Deep Learning Lab5: MaskGIT for Image Inpainting

ID: 412551030 Name: 黃羿寧

1. Introduction

Deep learning is widely used in many fields, like image recognition, medical diagnosis, and natural language processing. In this lab, we focus on using MaskGIT[1] for image inpainting, which means filling in masking parts of token and re-generate to images by using specific masking schedules. MaskGIT uses a Bidirectional Transformer to predict missing parts of the image all at once, making it faster than older methods. We will practice at three main things: the implement of multi-head attention, training the transformer, and using it for inpainting. We will also test different mask scheduling settings to see how they affect the results, which is cosine function, linear function and square function. In this report, I will explain how I built and trained the MaskGIT model, how I processed the data, and the results I got. The results show that square function of MaskGIT is the most effective method for image inpainting, with good performance of FID = 29.094 in restoring missing parts of images.

Dataset:

Cat face (12000 png files for training, 3000 png files for validation)

Masked cat face (747 png files for testing)

2. Implementation Details

A. The details of your model (Multi-Head Self-Attention)

The bidirectional transformer requires multi-head self-attention module to encode the tokens. To start from scratch, I first prepared the MultiHeadAttention class. This class starts with the setting of input dimension, number of heads, and dropout probability. It includes linear layers for projecting the input tensor into queries (q), keys (k), and values (v), and another linear layer to project the attention outputs back to the original tensor size.

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

# These linear layers are used for projecting the input tensor to queries, keys, and values tensors
        self.qkv_proj = nn.Linear(dim, 3 * dim)
        self.out_proj = nn.Linear(dim, dim)
        self.attn_drop = nn.Dropout(attn_drop)
```

In the forward method, I took the input tensor of shape (batch_size, num_image_tokens, dim), project it to the q, k, and v, and split them for each head. Then, I compute the scaled dot-product attention (scores) by taking the dot product of q and k, applying softmax to the scores to get the attention weights, and performing a weighted sum with the v. The attention outputs are then reshaped and projected back to the original dimension.

```
    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)
        batch_size, num_tokens, dim = x.shape

        # Apply the linear layer and split q, k, and v for each head
        qkv = self.qkv_proj(x)
        qkv = qkv.reshape(batch_size, num_tokens, 3, self.num_heads, self.d_k)
        qkv = qkv.permute(2, 0, 3, 1, 4)  # reorder to (3, batch_size, num_heads,
num_tokens, d_k)
        query, key, value = qkv[0], qkv[1], qkv[2]
```

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

```
        # Compute the scaled dot-product attention (MatMul of q with k, scaled by
sqrt(d_k))
        scores = torch.matmul(query, key.transpose(-2, -1)) / math.sqrt(self.d_k)
        attn = F.softmax(scores, dim=-1)  # apply softmax to scores
        attn = self.attn_drop(attn)
        weighted_avg = torch.matmul(attn, value)
        weighted_avg = weighted_avg.transpose(1, 2).reshape(batch_size, num_tokens,
dim)

