Lab 5 MaskGIT

109550121 温柏萱

Introduction

This lab aims to implement MaskGIT for the image inpainting task, which is an advanced generative bidirectional image transformer. Image inpainting aimed at filling in missing or damaged parts of an image to restore its original visual coherence. We solve this task by MaskGIT, by first encoding the image into codebook representations, we randomly place masks on the image and train a bidirectional transformer to predict the correct codebook indices.

When performing inpainting, we adopted the Masked Visual Token Modeling, which was inspired by human drawing logic, by having a draft first then refining the draft to a final work, the approach iteratively refines the masked tokens and only kept a portion of tokens that were the best. We compared different decoding strategies yield different results, and the fact that lower loss doens't lead to higher performances.

Implementation Details

Multi-Head Self-Attention (2 implementations)

Multi-head attention allows the model to attend to different positions of the input sequence and capture different aspects of the input information by using multiple attention heads. Each attention head performs a scaled dot-product attention, and the outputs of all the heads are concatenated and projected to obtain the final output.

The variables `self.d_v` and `self.d_k` represent the value and key/query vectors for each head, three linear layers `Q, K, V` project the input `x` to query vectors, key vectors and value vectors respectively. The layer `W_out` connects all attention heads back to the original `dim` dimension. The `attn_drop` layer is a dropout layer to prevent overfitting.

The `forward` function of `MultiHeadAttention` takes the input `x` and projects the input to query vector `q`, key vector `k` and value vector `v`, the attention is

$$\text{Attention}(q, k, v) = \text{softmax}(\frac{qk^T}{\sqrt{d_k}})v \tag{1}$$

which is calculated by the lines `qk = torch.matmul(q.transpose(1, 2), k) * (1.0 / math.sqrt(self.d_k))`, `attention = torch.softmax(qk, dim=-1)`, and `output = torch.matmul(attention, v.transpose(2, 3)).transpose(1, 2)`. Finally, drop a portion of the attention weights to prevent overfitting.

```
class MultiHeadAttention(nn.Module):
    def __init__(self, dim=768, num_heads=16, attn_drop=0.1):
        super(MultiHeadAttention, self).__init__()
```

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

        self.Q = nn.Linear(dim, dim)
        self.K = nn.Linear(dim, dim)
        self.V = nn.Linear(dim, dim)

        self.W_out = nn.Linear(dim, dim)
        self.attn_drop = nn.Dropout(attn_drop)


    def forward(self, x):
        batch_size, num_image_tokens, dim = x.size()
        q = self.Q(x).view(batch_size, num_image_tokens, self.num_heads, self.d_k)
        k = self.K(x).view(batch_size, num_image_tokens, self.num_heads, self.d_k).permute(0, 2, 3, 1) # (batch_size, nhead, d_head, time)
        v = self.V(x).view(batch_size, num_image_tokens, self.num_heads, self.d_v).permute(0, 2, 3, 1) # (batch_size, nhead, d_head, time)

        qk = torch.matmul(q.transpose(1, 2), k) * (1.0 / math.sqrt(self.d_k))
        attention = torch.softmax(qk, dim=-1)

        output = torch.matmul(attention, v.transpose(2, 3)).transpose(1, 2) # (batch_size, time, nhead, d_head)
        output = output.contiguous().view(batch_size, -1, self.dim) # (batch_size, time, d_model)

        output = self.attn_drop(self.W_out(output))
        return output
```

In order to test the correctness of the implementation between my own version and a "modified" version online, I consulted the implementation MaskGIT-PAT.

Both implementations are interchangeable since the first implementation had the `Q`, `K`, and `V` being fully connected layers of shape `dim x dim` while the second implementation had `num_heads` of `dim x (dim // num_heads)` linear layers.

```
class Attention(nn.Module):
    def __init__(self, dim=768, num_heads=16):
        super(Attention, self).__init__()
        self.dim = dim
        self.num_heads = num_heads
        self.d_v = dim // num_heads
        self.d_k = dim // num_heads

        self.Q = nn.Linear(dim, dim // num_heads)
        self.K = nn.Linear(dim, dim // num_heads)
        self.V = nn.Linear(dim, dim // num_heads)

        self.attn_drop = nn.Dropout(0.1)

    def forward(self, x):
        batch_size, num_image_tokens, dim = x.size()
        # q = self.Q(x).view(batch_size, num_image_tokens, self.num_heads,
self.d_k)
        # k = self.K(x).view(batch_size, num_image_tokens, self.num_heads,
self.d_k).permute(0, 2, 3, 1) # (batch_size, nhead, d_head, time)
        # v = self.V(x).view(batch_size, num_image_tokens, self.num_heads,
self.d_v).permute(0, 2, 3, 1) # (batch_size, nhead, d_head, time)
        q = self.Q(x)
        k = self.K(x)
        v = self.V(x)

        qk = torch.matmul(q, k.transpose(1, 2)) * (1.0 /
math.sqrt(self.d_k))
        attention = torch.softmax(qk, dim=1)
        attention = self.attn_drop(attention)
        attention = torch.matmul(attention, v)

        return attention
```

```python
class MultiHeadAttention(nn.Module):
    def __init__(self, dim=768, num_heads=16):
        super(MultiHeadAttention, self).__init__()
        self.self_attention_heads = nn.ModuleList([Attention(dim,
num_heads) for _ in range(num_heads)])
        self.W_out = nn.Linear(dim, dim)

    def forward(self, x):
        for i, attn_head in enumerate(self.self_attention_heads):
            if i == 0:
                output = attn_head(x)
            else:
                output = torch.cat((output, attn_head(x)), axis=-1)
        output = self.W_out(output)
        return output
```

This implementation modulizes the `MultiHeadAttention` into multiple `Attention` instances, where each `Attention` instance has its own `Q,D,V` networks. For each attention head, calculate the attention using Equation 1, then concatenate all the outputs of the attention heads and project the output using a linear layer of shape `dim x dim`.

MVTM Training (Masked Visual Token Modeling)

One-Iteration Training

