Deep Learning Lab3 Report

陳以瑄 (ID: 313551047)

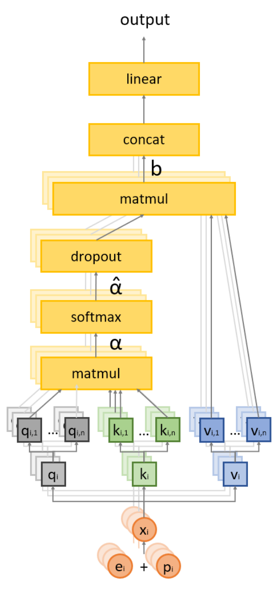
1 Introduction

In this assignment, I implemented the multi-head self-attention mechanism and the MVTM of MaskGit to generate images from masked inputs. A specail part of my approach is using the optimizer from minGPT for training. The best FID score I achieved is 31.5044.

2 Implementation Details

2.1 The details of model (Multi-Head Self-Attention)

I follow the multi-head attention design mentioned in Lecture 8 - Transformer (p. 33). First, the input x passes through the linear layers Q, K, and V. Then, it is split into multiple heads. For each head, the attention is computed as:
$Attention(Q, K, V) = \text{dropout}\left(\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)\right) V$
After computing attention for all heads, their outputs are concatenated to form the result:
$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \text{head}_2, \dots, \text{head}_h)W^o$
The main difference from the standard specification is that I apply the dropout layer after the softmax function to prevent overfitting.



The implementation of initialization:

```python
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.head_dim = dim//num_heads

        # Linear layer Wq, Wk, Wv (N*N)
        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)
        self.dropout = nn.Dropout(attn_drop)
```

The implementation of forward:

```
def forward(self, x):
    batch_size, num_image_tokens, _ = x.shape

    # Linear, get (batch_size, # tokens, dim)
    Q = self.q_linear(x)
    K = self.k_linear(x)
    V = self.v_linear(x)

    # Reshape for multi-head, get (batch_size, # tokens, # heads, head_dim)
    Q = Q.view(batch_size, num_image_tokens, self.num_heads, self.head_dim)
    K = K.view(batch_size, num_image_tokens, self.num_heads, self.head_dim)
    V = V.view(batch_size, num_image_tokens, self.num_heads, self.head_dim)

    # Reorder, so that compute independently for each head, get (batch_size
   , # heads, # tokens, head_dim)
    Q = Q.transpose(1, 2)
    K = K.transpose(1, 2)
    V = V.transpose(1, 2)

    # Attention = dropout(softmax(QK^T/(d_k)**0.5))V
    A = torch.matmul(Q, K.transpose(-2,-1))/(self.head_dim**0.5)
    A_hat = torch.nn.functional.softmax(A, dim =-1)
    A_hat = self.dropout(A_hat)
    B = torch.matmul(A_hat, V)

    # concat the output
    B = B.transpose(1, 2).reshape(batch_size, num_image_tokens, self.dim)
    output = self.out_linear(B)

    return output
```

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

2.2.1 function encode_to_z

The encode_to_z function applies the VQGAN encoder to the input and obtains the latent representation after vector quantization. Here, zq represents the closest codebook entry, while codebook_indices contains the corresponding indices in the codebook.

```
##TODO2 step1-1: input x fed to vqgan encoder to get the latent and zq
@torch.no_grad()
def encode_to_z(self, x):
    zq, codebook_indices, q_loss = self.vqgan.encode(x)
    return zq, codebook_indices
```

2.2.2 function gamma_func

I designed the gamma_func to map a ratio in the range $(0, 1]$ to $[0, 1)$ based on the plot provided in the Lab 3 specification (p.8).

- **Linear**: $\gamma = 1 - \text{input}$.
- **Cosine**: ranging from $\cos(0)$ to $\cos\left(\frac{\pi}{2}\right)$.
- **Squared**:$\gamma = 1 - \text{input}^2$.

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

2.2.3 TODO2 step1-3: forward of MaskGit

The function first obtains the VQ-encoded representation of the input using encode_to_z(x). Then, a random mask ratio is generated. Since masking too many positions would make it difficult for the model to learn, I set the constraint that no more than 50% of the positions are masked. Next, for each position in the input, a random number is generated. If the random number is less than the mask ratio, that position is considered to be masked. Since there are 1024 entries in the codebook, with indices ranging from 0 to 1023, masked positions are assigned the index 1024. Finally, the masked representation (masked_z) is fed into a bidirectional transformer model to predict the missing tokens.

```
##TODO2 step1-3:
def forward(self, x):
    # get ground truth, and reshape as (batch_size, num_img_tokens)
    _, z_indices= self.encode_to_z(x)
    z_indices = z_indices.view(-1, self.num_image_tokens)

    # set the mask ratio at most 50%
    mask_ratio = torch.rand(1)
    mask_ratio = (mask_ratio * 0.5).item()

    # randomize the mask and set those < mask_ratio as masked
    mask = torch.rand_like(z_indices, dtype = float)
    mask = mask < mask_ratio
    # since there are 0~1023 entries in codebook => the masked one = 1024
    masked_z = torch.where(mask, 1024, z_indices)

    #transformer predict the probability of tokens
    logits = self.transformer(masked_z)
    return logits, z_indices
```