        # Project the attention outputs back to the original tensor size
        output = self.out_proj(weighted_avg)
        return output
```

B. The details of your stage2 training (MVTM, forward, loss)

The MaskGIT includes the masked visual token modeling (MVTM) training of a bidirectional transformer. The procedure starts from the tokenization, random masking, prediction and model update. The schema of the training is shown below (Figure 1).

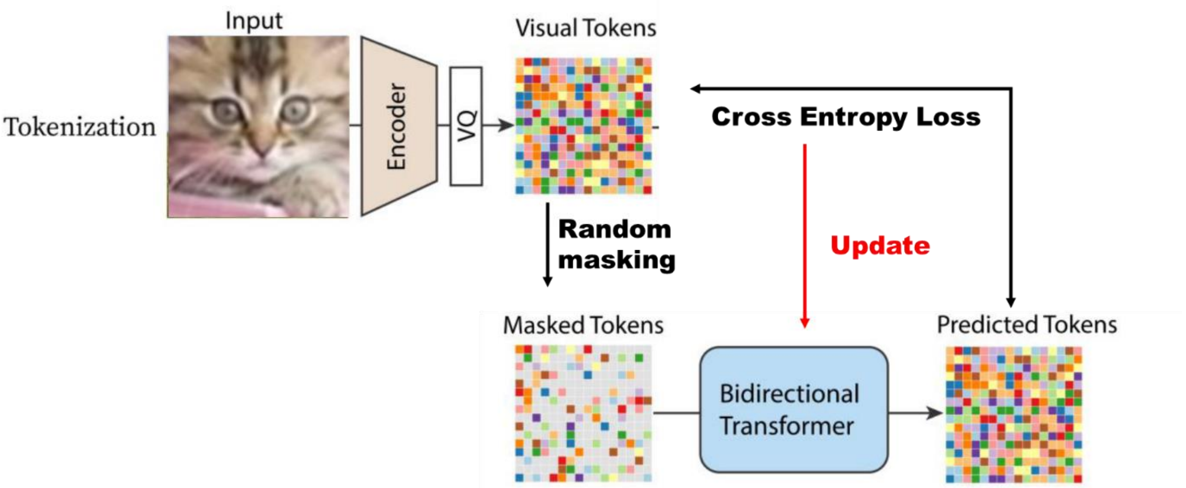


Figure 1. The schematic procedure of training the bidirectional transformer

Therefore, to train the bidirectional transformer, I prepared two classes: "TrainTransformer" in the training transformer and "MaskGit" in the VQGAN Transformer. The TrainTransformer basically comprises of initialization, training and evaluation settings, and optimizer settings.

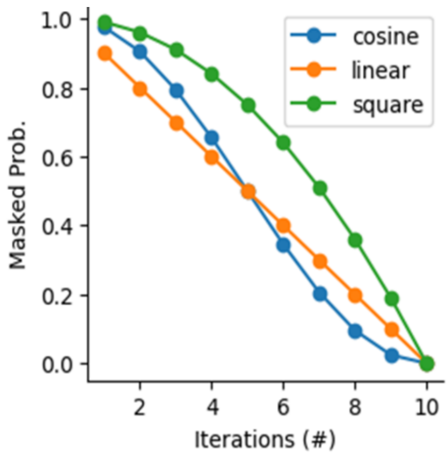
```
class TrainTransformer:
    def __init__(self, args, MaskGit_CONFIGS):
        self.args = args
        self.model = 
VQGANTransformer(MaskGit_CONFIGS["model_param"]).to(device=self.args.device)
        self.optim,self.scheduler = self.configure_optimizers()
        self.prepare_training()

    @staticmethod
    def prepare_training():
        os.makedirs(args.save_root, exist_ok=True)

    def train_one_epoch(self, train_loader):
        self.model.train()
        total_loss = 0
        # Check the structure of the batch and adjust accordingly
        for batch_idx, data in enumerate(tqdm(train_loader, desc="Training Epoch")):
            if isinstance(data, tuple) and len(data) == 2:
                images, _ = data  # If data is a tuple and only has two elements
            elif isinstance(data, tuple) and len(data) > 2:
                images, targets = data[0], data[1]  # Adjust based on actual data 
structure
            else:
                images = data  # If data is directly the images

            images = images.to(self.args.device)
            self.optim.zero_grad()
            logits, target = self.model(images)
            loss = F.cross_entropy(logits.reshape(-1, logits.size(-1)), 
target.reshape(-1))
            loss.backward()
            self.optim.step()
            total_loss += loss.item()