```python
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'] # 256
        self.mask_token_id = configs['num_codebook_vectors'] # 1024
        self.choice_temperature = configs['choice_temperature']# 4.5
        self.gamma = self.gamma_func(configs['gamma_type'])
        self.transformer =
BidirectionalTransformer(configs['Transformer_param'])
        # For inpainting inference
        self.image = None
        self.image_indices = None
```

```
        self.init_mask = None

    def load_transformer_checkpoint(self, load_ckpt_path):
        self.transformer.load_state_dict(torch.load(load_ckpt_path),
strict=False)

    @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, _ = 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"):
        """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.

        ratio:   The uniformly sampled ratio [0, 1) as input.
        Returns: The mask rate (float).

        """
        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):
        _, z_indices = self.encode_to_z(x)
        r = math.floor(self.gamma(np.random.uniform()) *
z_indices.shape[1])
        sample = torch.rand(z_indices.shape,
device=z_indices.device).topk(r, dim=1).indices
        mask = torch.zeros(z_indices.shape, dtype=torch.bool,
device=z_indices.device)
        mask.scatter_(dim=1, index=sample, value=True)

        masked_indices = self.mask_token_id * torch.ones_like(z_indices,
device=z_indices.device)

        a_indices = mask * z_indices + (~mask) * masked_indices # Apply
mask to z_indices


        logits = self.transformer(a_indices)

        return logits, z_indices
```

gamma_func

I implemented three mask scheduling method, given the mask ratio $r \in [0, 1]$

$$\begin{aligned} \text{cosine} &: \cos(\frac{\pi r}{2}) \\ \text{linear} &: 1 - r \\ \text{square} &: 1 - r^2 \end{aligned} \tag{2}$$

`forward`

For each iteration of an input `x`, which is an image, we first encode it to the codebook indices, which is `z_indices`, and determine the ratio `r` of the portion on the image to be masked, which is calculated by `gamma_func`. The mask was randomly placed and applied to the image, then feed the masked image to the transformer to obtain the logits.

Training One Epoch

```
def train_one_epoch(self, train_loader, cur_epoch):
        ret_loss = 0.0
        for i, data in enumerate(tqdm(train_loader, desc=f"Training Epoch
{cur_epoch}")):
            inputs = data.to(args.device)

            logits, targets = self.model(inputs)

            # Flatten the outputs and targets to fit cross-entropy
expectation
            loss = F.cross_entropy(logits.view(-1, logits.shape[-1]),
targets.view(-1))
            loss.backward()

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

            ret_loss += loss.item()

        average_loss = ret_loss / len(train_loader)
        if args.scheduler == 'warm-up':
            lr = trainer.scheduler.lr
        else:
            lr = trainer.scheduler.get_last_lr()[0]
        print(f"Average training loss for epoch {cur_epoch}:
{average_loss:.4f}, lr: {lr}")
        return average_loss
```

For every epoch, we feed the input data into the model and calculate the loss as the cross entropy between `logits` and `target`, which is the encoded image `z_indices`. Then perform back propagation on the model to minimize the loss.

Inpainting Inference

```
def inpainting(self,image,mask_b,i, true_scheduling=False): #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) #z_indices: masked tokens (b,16*16)
            # print(f"z_indices: {type(z_indices)}")
            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 + 1) / self.total_iter #this should be updated
                z_indices_predict, mask_bc = self.model.inpainting(z_indices_predict, mask_bc, ratio, true_scheduling, mask_b)
```

```
                z_indices_predict = torch.where(mask_bc,
z_indices_predict, z_indices)
                z_indices_predict = torch.clamp(z_indices_predict, 0,
1023)
                #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

                vutils.save_image(dec_img_ori,
os.path.join("test_results/" + self.save_root + "/" + str(step),
f"image_{i:03d}.png"), nrow=1)

                #demo score
                vutils.save_image(maska, os.path.join("mask_scheduling/" +
self.save_root + "/" + str(step), f"test_{i}.png"), nrow=10)
                vutils.save_image(imga, os.path.join("imga/" +
self.save_root + "/" + str(step), f"test_{i}.png"), nrow=7)
```

Before iterative decoding, the image was first decoded and stored in the variable `z_indices` for later recovering, `z_indices_predict` is set to be the same as `z_indices` as the initial image was the same as the masked image, `mask_bc` was the dynamic mask that would be updated in the iterative decoding, and `ratio` was set to 0.

In each iteration, `ratio` was first updated with $\frac{\text{current iter}}{\text{total\_iter}}$ to determine the predicted portion to be kept, then use the `inpainting` function in the model to predict and update the `z_indices_predict` and mask `mask_bc`.

Since we wish to preserve the unmasked part in the image, we add back the part that was not masked by the original image to the predicted image, which is `z_indices_predict, mask_bc = self.model.inpainting(z_indices_predict, mask_bc, ratio, true_scheduling, mask_b)`, since the prediction may include some values out of bound, we use `z_indices_predict = torch.clamp(z_indices_predict, 0, 1023)` to ensure the predicted indices lies within $[0, 1023]$.

For each iteration, save all the results for testing the best sweet spot.

