DLP
Lab3: MaskGIT for Image Inpainting
Report

林睿騰(Rui-Teng Lin)
313551105

March 31, 2025

Contents

1 Introduction

In this lab, we will implement the MaskGIT for image inpainting. There are three stages in this lab: Train a VQVAE to encode the image into a discrete latent space, train a MaskGIT to generate the image from masked latent codes, and use the MaskGIT to inpaint the image.

1.1 VQVAE

The VQVAE is a type of variational autoencoder that uses a discrete latent space. The encoder maps the input image to a latent code, and the decoder maps the latent code to the reconstructed image. The latent code is a discrete codebook, which is a set of discrete latent codes. The decoder maps the latent code to the reconstructed image.

1.2 MaskGIT

MaskGIT is a type of generative adversarial network that uses a masked latent code to generate the image. In training stage, we use the VQVAE to encode the image into a latent code, and then use the MaskGIT to generate the image from the masked latent code.

1.3 Image Inpainting

In the image inpainting stage, we use the MaskGIT to inpaint some masked images. For each iteration, we paint all the masked pixels with the generated image from the MaskGIT. Then, we choose specified percentage of the regions where the agent has less confidence to paint, and mask them again. The percentage of each iteration is configured by the mask scheduling function.

2 Implementation Details

2.1 Multi-head Attention

Implementation of the multi-head attention mechanism is straightforward. We just need to prepare the query, key, and value matrices, and then perform the attention mechanism.

```python
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.
    """

    # prepare q, k, v
    b, n, _ = x.shape
    q = self.W_Q(x)
    k = self.W_K(x)
    v = self.W_V(x)

    # split q, k, v into num_heads
    q = q.view(b, n, self.num_heads, self.dim_head).transpose(1, 2)
    k = k.view(b, n, self.num_heads, self.dim_head).transpose(1, 2)
    v = v.view(b, n, self.num_heads, self.dim_head).transpose(1, 2)

    # calculate attn scores
    scores = torch.matmul(q, k.transpose(-2, -1)) /
    math.sqrt(self.dim_head)
    scores = torch.softmax(scores, dim=-1)
    scores = self.attn_drop(scores)

    # calculate output
    out = torch.matmul(scores, v)
    out = out.transpose(1, 2).contiguous().view(b, n, -1)
    out = self.proj(out)

    return out
```

Listing 1: models/Transformer/modules/layers.py: MultiHeadAttention.forward

2.2 Training Stage

To complete the training stage, we need to implement the following steps:

1. Encode the image into a latent code.
2. Randomly mask some pixels of the latent code.
3. Generate the image from the masked latent code.
4. Calculate the loss and update the model.

And the imeplementation is again not too complex, just need to figure out the correct usage of each function and tensor. First, we need to encode the image into a latent code.

```python
def encode_to_z(self, x):
    z_q, z_idx, _ = self.vqgan.encode(x)
    return z_q, z_idx.reshape(z_q.shape[0], -1)
```

Listing 2: models/VQGAN_Transformer.py: encode_to_z

Second, we need to randomly mask some pixels of the latent code. After that, we need to generate the image from the masked latent code and return the logits and the masked latent code. In the implementation, I use the `torch.distributions.Bernoulli` to sample the mask.

```python
def forward(self, x):
    z_q, z_idx = self.encode_to_z(x)

    mask = torch.distributions.Bernoulli(probs=0.5 *
    torch.ones_like(z_idx)).sample().bool()
    masked_idx = z_idx.clone()
    masked_idx[mask] = self.mask_token_id

    logits = self.transformer(masked_idx)

    return logits, z_idx
```

Listing 3: models/VQGAN_Transformer.py: forward

And finally, we need to calculate the loss and update the model. Since the z_idx is the one-hot encoded masked parts, and the logits is the probability distribution of the masked parts, we can use the `nn.CrossEntropyLoss` to calculate the loss between the logits and the masked latent code.

For evaluation, we just calculate the loss of the validation set, and is almost the same as the training stage. So I will not repeat it here.

```python
def train_one_epoch(self, train_loader):
    self.model.train()
    total_loss = 0
    idx = 0
    for data in tqdm(train_loader):
        data = data.to(self.args.device)
        logits, z_idx = self.model(data)

        loss = self.criterion(logits.reshape(-1, logits.shape[-1]),
        z_idx.reshape(-1))
        total_loss += loss.item()

        idx += 1
        loss.backward()

        if idx % self.args.accum_grad == 0:
            self.optim.step()
            self.optim.zero_grad()

    self.optim.step()
    self.optim.zero_grad()
    self.scheduler.step()

    avg_loss = total_loss / len(train_loader)
    return avg_loss
```

Listing 4: training_transformer.py: TrainTransformer.train_one_epoch

2.3 Image Inpainting

To inpaint the masked images, we need to implement the following steps:

1. Encode the image into a latent code.
2. Predict the latent code in the masked parts.
3. Decode the predicted latent code into an image.