        average_loss = total_loss / len(train_loader)
        return average_loss

    def eval_one_epoch(self, val_loader):
        self.model.eval()
        total_loss = 0
        with torch.no_grad():
            for images in tqdm(val_loader, desc="Validation Epoch"):
                images = images.to(self.args.device)
                logits, target = self.model(images)
                loss = F.cross_entropy(logits.reshape(-1, logits.size(-1)), 
target.reshape(-1))
                total_loss += loss.item()
        average_loss = total_loss/len(val_loader)
        return average_loss
```

On the other hand, the class MaskGit starts with the import of codebook parameters, token amounts and transformer parameters. Next, since we are going to tokenize the figure, we load pre-trained vqgan and use the ecode_to_z to produce the tokens and resize to (1, 256): (`quant_z, z_indices = self.encode_to_z(x)`). Then we generate a random mask and apply to the tokens. Finally, we predict the masked tokens from the bidirectional transformer through (`logits = self.transformer(masked_indices)`). Therefore, back to the TrainTransformer class, the loss is calculated between `z_indices` and `logits`.

```python
#TODO2 step1: design the MaskGIT model
class MaskGit(nn.Module):
    def __init__(self, configs):
        super().__init__()
        self.vqgan = self.load_vqgan(configs['VQ_Configs'])

        self.num_image_tokens = configs['num_image_tokens']
        self.mask_token_id = configs['num_codebook_vectors']
        self.choice_temperature = configs['choice_temperature']
        self.gamma = self.gamma_func(configs['gamma_type'])
        self.transformer = BidirectionalTransformer(configs['Transformer_param'])

    def load_transformer_checkpoint(self, load_ckpt_path, device):
        # self.transformer.load_state_dict(torch.load(load_ckpt_path))
        checkpoint = torch.load(load_ckpt_path, map_location=device)
        self.load_state_dict(checkpoint)

    @staticmethod
    def load_vqgan(configs):
        cfg = yaml.safe_load(open(configs['VQ_config_path'], 'r'))
        model = VQGAN(cfg['model_param'])
        model.load_state_dict(torch.load(configs['VQ_CKPT_path']), strict=True)
        model = model.eval()
        return model

##TODO2 step1-1: input x fed to vqgan encoder to get the latent and zq
    @torch.no_grad()
    def encode_to_z(self, x):
        quant_z, indices, metric = self.vqgan.encode(x)
        indices = indices.view(quant_z.shape[0], -1)
        return quant_z, indices

##TODO2 step1-2:
    def gamma_func(self, mode="cosine"):
        if mode == "linear":
            return lambda r:1-r
        elif mode == "cosine":
            return lambda r:np.cos(r * np.pi / 2)
        elif mode == "square":
            return lambda r:1-r ** 2
        else:
            raise NotImplementedError

##TODO2 step1-3:
    def forward(self, x):
        quant_z, z_indices = self.encode_to_z(x)
        r = np.random.uniform()
        mask_rate = self.gamma(r)
        mask = torch.rand(z_indices.shape, device=z_indices.device) < mask_rate
        masked_indices = z_indices.masked_fill(mask, self.mask_token_id)
        logits = self.transformer(masked_indices)

        return logits, z_indices
```

C. The details of your inference for inpainting task (iterative decoding)

The inference for inpainting task which is so called iterative decoding is the crucial part of the MaskGit. This method accelerates the prediction process by applying a masking schedule. The three schedules were selected from the previous studies, which are linear function, cosine function and square function. The Figure 2 below shows the masked ratio during the 10 iterations of three methods and the corresponding equations.



Cosine function

$$ratio = \left(0.5 \times \left(1 + \cos\left(\frac{\pi \times step}{totaliter}\right)\right)\right)$$

Linear function

$$ratio = \left(1 - \frac{step}{totaliter}\right)$$

Square function

$$ratio = \left(1 - \left(\frac{step}{totaliter}\right)^2\right)$$

Figure 2. Masked ratio and equations during the 10 iterations of three masking methods: Cosine, Linear and Square

The ratio is updated through `update_ratio`.