```
def inpainting(self, z_indices, mask_b, ratio, true_scheduling=False, mask_original=None):
        z_indices = torch.where(mask_b, self.mask_token_id, z_indices)
        logits = self.transformer(z_indices) # [1, 256, 1025]
        #Apply softmax to convert logits into a probability distribution across the last dimension.
        probs = torch.softmax(logits, dim=-1)  # [1, 256, 1025]
        infinite_mask = torch.full_like(probs, float('inf'))
        probs = torch.where(mask_b.unsqueeze(-1), probs, infinite_mask)
        z_indices_predict_prob, z_indices_predict = probs.max(dim=-1) # [1, 256]

        ratio = self.gamma(ratio)
        g = torch.distributions.gumbel.Gumbel(0, 1).sample(z_indices_predict_prob.shape).to("cuda")
        temperature = self.choice_temperature * (1 - ratio)
        confidence = z_indices_predict_prob + temperature * g
        sorted_confidence, _ = torch.sort(confidence, dim=-1)
        if true_scheduling:
            mask_len = torch.unsqueeze(torch.floor(mask_original.sum().view([1]) * ratio), 1)
        else:
            mask_len = torch.unsqueeze(torch.floor(mask_b.sum().view([1]) * ratio), 1)
        mask_len = torch.maximum(torch.zeros_like(mask_len), torch.minimum(torch.sum(mask_b, dim=-1, keepdim=True)-1, mask_len))
        cut_off = torch.take_along_dim(sorted_confidence, mask_len.to(torch.long), dim=-1)
        masking = (confidence < cut_off) # preserve the max confidence part
```

```python
        z_indices_predict = torch.where(mask_b, z_indices_predict,
z_indices)
        z_indices_predict = torch.clamp(z_indices_predict, 0, 1023)