The implementation is harder than the previous parts, so it takes me some time to figure out the correct usage of each function and tensor.

To achieve iterative inpainting, we need to iterate the above steps for several times. And in each iteration, we need to update the mask and the predicted latent code according the scheduling strategy and confidence scores.

The most easy thing in this stage is the scheduling strategy, which is implemented in the `gamma_func` function, and is just a simple function that returns a lambda 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.
    """
    if mode == "linear":
        return lambda x: 1 - x
    elif mode == "cosine":
        return lambda x: math.cos(x * math.pi / 2)
    elif mode == "square":
        return lambda x: 1 - x ** 2
    else:
        raise NotImplementedError
```

Listing 5: models/VQGAN_Transformer.py: gamma_func

In order to have a better view of the iterative inpainting process, I firstly implement the one iteration inpainting.

First, convert the image into a latent code, and then mask the latent code with a specific code, and then pass the masked latent code to the transformer. After that, apply the softmax to the logits and find the maximum probability for each token value. Then, update the mask and the predicted latent code according the scheduling strategy and confidence scores. In the process, we need to add back the original(non-masked) token values to the predicted latent code. Also, a temperature is applied to the confidence scores to add some randomness to the prediction.

```python
@torch.no_grad()
def inpainting(self, z_indices, mask_bc, mask_num, ratio):
    masked_z_idx = z_indices.clone()
    masked_z_idx[mask_bc] = self.mask_token_id
    logits = self.transformer(masked_z_idx)
    # Apply softmax to convert logits into a probability distribution
    across the last dimension.
    logits = torch.softmax(logits, dim=-1)

    # FIND MAX probability for each token value
    z_indices_predict_prob, z_indices_predict = torch.max(logits, dim=-1)

    ratio = self.gamma(ratio)
    # predicted probabilities add temperature annealing gumbel noise as
    confidence
    g = torch.distributions.Gumbel(0,
    1).sample(z_indices_predict.shape).to(z_indices_predict.device)  #
    gumbel noise
    temperature = self.choice_temperature * (1 - ratio)

    # hint: If mask is False, the probability should be set to infinity,
    so that the tokens are not affected by the transformer's prediction
    z_indices_predict_prob = torch.where(mask_bc, z_indices_predict_prob,
    torch.tensor(float('inf')))
    confidence = z_indices_predict_prob + temperature * g

    # sort the confidence for the rank
    sorted_confidence, sorted_indices = torch.sort(confidence, dim=-1)

    # define how much the iteration remain predicted tokens by mask scheduling
    mask_bc = torch.zeros_like(mask_bc)
    mask_bc[:, sorted_indices[:, :int(mask_num * ratio)]] = 1
    mask_bc = mask_bc.bool()

    # At the end of the decoding process, add back the original(non-masked) token values
    z_indices_predict[~mask_bc] = z_indices[~mask_bc]

    return z_indices_predict, mask_bc
```

Listing 6: models/VQGAN_Transformer.py: MaskGit.inpainting

The last missing piece in this stage is to run iterative inpainting for several times (some of the given code is pruned to save space). I just need to iterate the inpainting process for several times, make sure to forward the new mask and predicted latent code to the inpainting function.

```python
def inpainting(self, image, mask_b, i):  # MakGIT inference
    self.model.eval()
    with torch.no_grad():
        z_indices = self.model.encode_to_z(image)[1]  # z_indices: masked tokens (b,16*16)
        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
        # 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  # this should be updated

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

Listing 7: inpainting.py: MaskGIT.inpainting

3 Experiment Results

3.1 Iterative Decoding

For iterative decoding, we implemented three different mask scheduling functions: linear, cosine, and square. There scheduling curve is shown in Figure 1.