```
    def update_ratio(self, step, method='linear', init_ratio = 1):
        if method == 'linear':
            return init_ratio*(1- (step / self.total_iter))
        elif method == 'cosine':
            return init_ratio*(0.5 * (1 + math.cos(math.pi * step /
self.total_iter)))
        elif method == 'square':
            return init_ratio*(1 - (step / self.total_iter) ** 2)
        else:
            raise ValueError("Unsupported scheduling method")
```

In this lab, our first mask is different from the from the mask schedule used in original paper. We start from the provided mask (20%-40% of the whole picture) but not 100% of the picture, so the calculation will need to consider the current masking probability and correct through the `true_count`, `mask_diff_count` during the inpainting.

$$n = \lceil \gamma(\tfrac{t}{T})N \rceil \qquad m_i^{(t+1)} = \begin{cases} 1, & \text{if } c_i < \text{sorted}_j(c_j)[n]. \\ 0, & \text{otherwise.} \end{cases}$$

Furthermore, to update the predicted tokens (unmask in the next round), I applied `token_update` to store the high-confident tokens.

```
##TODO3 step1-1: total iteration decoding
#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():
            mask_bc=mask_b
            mask_b=mask_b.to(device=self.device)
```

```
            mask_bc=mask_bc.to(device=self.device)
            true_count = mask_bc.sum().item()
            true_count1 = true_count
            mask_diff_count = 0
            total_count = mask_bc.numel()
            init_ratio = true_count / total_count
            ratio = 0
            last_epoch = False
            token_update = torch.zeros(1, 256).to(device=self.device)
            #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
                elif step == self.total_iter-1:
                    last_epoch = True
                ratio = self.update_ratio(step+1, method=self.mask_func, init_ratio =
1)

                z_indices_predict, mask_bc = self.model.inpainting(image, mask_b,
token_update, ratio, true_count, mask_diff_count, last_epoch = last_epoch)
                # Store updated tokens
                unique_mask = mask_b ^ mask_bc
                token_update += z_indices_predict * unique_mask

                #update mask
                mask_b = mask_bc
                # correct mask_diff for iterative decoding
                mask_diff_count = true_count1-mask_bc.sum().item()
```

The inpainting function in the VQGAN Transformer comprises of masked indices production, adding temperature-scaled Gumbel noise to generated tokens, sorting confidence of masked parts, searching for threshold through ratio & first masking prob., and updating new mask. The most important thing is that, the temperature-scaled Gumbel noise influences the result a lot, which will be discuss in the section below. Finally, the predicted tokens will be decoded through VQGAN decoder to generate the figure.

```
##TODO3 step1-1: define one iteration decoding
    @torch.no_grad()
    def inpainting(self, x, mask_b, token_update, ratio, true_count,
mask_diff_count, last_epoch = False):
        # encode original image to latent
        quant_z, z_indices = self.encode_to_z(x)

        # Correct latent which was updated previously
        selected_z_indices = z_indices
        non_zero_indices = token_update != 0
        token_update = token_update.long()
        try:
            selected_z_indices[non_zero_indices] = token_update[non_zero_indices]
            # print(selected_z_indices)
        except Exception as e:
            selected_z_indices = selected_z_indices
            print(f"An error occurred: {e}")
        # print(selected_z_indices.shape)

        # Mask tokens through masked_fill
        selected_z_indices = selected_z_indices.masked_fill(mask_b,
self.mask_token_id)
        # print(selected_z_indices)
```

```
        # Biderectional transformer prediction
        logits = self.transformer(selected_z_indices)
        # Convert logits to a probability distribution
        logits = F.softmax(logits, dim=-1)

        # Simulate Gumbel noise for stochastic sampling
        g = -torch.log(-torch.log(torch.rand_like(logits)))
        temperature = self.choice_temperature *(1-ratio)  # Adjust temperature for
annealing
        # Calculate confidence scores by adding temperature-scaled Gumbel noise
        confidence = torch.log(logits) + temperature * g
        # Get the predicted indices with the highest confidence
        z_indices_predict_prob, z_indices_predict = torch.max(confidence, dim=-1)

        # Addback original tokens (unmasked ones)
        inverted_mask = ~mask_b
        z_indices_predict[inverted_mask] = selected_z_indices[inverted_mask]

