

Introduction

In this lab, we implement a Masked Visual Token Modeling (MVTM) approach using the MaskGIT model, focusing on Multi-Head Self-Attention and iterative decoding for image inpainting. Key processes such as token masking, transformer prediction, and loss calculation are explored. Experiment results compare the impact of different mask scheduling functions on the quality of image inpainting.

Implementation Details

Multi-Head Self-Attention

The input x is first passed through the linear layers to obtain the Q (query), K (key), and V (value) matrices.

After the linear layers, Q , K , and V are reshaped into multiple heads and then transpose them to prepare for parallel processing in the attention mechanism.

$scores$ is calculated by taking the dot product of Q and K , and scaling by the square root of d_k for numerical stability. Then, a softmax function is applied to normalize the scores into attention weights.

The attention weights are multiplied with the value matrix V to compute the weighted sum, which represents the attention output.

The output is finally reshaped back into its original dimensions (combining the heads), and a final linear transformation is applied to produce the output.

```
class MultiHeadAttention(nn.Module):
    def __init__(self, dim=768, num_heads=16):
        super(MultiHeadAttention, self).__init__()

        self.num_heads = num_heads
        self.dim = dim
        self.d_q = dim // num_heads
        self.d_k = dim // num_heads
        self.d_v = dim // num_heads

        self.q_linear = nn.Linear(dim, dim)
        self.k_linear = nn.Linear(dim, dim)
        self.v_linear = nn.Linear(dim, dim)

        self.out_linear = nn.Linear(dim, dim)

    def forward(self, x):
        batch_size = x.size(0)

        Q = self.q_linear(x)
        K = self.k_linear(x)
        V = self.v_linear(x)

        Q = Q.view(batch_size, -1, self.num_heads, self.d_q).transpose(1, 2)
        K = K.view(batch_size, -1, self.num_heads, self.d_k).transpose(1, 2)
        V = V.view(batch_size, -1, self.num_heads, self.d_v).transpose(1, 2)

        scores = torch.matmul(Q, K.transpose(-2, -1)) / math.sqrt(self.d_k)
        attn_weights = torch.softmax(scores, dim=-1)

        out = torch.matmul(attn_weights, V)
        out = out.transpose(1, 2).contiguous().view(batch_size, -1, self.dim)
        out = self.out_linear(out)

        return out
```

Training Transformer

Masked Visual Token Modeling (MVTM)

The helper function `mask_latent()` selects a subset of tokens in each `z_indices` to be replaced with a specified ID `mask_token_id`, based on a given rate `mask_rate`.

The calculation of mask rates happens in the `forward()` function in the `MaskGIT` class, which will be discussed in the next part.

```
def mask_latent(z_indices, mask_token_id, mask_rate):
    masked_z_indices = z_indices.clone()

    num_tokens = z_indices.shape[1]
    num_masked = int(mask_rate * num_tokens)

    batch_size = z_indices.shape[0]
    for i in range(batch_size):
        masked_indices = torch.randperm(num_tokens)[:num_masked]
        masked_z_indices[i, masked_indices] = mask_token_id

    return masked_z_indices
```

Forward Function (in `MaskGIT`)

For each call to the `forward()`, first, a mask ratio is sampled, and a mask rate is calculated according to the selected mask scheduling function (including cosine, linear and square functions) and the mask ratio.

After calculating the mask rate, a masked latent `masked_z_indices` is generated with the helper function `mask_latent()` mentioned above.

Finally, the transformer predicts the probability distribution for each token, regardless if a mask is applied on the token or not.

```
def forward(self, x):
    _, z_indices = self.encode_to_z(x)

    mask_ratio = np.random.uniform(0, 1)
    mask_rate = self.gamma(mask_ratio)
    masked_z_indices = mask_latent(z_indices, self.mask_token_id, mask_rate)

    logits = self.transformer(masked_z_indices)

    return logits, z_indices
```

Loss Calculation

The cross entropy loss is calculated between the predicted logits and the true token indices, which measures how well the model's predicted probability distribution aligns with the actual tokens. The logits and `z_indices` are reshaped to match the batched input format for the loss function.

```
for data in tqdm(train_loader, desc=f"Training Epoch {epoch}"):
    inputs = data.to(self.args.device)

    logits, z_indices = self.model(inputs)
    loss = F.cross_entropy(logits.view(-1, logits.size(-1)), z_indices.view(-1))

    self.optim.zero_grad()
    loss.backward()
    self.optim.step()
```

Iterative Decoding

The inpainting process involves iteratively predicting and filling in masked tokens in the latent space of the image.