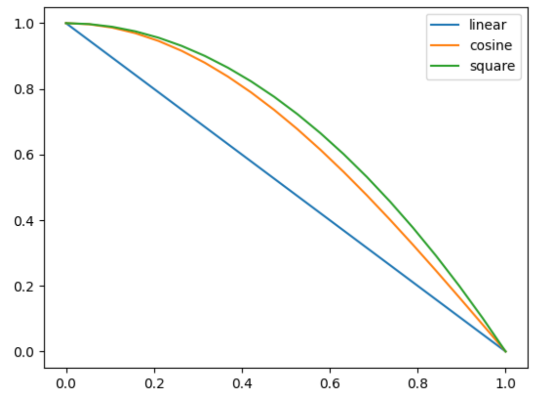


Figure 1: Mask Scheduling Curve

For their behavior in the iterative decoding, we can see the comparison in Figure 1. From the figure, we can see that the linear scheduling function decreases faster than the square and cosine scheduling functions, and cosine and square scheduling functions have similar performance.

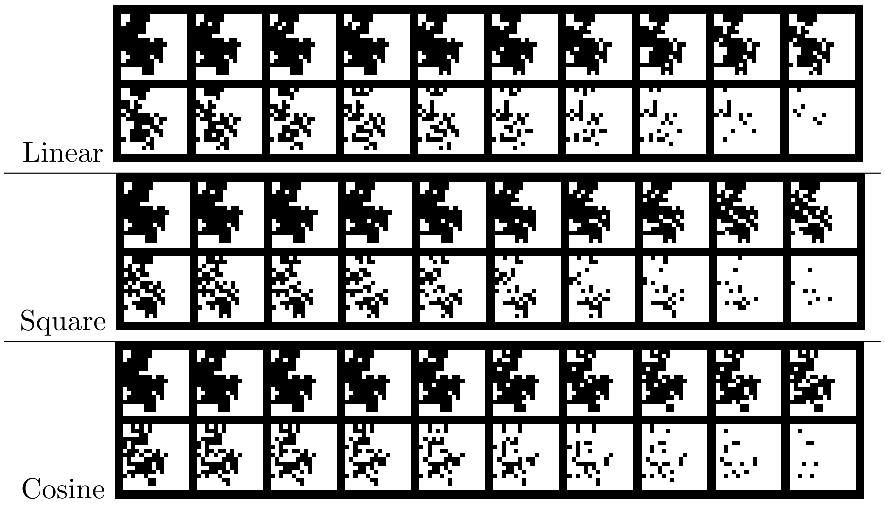


Table 1: Comparison of different mask scheduling functions in iterative decoding

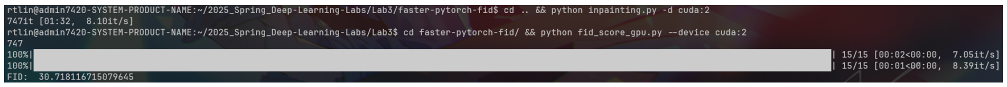
And the following figure 2 is the comparison of the iterative decoding procedure. Their performance is really close, and cosine and square has slightly better performance than linear. They all have FID scores around 31 afrer 20 iterations of decoding.

3.2 Visualization

For the best performance decoding, I found that using square or cosine scheduling with sweet spot at 1 epoch has the best performance. And the following figure 3 is the visualization of the decoding results.

3.3 FID Score

The result is shown in the screenshot 2 below, the best FID score is 30.71.



```
rtlin@admin7420-SYSTEM-PRODUCT-NAME:~/2025_Spring_Deep-Learning-Labs/Lab3/faster-pytorch-fid$ cd .. && python inpainting.py -d cuda:2
747it [01:32,  8.10it/s]
rtlin@admin7420-SYSTEM-PRODUCT-NAME:~/2025_Spring_Deep-Learning-Labs/Lab3$ cd faster-pytorch-fid/ && python fid_score_gpu.py --device cuda:2
747
100%|                                                                                          | 15/15 [00:02<00:00,  7.05it/s]
100%|                                                                                          | 15/15 [00:01<00:00,  8.39it/s]
FID:  30.718116715079645
```

Figure 2: FID Score

4 Discussion

4.1 Learning Rate Scheduler

In this project, I realized the importance of the learning rate scheduler. In the original setup, I only uses `Adam` optimizer with `CosineAnnealingLR` to schedule the learning rate, but it is converging too slow. And it converges to around 3.7 validation loss with a long long 500 epochs.

After some discussion with my classmates, I tried to use `Adam` optimizer with `LinearLR` at the beginning of the training as warmup, and `CosineAnnealingLR` after that. With this setup, the training process converges to a validation loss around 1.3 in only 150 epochs.