        # calculated threshold using only masked token prediction (# true means
mask)
        count_threshold_z_indices = z_indices_predict_prob[mask_b]
        if not last_epoch: #(last epoch no threshold problem)
            # Update the mask: Reduce the number of masked indices based on
confidence and ratio
            # Sort confidence to find a threshold for high confidence predictions
            sorted_confidence, _ = torch.sort(count_threshold_z_indices,
descending=True)
            threshold_index = int((1-ratio)*true_count)-mask_diff_count # correct
through mask_diff amount
            try:
                threshold_value = sorted_confidence[threshold_index].unsqueeze(-1)
            except:
                threshold_index = int((1 - ratio) * true_count)
                print(threshold_index, ':error')
                threshold_value = sorted_confidence[threshold_index].unsqueeze(-1)
            # Update the mask where confidence is below threshold (above threshold =
no mask)
            z_indices_predict_prob[~mask_b] = float('inf')
            mask_bc = z_indices_predict_prob > threshold_value
            mask_bc = ~mask_bc
        else:
            threshold_value = float('-inf')
            mask_bc = z_indices_predict_prob > threshold_value
            mask_bc = ~mask_bc
        return z_indices_predict, mask_bc
```

3. Experimental results

A. The best testing fid

- **The setting about training strategy, mask scheduling parameters**