At each step, the model uses a transformer to predict the probability distribution of the tokens, applies temperature-annealed Gumbel noise to adjust the confidence, and then updates the masked tokens based on the highest confidence, gradually refining the image until the target step is reached.

Specifically, the mask is updated by first calculating the confidence scores for all tokens, then unmasking the tokens with the highest confidence scores based on a threshold determined by the current ratio and current mask count. The mask is updated so that tokens with the highest confidence are retained, while others may continue to be masked in the next iteration.

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

    z_indices_predict, mask_b = self.model.inpainting(
        image, mask_b, ratio, mask_num
    )
```

```
@torch.no_grad()
def inpainting(self, image, mask_b, ratio, mask_num):
    _, z_indices = self.encode_to_z(image)
    logits = self.transformer(z_indices)
    z_indices_predict_prob = torch.nn.functional.softmax(logits, dim=-1)

    # Add temperature annealing gumbel noise to the confidence
    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

    # Predict the tokens and replace the masked tokens
    z_indices_predict = torch.argmax(confidence, dim=-1)
    z_indices_predict[~mask_b] = z_indices[~mask_b]


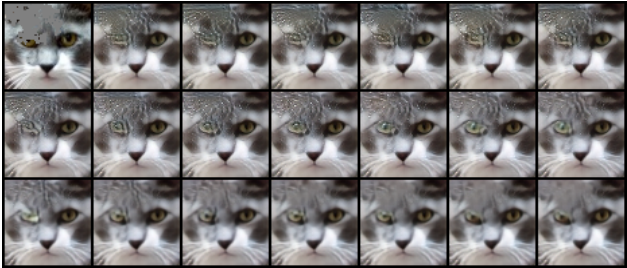
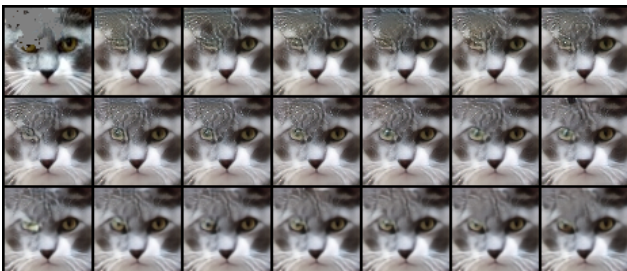
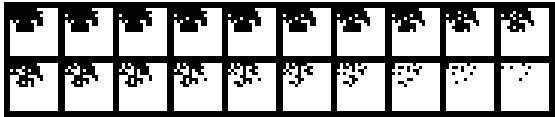
    # If the prediction is special token, replace it with 0 and set confidence to -inf
    z_indices_predict[z_indices_predict == self.mask_token_id] = 0

    # Get next mask
    confidence, _ = torch.max(confidence, dim=-1)
    # Replace the mask token with -inf if the prediction is mask token
    confidence[z_indices_predict == self.mask_token_id] = -float("inf")
    confidence[~mask_b] = -float("inf")
    tokens_to_unmask = mask_b.sum() - math.floor(self.gamma(ratio) * mask_num)
    _, indices = torch.topk(confidence, tokens_to_unmask, dim=-1)
    if indices.nelement() > 0:
        mask_b[0, indices[0]] = False

    return z_indices_predict, mask_b
```

Experiment Results

Iterative Decoding with different Mask Scheduling Functions

Cosine	 FID: 49.43946927327548	
Linear	 FID: 48.82109676227037	
Square	 FID: 49.467322378922916	

The Best FID Score

FID Score

FID: 48.82109676227037

```
~/dlp/lab5/faster-pytorch-fid [ ] main *5
> python3 fid_score_gpu.py --predicted-path ../permanent/cosine/test_results --device cuda:0
747
100%|
100%|
FID: 49.43946927327548
~/dlp/lab5/faster-pytorch-fid [ ] main *5
> python3 fid_score_gpu.py --predicted-path ../permanent/linear/test_results --device cuda:0
747
100%|
100%|
FID: 48.82109676227037
~/dlp/lab5/faster-pytorch-fid [ ] main *5
> python3 fid_score_gpu.py --predicted-path ../permanent/square/test_results --device cuda:0
747
100%|
100%|
FID: 49.467322378922916
```

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

Masked Images						
MaskGIT Inpainting Results						
Ground Truth						

Hyperparameters

Inpainting Parameters:

- **sweet_spot**: 20
- **total_iter**: 20
- **choice_temperature**: 4.5
- **mask_func**: cosine
- **random_seed**: 910615

Transformer Training Parameters:

- **batch_size**: 24
- **epochs**: 15
- **learning_rate**: 0.0001
- **mask_func**: cosine
- **random_seed**: 910615