2.2.4 TODO2 step1-4-1: train_one_epoch and eval_one_epoch

During the training stage, for each image, the model returns both the predicted logits and the ground truth z_indices. The goal is for these two to be as similar as possible, so I use cross-entropy loss as the loss function to measure the discrepancy between the predictions and the ground truth. The computed loss is then used to update the model's parameters.

```
def train_one_epoch(self, dataloader):
    self.model.train()
    total_loss = 0
    for imgs in tqdm(dataloader):
        x = imgs.to(self.args.device)
        self.optim.zero_grad()
        # get y_pred and gt y
        logits, z_indices = self.model(x)

        # reshape to (batch_size * num_tokens, token_size)
        logits = logits.view(-1, logits.size(-1))
        z_indices = F.one_hot(z_indices.view(-1), num_classes = logits.size
   (-1)).float()

        loss = F.cross_entropy(logits, z_indices)
        total_loss += loss.item()
        loss.backward()
        self.optim.step()

    total_loss /= len(dataloader)
    print(f"Training Loss: {total_loss}")
    return total_loss
```

The evaluation stage is similar to the training stage, with the main difference being that we only compute the loss without updating the model's weights.

```
@torch.no_grad()
def eval_one_epoch(self, dataloader):
    total_loss = 0
    for imgs in tqdm(dataloader):
        x = imgs.to(self.args.device)
        # get y_pred and gt y
        logits, z_indices = self.model(x)
        # reshape to (batch_size * num_tokens, token_size)
        logits = logits.view(-1, logits.size(-1))
        z_indices = F.one_hot(z_indices.view(-1), num_classes = logits.size
    (-1)).float()

        loss = F.cross_entropy(logits, z_indices)
        total_loss += loss.item()

    total_loss /= len(dataloader)
    print(f"Validation Loss: {total_loss}")
    return total_loss
```

2.2.5 TODO2 step1-4-2: configure_optimizers

The training strategy I used is based on the one applied in minGPT, which is a simpler yet powerful version of GPT. I believe the optimizer they use is a good choice for my model as well. The key idea behind this optimizer is to prevent overfitting by applying weight decay, but only to the weights of the linear layers. This helps to regularize the model while ensuring that other layers, such as biases, are not penalized.
Additionally, I apply a warmup step during training. Initially, the model's weights have unstable gradients (or loss), so starting with a smaller learning rate and then gradually increasing it before decreasing it can help prevent the model from being overly influenced by the early stages of training.

```
def configure_optimizers(self):
    decay = set()
    no_decay = set()
    # seperate 2 class: weight-decay or not
    whitelist = (torch.nn.Linear, )
    blacklist = (torch.nn.LayerNorm, torch.nn.Embedding)

    for mn, m in self.model.transformer.named_modules():
        for pn, p in m.named_parameters():
            fpn = '%s.%s' % (mn, pn) if mn else pn
            if pn.endswith('bias'):
                # all biases will not be decayed
                no_decay.add(fpn)
            elif pn.endswith('weight') and isinstance(m, whitelist):
                # weights of whitelist modules will be weight decayed
                decay.add(fpn)
            elif pn.endswith('weight') and isinstance(m, blacklist):
                # weights of blacklist modules will NOT be weight decayed
                no_decay.add(fpn)

    # validate that we considered every parameter
    param_dict = {pn: p for pn, p in self.model.transformer.
   named_parameters()}

    # create the pytorch optimizer object
    optim_groups = [
        {"params": [param_dict[pn] for pn in sorted(list(decay))], "
   weight_decay": self.args.weight_decay},
        {"params": [param_dict[pn] for pn in sorted(list(no_decay))], "
   weight_decay": 0.0},
    ]
    optimizer = torch.optim.AdamW(optim_groups, lr=self.args.learning_rate,
     betas=self.args.betas)

    scheduler = torch.optim.lr_scheduler.LambdaLR(optimizer, lambda steps:
   min((steps+1)/self.args.warmup_steps,1))

    return optimizer,scheduler
```

2.2.6 TODO2 step1-5: main of training

In this function, I record everything I need during training.

```
#TODO2 step1-5:
for epoch in range(args.start_from_epoch+1, args.epochs+1):
    print(f"Epoch: {epoch}, lr: {train_transformer.scheduler.get_last_lr()
    [0]}")
    train_loss = train_transformer.train_one_epoch(train_loader)
    valid_loss = train_transformer.eval_one_epoch(val_loader)

    writer.add_scalar("Loss/Train", train_loss, epoch)
    writer.add_scalar("Loss/Valid", valid_loss, epoch)
    if epoch % args.save_per_epoch == 0:
        print(f"Saving epoch {epoch} ...")
        torch.save(train_transformer.model.transformer.state_dict(), f"
   transformer_checkpoints/epoch_{epoch}.pt")
    train_transformer.scheduler.step()
```

2.3 The details of your inference for inpainting task

2.3.1 one iteration decoding

For the masked positions, the transformer predicts the probability distribution over the k codebook entries. The entry with the maximum probability is selected as the predicted index, and its probability is recorded as the confidence for that position. To help the model generalize better, we add Gumbel noise to the confidence. For the masked positions, we then sort them based on their confidence values from low to high. The n lowest-confidence positions, where n is correlated to the timestep, are masked again, as we assume these positions have not been learned well.