  The training of the bidirectional transformer includes an implement of Adam optimizer with a learning rate of 0.00005. The total epochs are 60. For the inference stage in image inpainting task, the epoch with the lowest validation loss (the 54st epoch) was selected to predicted the masked tokens (Figure 3). The training loss in this epoch is 1.325, and the validation loss is 1.377.

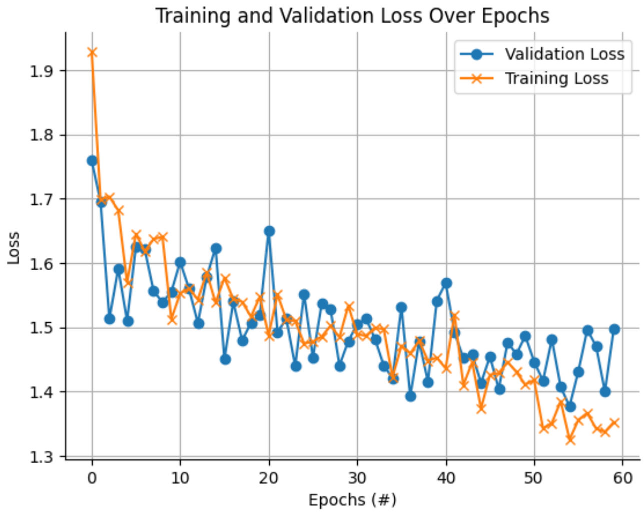
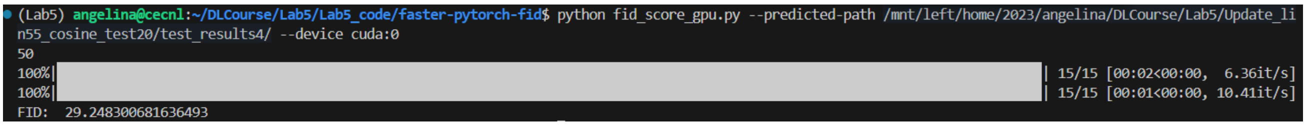


Figure 3. The training and validation loss over 60 epochs

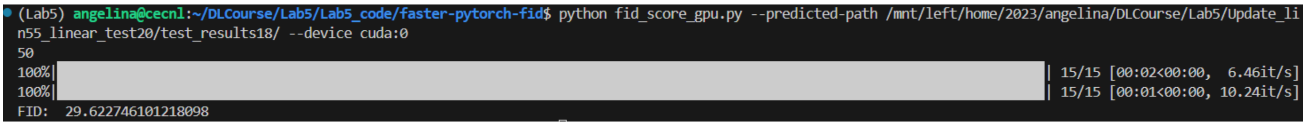
- **Screenshot of the best testing fid**

  The best testing fid is shown below in Figure 4. The fid I gained from the result is 29.094 (square function), 29.248 (cosine function), and 29.623 (linear function), showing its medium quality of the reconstructed figure.

Cosine: FID = 29.248 (t = 5, T = 20, choice_temp = 1)



Linear: FID = 29.623 (t = 19, T = 20, choice_temp = 1)



Square: FID = 29.094 (t = 18, T = 20, choice_temp = 1)

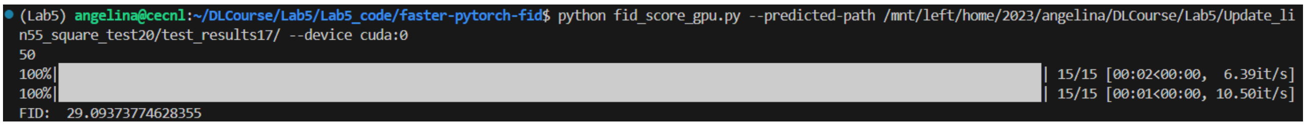


Figure 4. The screenshot of the best fid figure

- **Predicted image, Mask in latent domain with mask scheduling**
  The predicted image and the mask with cosine mask scheduling are shown below (Figure 5). The test_2 was randomly selected to show the effect of the reconstruction and masking schedule. The masking probability was decreased using cosine function. In the figure at the left side, we can see the making probability is perfectly in line with the gamma (cosine) function.

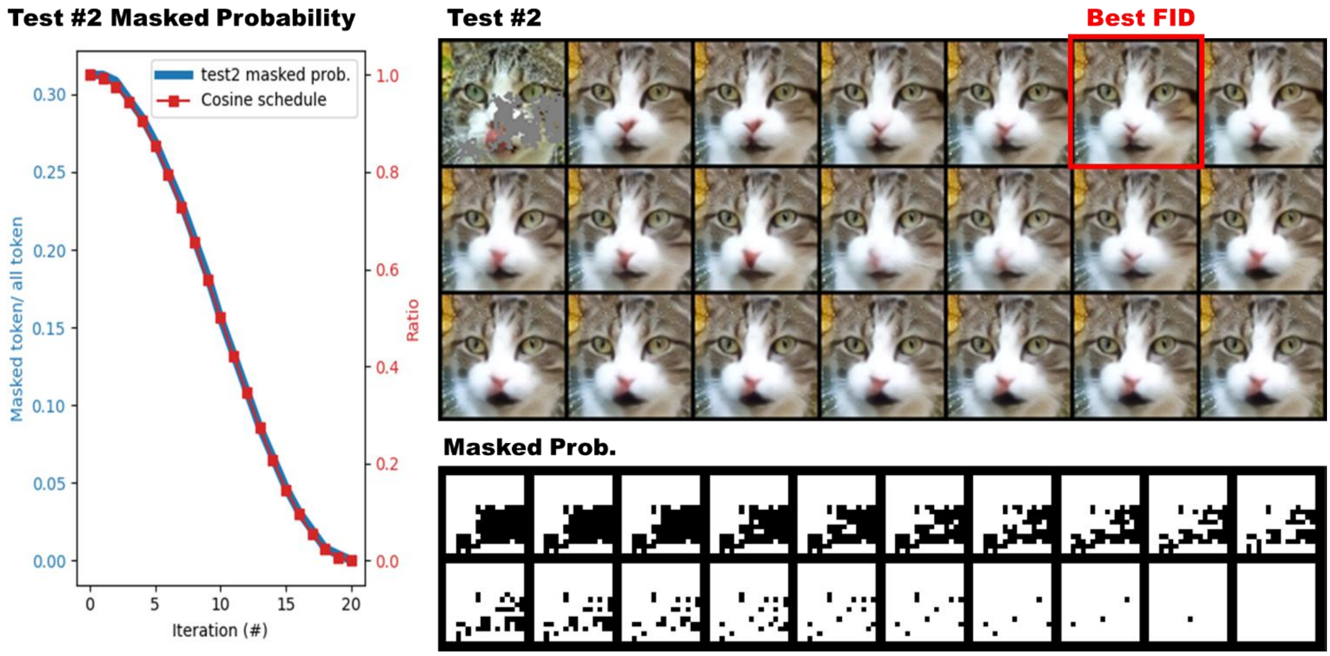


Figure 5. The predicted image and the mask in latent domain with cosine mask scheduling

