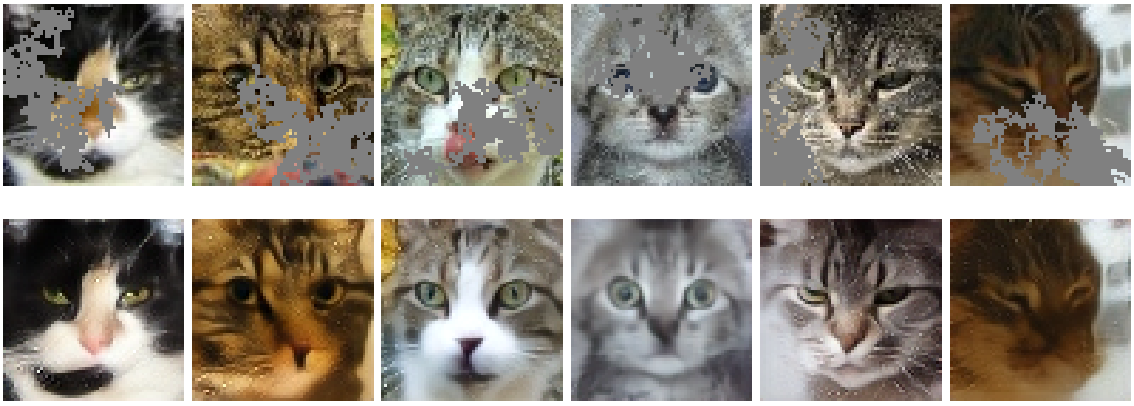

Figure 8: Best FID Result

Below are my hypermeter setting of transformer. **Training Strategy:**

- Epochs: 150 (200 epochs caused overfitting)
- Learning Rate: 1×10^{-4}
- Batch Size: 10 (batch size 20 yielded worse results)

- Optimizer: Adam
- Scheduler: Cosine decay
- Total Iterations: 7

In summary, the linear mask function combined with a sweet spot value of 3 produced the best inpainting quality, both visually and in terms of FID score.