From this lesson, I learned that the choice of learning rate scheduler is very important for the training process.

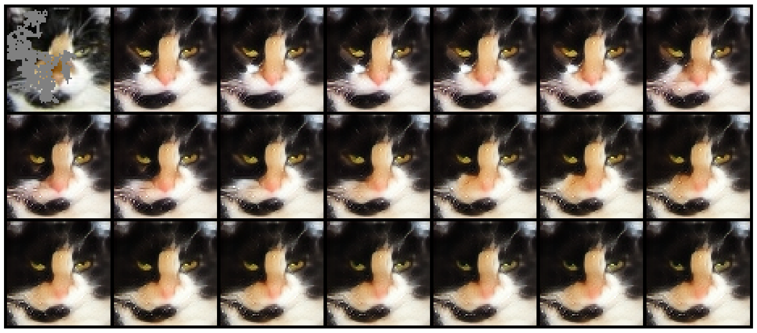
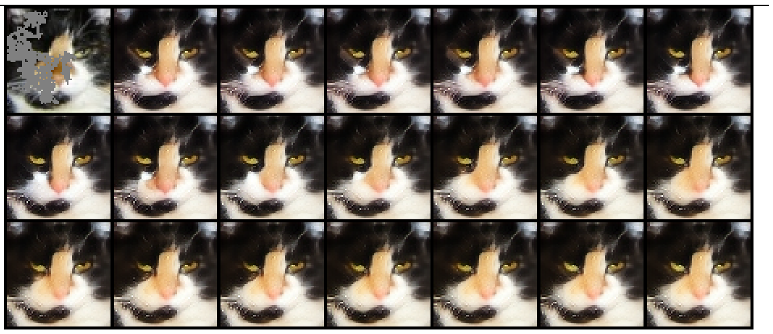
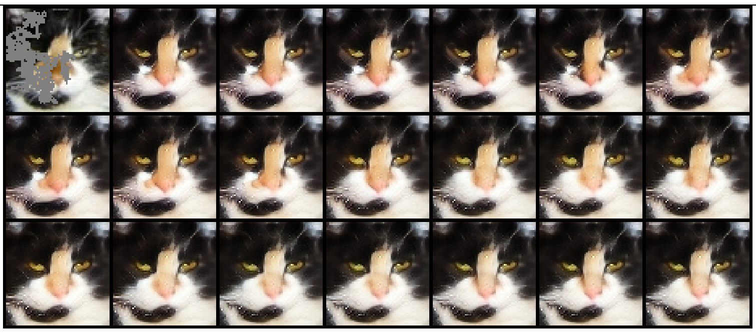
Linear	
Square	
Cosine	

Table 2: Comparison of different mask scheduling functions in inpainting

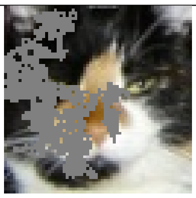
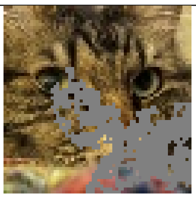
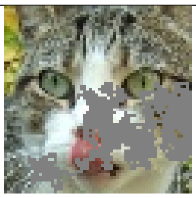
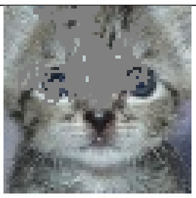
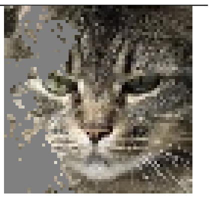
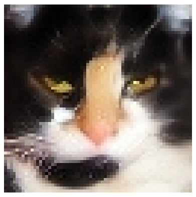
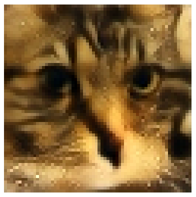
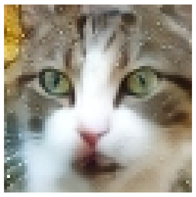
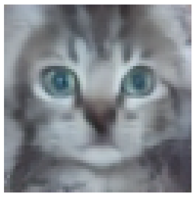
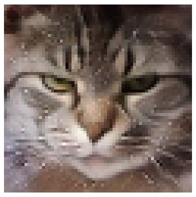
	Image 0	Image 1	Image 2	Image 3	Image 4
Masked					
Decoded					

Table 3: Visualization of inpainting results. Rows show masked input images and their corresponding decoded outputs.