B. Comparison figures with different mask scheduling parameters setting

To compare three mask scheduling, I draw the masking probability of test 4 using different masking schedules of cosine function, linear function and square function with a T of 10 (Figure 6).

Figure 6. The mask probability and mask in latent of three methods (T=10)

On the other hand, the results of three different mask scheduling were significantly different. As the result shown in Figure 7, the FID among iterative decoding (T = 20) shows that square scheduling outperforms other scheduling, with the best FID of 29.094. The linear scheduling and cosine scheduling have a limited performance of 29.623 and 29.248 FID. The test 2 reconstruction qualities were similar under different schedule methods in this lab practice.

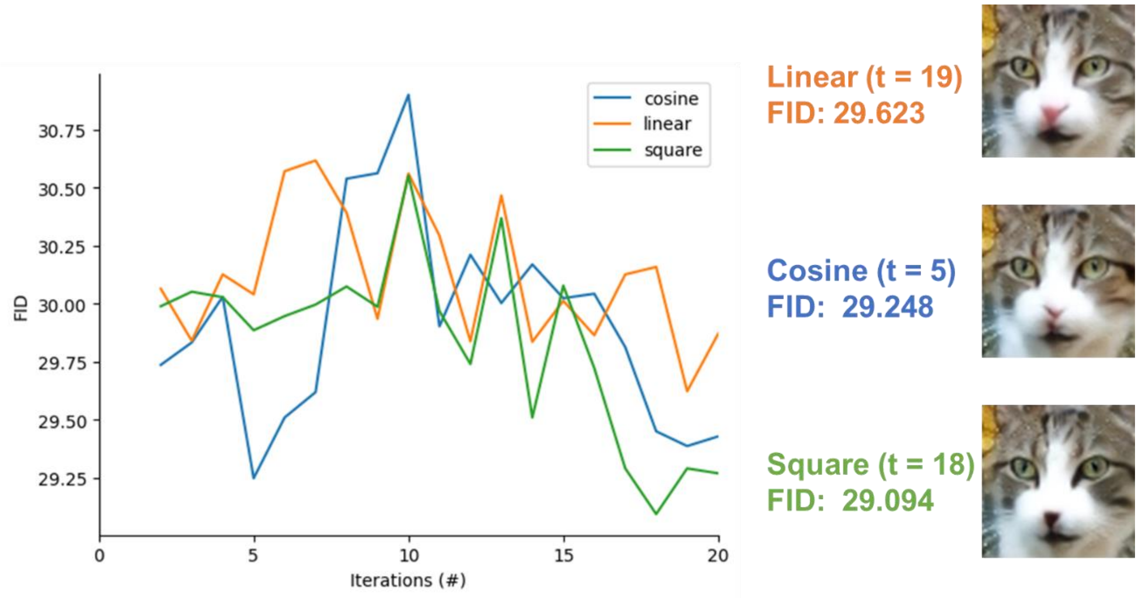


Figure 7. The predicted image 2 and its FID under different mask scheduling methods & different iterations (#)

4. Discussion

Our result shows that all three function showed influence on reconstruction speed and quality, which is in line with the previous study[1]. However, the FID value was relatively high in our practice. Moreover, the mask scheduling did not show significant difference between each other. Two reasons might explain this result: the effectiveness of the bidirectional transformer and the consistency of the equation used. In this practice, the performance of the transformer did not converge and was early stopped at 60 epochs due to time and resource concerns. It should be trained for more