        return z_indices_predict, masking
```

The image including predictions is passed in with `z_indices`, we apply the mask to the image with `z_indices = torch.where(mask_b, self.mask_token_id, z_indices)` to let the transformer predict the masked part. The transformer will output the `logits`, we use the softmax function to transform it into a probability distribution and set the probability of unmasked part of the image to be infinite (`probs = torch.where(mask_b.unsqueeze(-1), probs, infinite_mask)`). The predicted probability `z_indices_predict_prob` and predicted mask `z_indices_predict` are obtained from taking the maximum probability from the probability distrbution of `logits`.

The size of the updated mask is determined by the mask scheduling function `gamma_func`, Gumbel noise is sampled from a Gumbel distribution with location `0` and scale `1`. This noise introduces randomness to the confidence scores, which helps in exploring different possible values during inpainting. Temperature is used to adjust the scale of randomness. `mask_len` calculates the number of masked positions based on the sum of `mask_b` and the adjusted `ratio` and `cut_off` determines the confidence threshold for masking by selecting the corresponding value from `sorted_confidence`. Finally insert the unmasked part that shouldn't be altered into the results and insert predictions into where the image was masked, and return the predictions and updated mask.

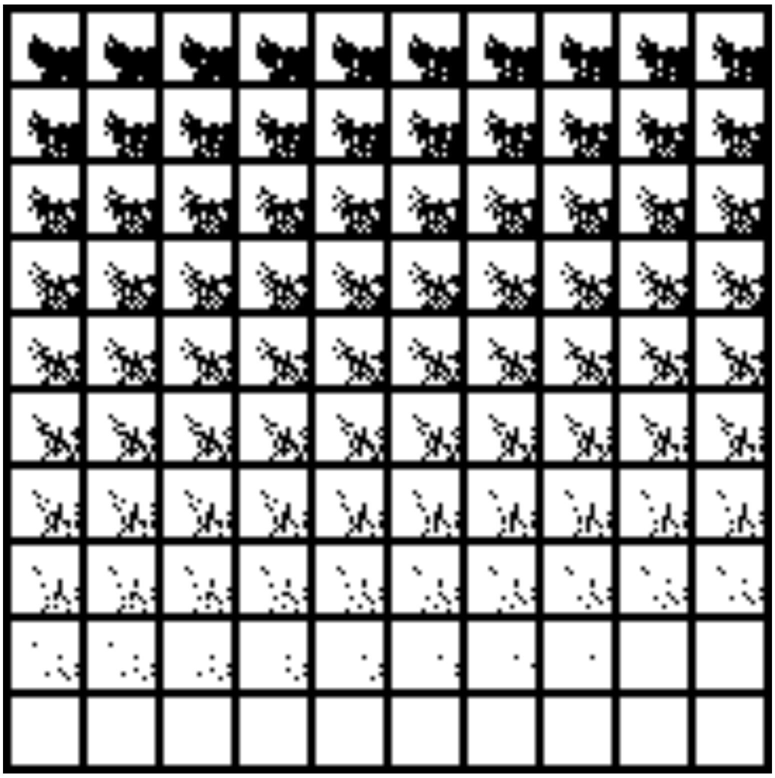
The parameter `true_scheduling` is just to determine whether we choose to use the mask size from the previous iteration or the original mask size.

Experimental Results

Best FID: 40.286599142976684

Result Image

Mask Scheduling



Decode Process

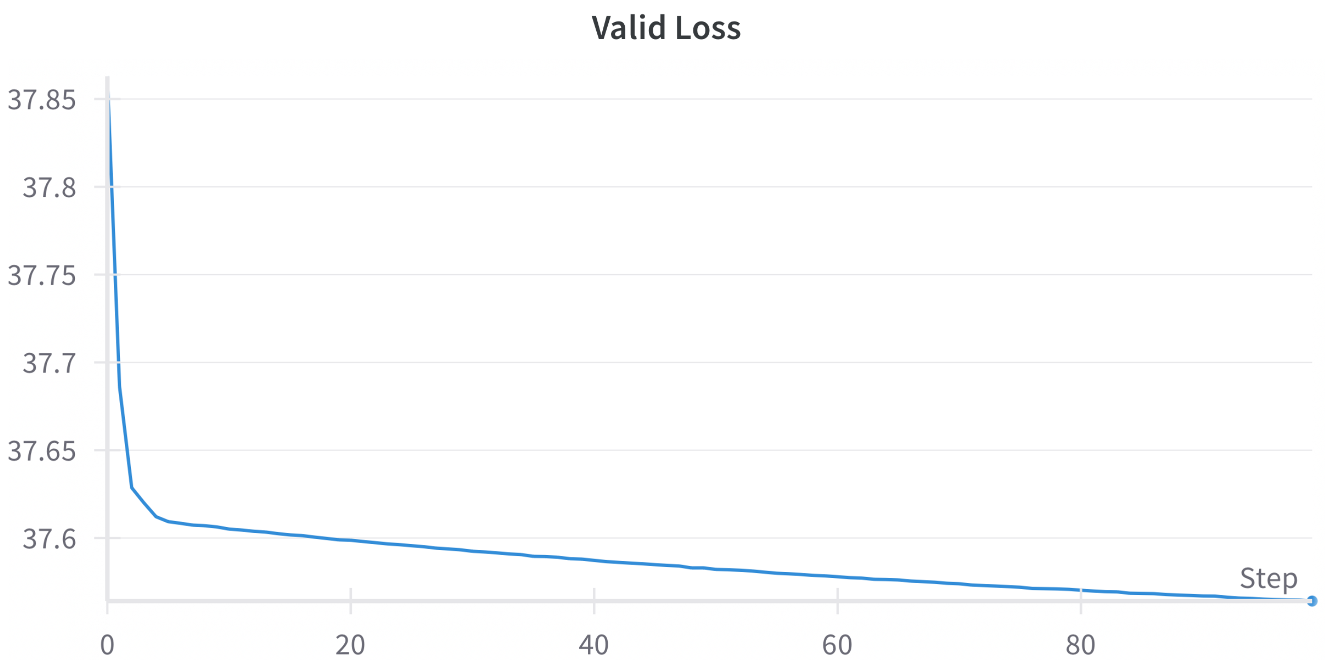
Mask Scheduling Parameters

Parameter	Setting
`total-iter`	100
`sweet-spot`	7
`mask-func`	Cosine

Training Detail

Learning Rate Scheduler

In the begining I used a multi-stage learning rate scheduling but all didn't show strong learning behavior on reducing the loss as shown in the following figure.



The reduction was minor (only 0.3), therefore, I adapted the learning rate scheduler from the repo MaskGIT-pytorch.

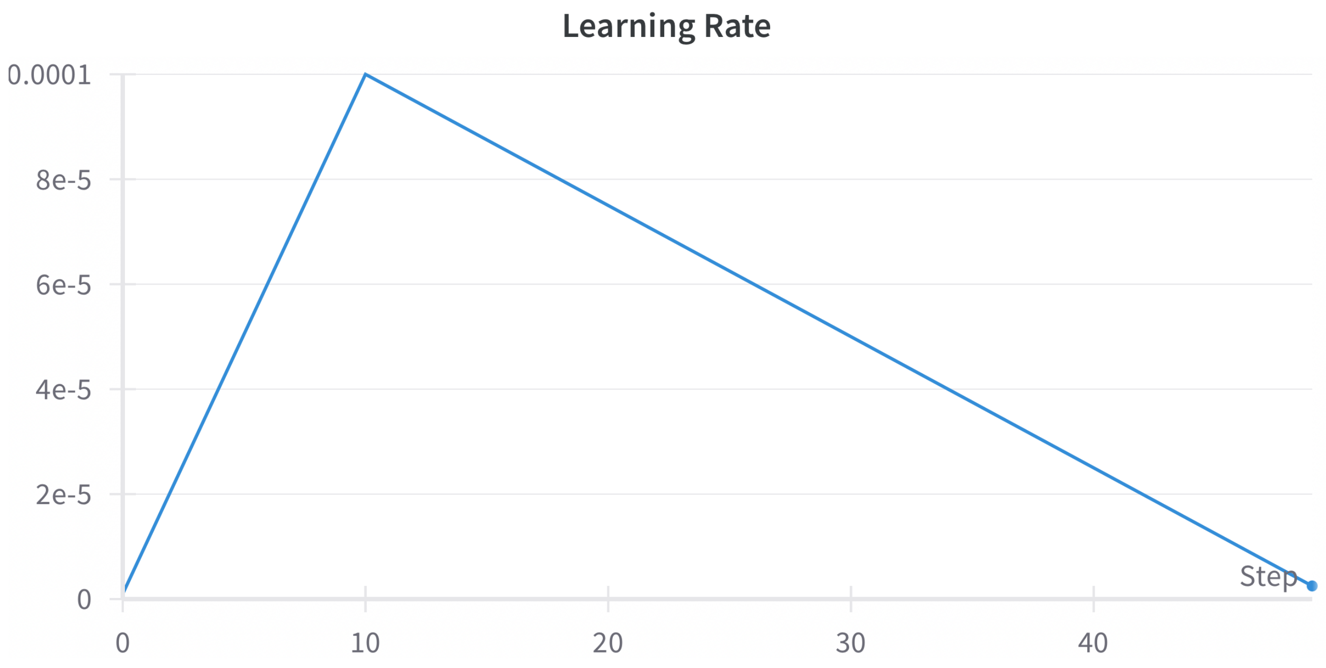
```
class WarmupLinearLRSchedule:
    """
    Implements Warmup learning rate schedule until 'warmup_steps', going from 'init_lr' to 'peak_lr' for multiple optimizers.
    """
    def __init__(self, optimizer, init_lr, peak_lr, end_lr, warmup_epochs, epochs=100, current_step=0):
```

```
        self.init_lr = init_lr
        self.peak_lr = peak_lr
        self.optimizer = optimizer
        if warmup_epochs != 0:
            self.warmup_rate = (peak_lr - init_lr) / warmup_epochs
        else:
            self.warmup_rate = 0
        # print(f"end_lr: {end_lr}, peak_lr: {peak_lr}, epochs: {epochs}, warmup_epochs: {warmup_epochs}")
        self.decay_rate = (end_lr - peak_lr) / (epochs - warmup_epochs)
        self.update_steps = current_step
        self.lr = init_lr
        self.warmup_steps = warmup_epochs
        self.epochs = epochs
        if current_step > 0:
            self.lr = self.peak_lr + self.decay_rate * (current_step - 1 - warmup_epochs)

    def set_lr(self, lr):
        print(f"Setting lr: {lr}")
        for g in self.optimizer.param_groups:
            g['lr'] = lr

    def step(self):
        if self.update_steps <= self.warmup_steps:
            lr = self.init_lr + self.warmup_rate * self.update_steps
        # elif self.warmup_steps < self.update_steps <= self.epochs:
        else:
            lr = max(0., self.lr + self.decay_rate)
        self.set_lr(lr)
        self.lr = lr
        self.update_steps += 1
        return self.lr
```

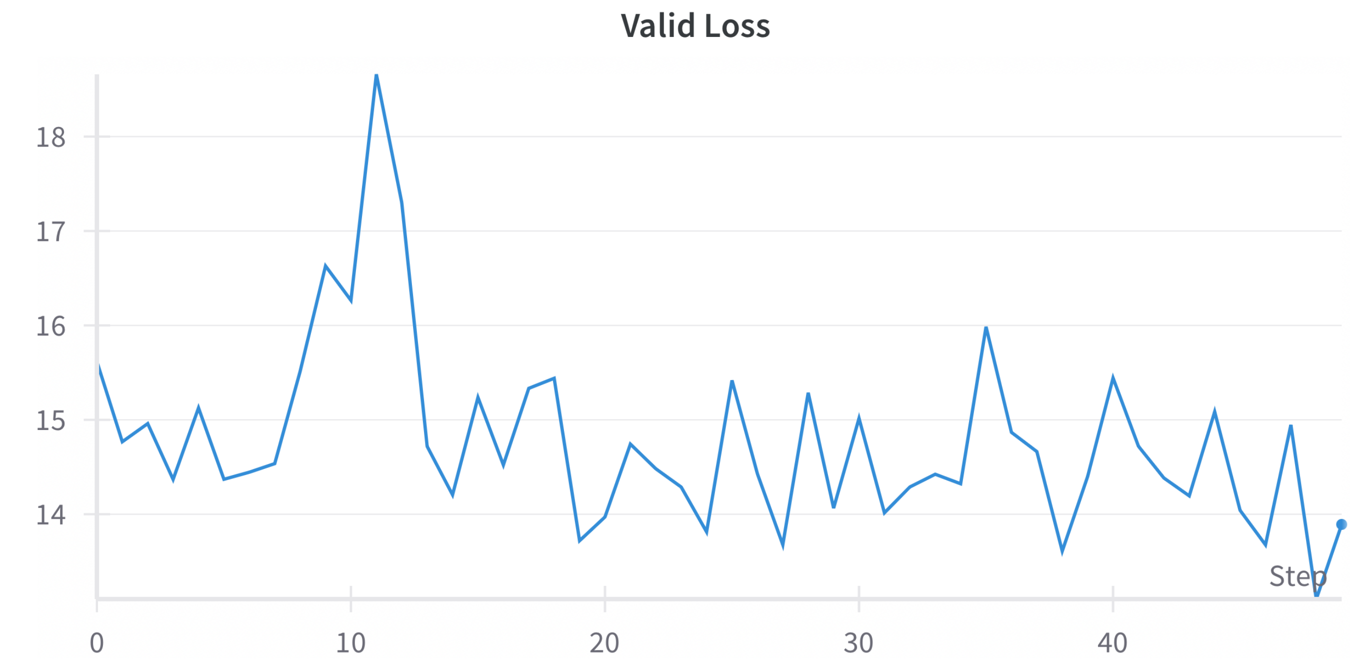
The `WarmupLinearLRSchedule` class implements a learning rate scheduler with a warm-up phase followed by a linear decay phase. If `warmup_epochs` is not 0, `warmup_rate` is calculated as the slope of warming up so that the learning rate starts from `init_lr` and reaches `args.lr` by `warmup_epochs`. After reaching the peak learning rate, it strats decaying until the end of traning when it reaches `end_lr`. The effect can be visualized as the following figure.



We configure the `WarmupLinearLRSchedule` as the following:

```
scheduler = WarmupLinearLRSchedule(
                optimizer=optimizer,
                init_lr=1e-6,
                peak_lr=args.learning_rate,
                end_lr=0.,
                warmup_epochs=10,
                epochs=args.epochs,
                current_step=args.start_from_epoch
            )
```

Such learning rate scheduling resulted in a more reasonable learning curve and a more dynamic environment that allows the agent to escape from local minima.



Settings

Parameter	Value
batch size	10
epochs	50
Warmup-epochs	10
Peak_lr	0.0001
End_lr	0

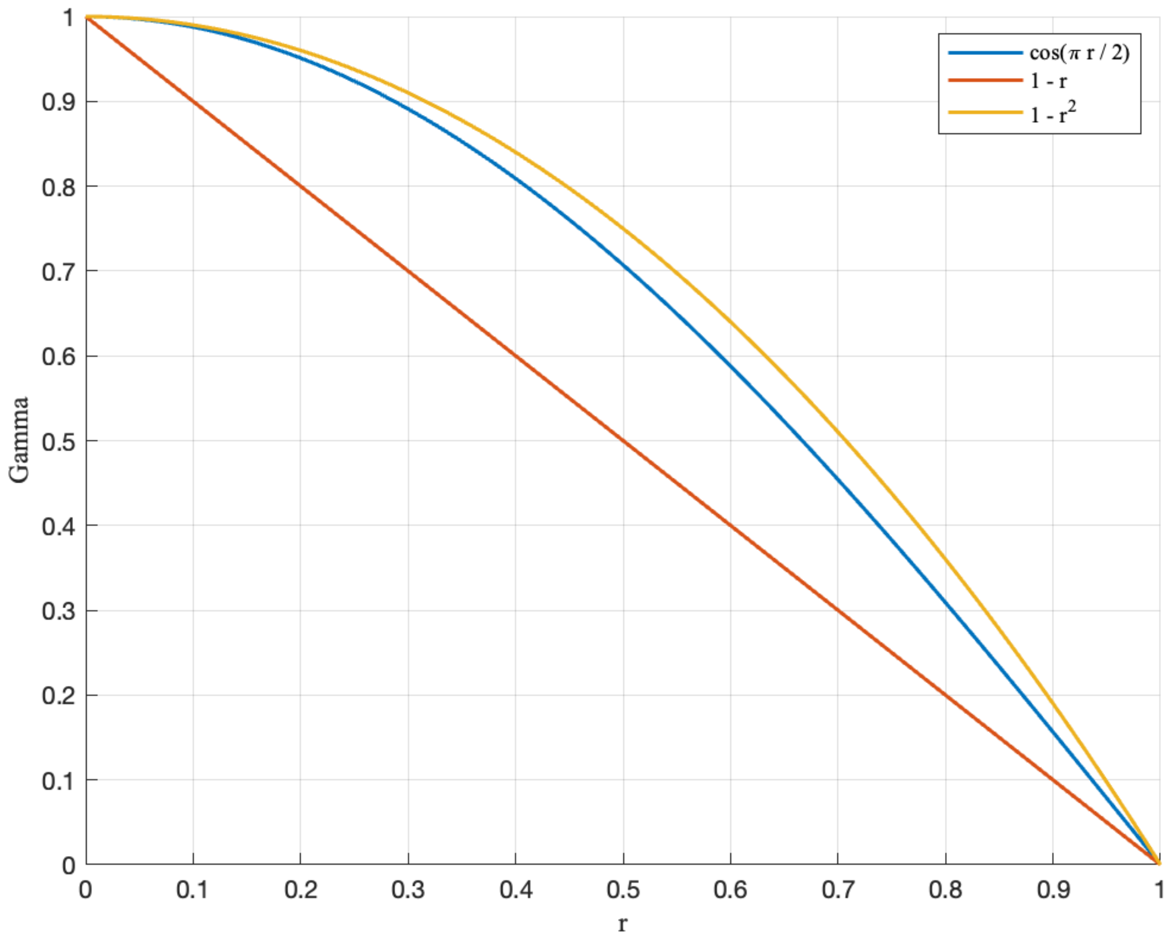
Scheduling Comparisons

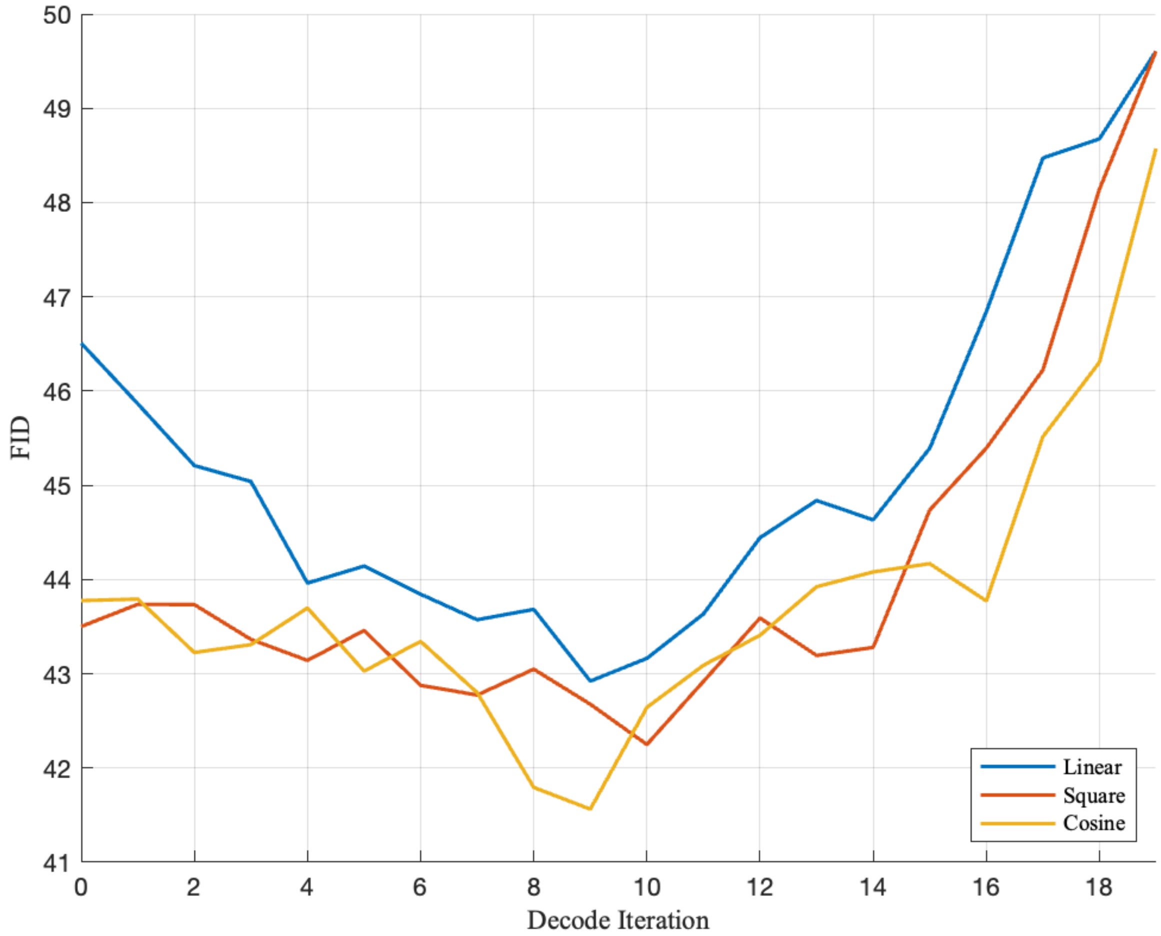
The comparisons are done under decoding for 20 iterations.

Cosine Scheduling

Linear Scheduling