epochs to increase the effectiveness of the MVTM evaluation. Additionally, the equation used in the cosine masking schedule produced a different curve pattern compared to the original paper[1]. Although both were considered cosine degradation of ratio, using a more suitable function would be better.

On the other hand, during prediction and reconstruction, the temperature annealing method was applied in the previous study, so I experimented with different parameters to see the effects. I randomly selected three temperature values: 0, 1, 2, and 3. As shown in Figure 8, the figure with higher temperature scaling showed more diversity in the reconstruction, indicating its potential to control the generative model. However, to achieve the highest reconstruction quality in this practice, a temperature parameter of 1 was selected based on the experimental results.

In conclusion, in this practice, we built the multi-head self-attention module from scratch and trained a bidirectional transformer. We then predicted the masked tokens and regenerated the figure using the VQGAN decoder. This series of practices helped us understand the mechanism of image inpainting well. Furthermore, since several steps of machine learning are inspired by the human, the MaskGit shows that it will be useful to develop efficient and effective methods based on human patterns.

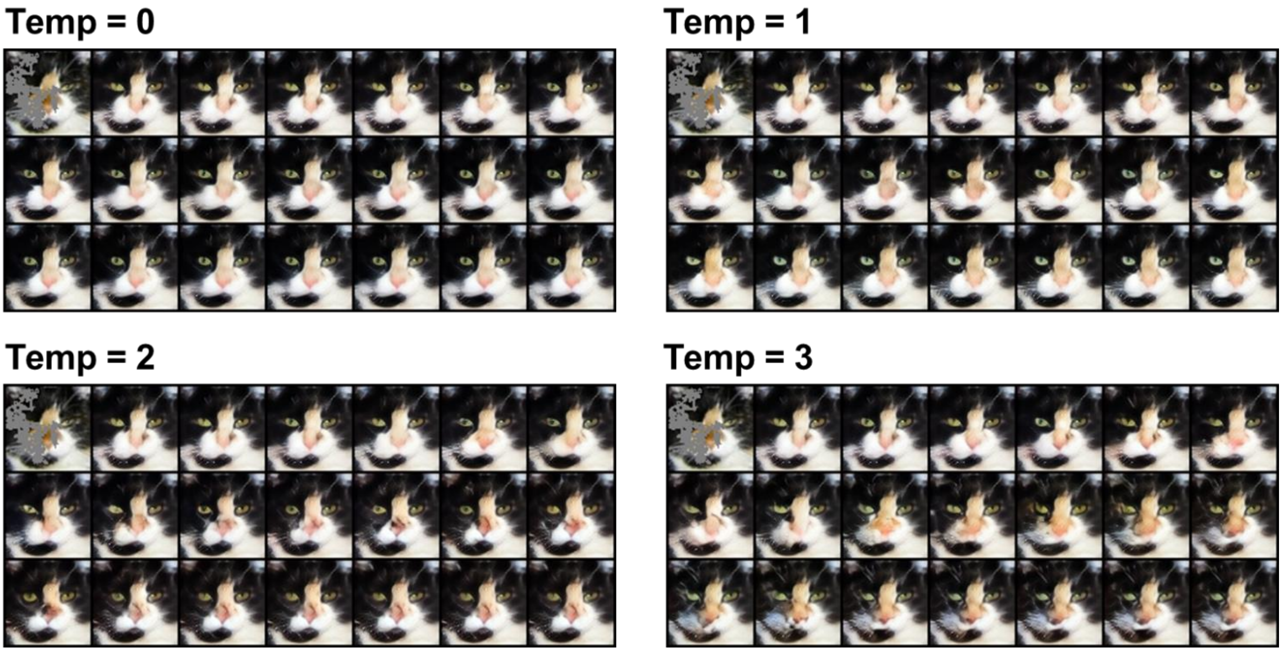


Figure 8. The predicted figure under different temperature settings (T=20)

5. 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*, 2022, pp. 11315-11325.