Square Scheduling


cos(π r / 2)
1 - r
1 - r^2
Gamma
r
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1



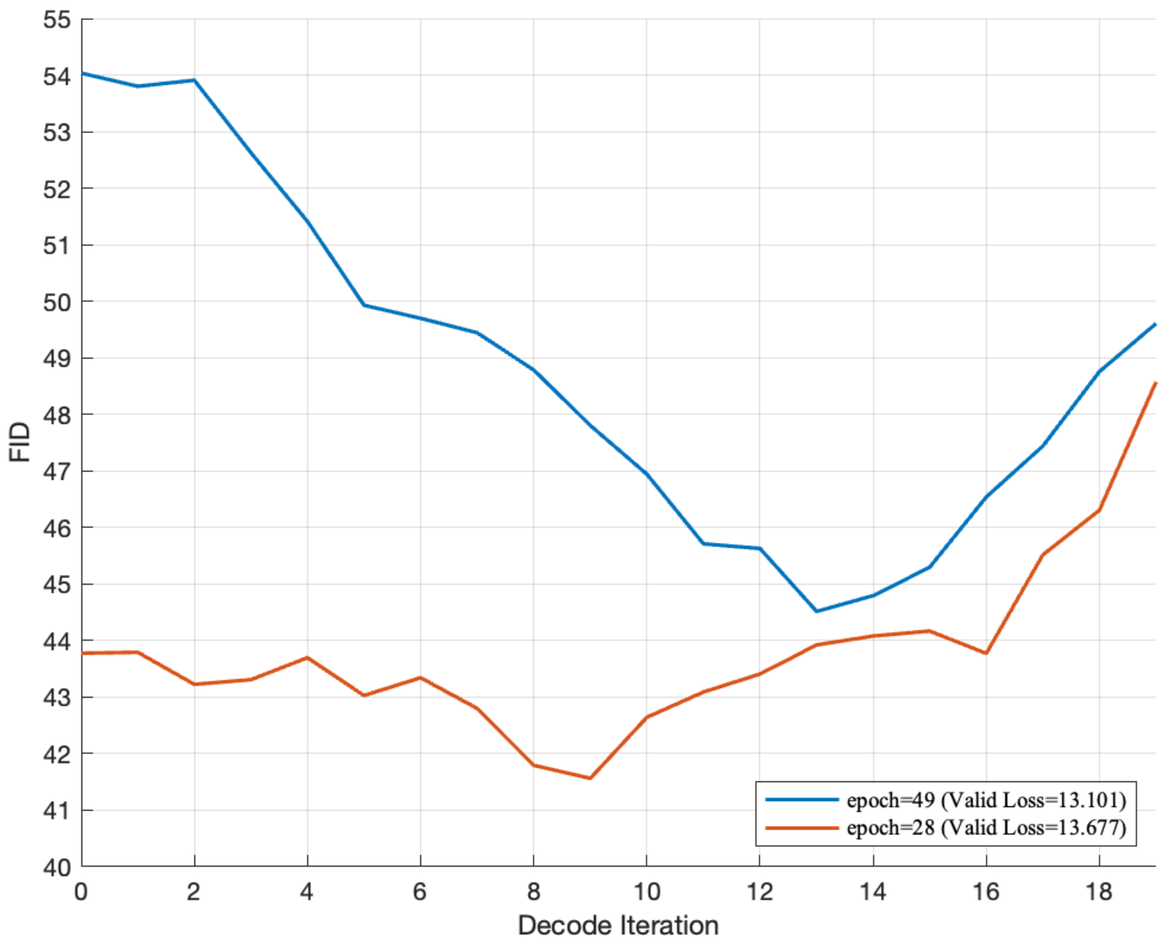
Scheduling Method	Best FID	Mean FID
Cosine	41.5636	43.7906
Square	42.2494	44.0826
Linear	42.9221	45.2226

From the figure, it is obvious that the cosine mask scheduling yields the best decoding results with the lowest Best FID and mean, followed by the square mask scheduling, the poorest is the linear mask scheduling.

Discussion

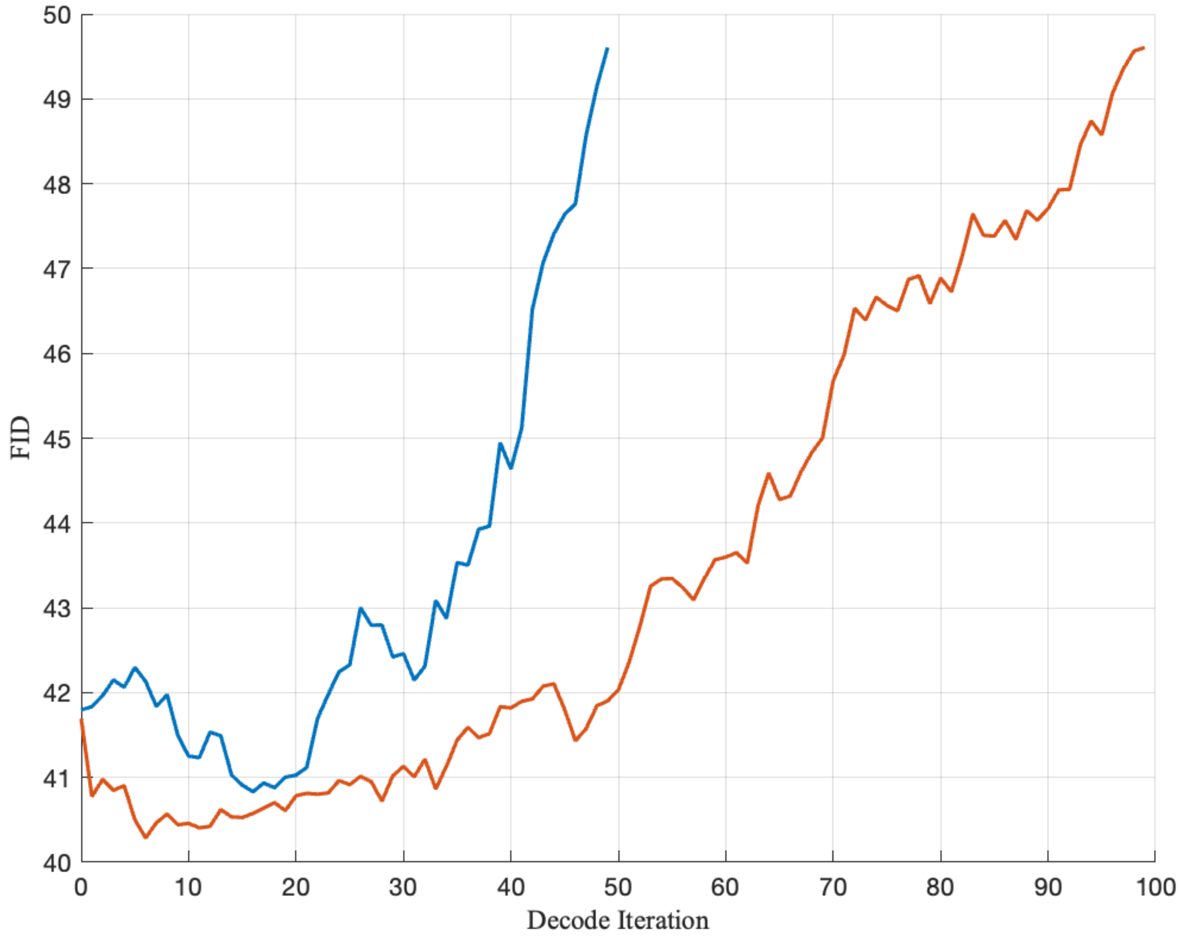
Smaller Loss Doesn't Indicate Smaller FID

The following figure is the FID curve of each decode iteration under the checkpoint with different loss. The checkpoint of epoch 49 had valid loss of value 13.101 while the checkpoint of epoch 28 had a valid loss of 13.677, yet in the figure, the checkpoint from epoch 28 had significantly better ability at predicting the masked pictures. Therefore, the valid loss doesn't always align with the performance. Such result can result from the difference in distribution of the valid dataset and test dataset.



Mask Scheduling Iterations Matter

The inpainting task decodes all the tokens every iteration, the scheduling function determines the number of tokens to be kept, hence, the longer the total iteration, the smaller the number of tokens are kept at each iteration, meaning that we only choose the "best" token at each iteration. That's why decoding for 100 iterations yields a significantly better result compared to decoding for 50 epochs.



Reference:

- MaskGIT-pytorch
- MaskGIT-PAT