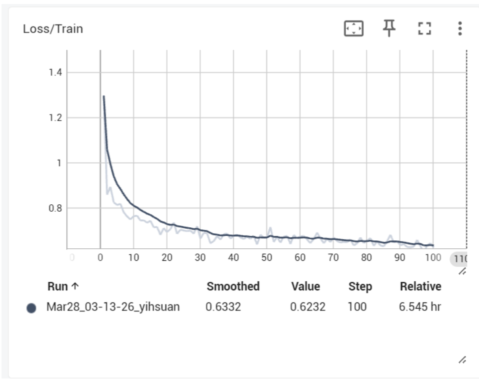
```
@torch.no_grad()
def inpainting(self, mask, z_indices, ratio, mask_num):
    masked_z = torch.where(mask, 1024, z_indices)
    logits = self.transformer(masked_z)

    #Apply softmax to convert logits into a probability distribution across
     the last dimension.
    logits = torch.nn.functional.softmax(logits, dim = -1).squeeze(0)

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

    #predicted probabilities add temperature annealing gumbel noise as
   confidence
    g = -torch.empty_like(z_indices_predict_prob).exponential_().log()
    temperature = self.choice_temperature * (1 - ratio)
    confidence = z_indices_predict_prob + temperature * g

    confidence_new = torch.where(mask, confidence, float('inf'))
    z_sort = torch.argsort(confidence_new, descending = False)
    n = int(ratio * mask_num)
    threshold = confidence_new[0, z_sort[0,n]]

    z_indices_predict = torch.where(mask, z_indices_predict, z_indices)
    mask_bc=confidence_new < threshold
    return z_indices_predict, mask_bc
```

2.3.2 total iteration decoding

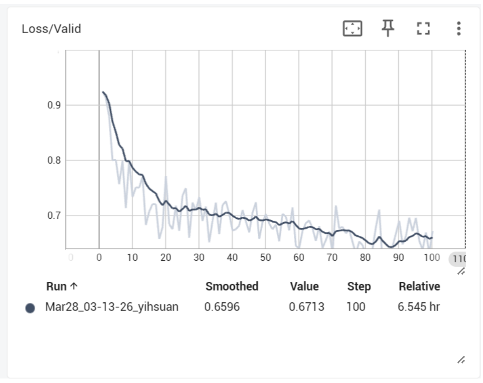
This function do T times iterative decoding and return the final predicted latent tokens.

```
def inpainting(self,image,mask_b,i):
    maska = torch.zeros(self.total_iter+1, 3, 16, 16)
    imga = torch.zeros(self.total_iter+1, 3, 64, 64)
    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

    # reshape the mask
    mask_i=mask_b.view(1, 16, 16)
    mask_image = torch.ones(3, 16, 16)
    indices = torch.nonzero(mask_i, as_tuple=False)
    mask_image[:, indices[:, 1], indices[:, 2]] = 0 #3,16,16
    maska[0]=mask_image

    self.model.eval()
    with torch.no_grad():
        #z_indices: masked tokens (b,16*16)
        _, z_indices = self.model.encode_to_z(image)
        mask_num = mask_b.sum()
        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
        for step in range(self.total_iter):
            if step == self.sweet_spot:
                break
            ratio = self.model.gamma(step/self.total_iter)

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

            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+1]=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
```

3 Discussion



Run ↑	Smoothed	Value	Step	Relative
● Mar28_03-13-26_yihsuan	0.6332	0.6232	100	6.545 hr



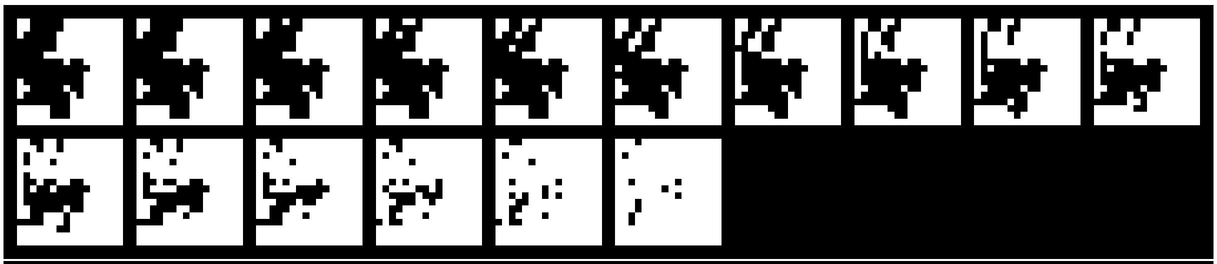
Run ↑	Smoothed	Value	Step	Relative
● Mar28_03-13-26_yihsuan	0.6596	0.6713	100	6.545 hr

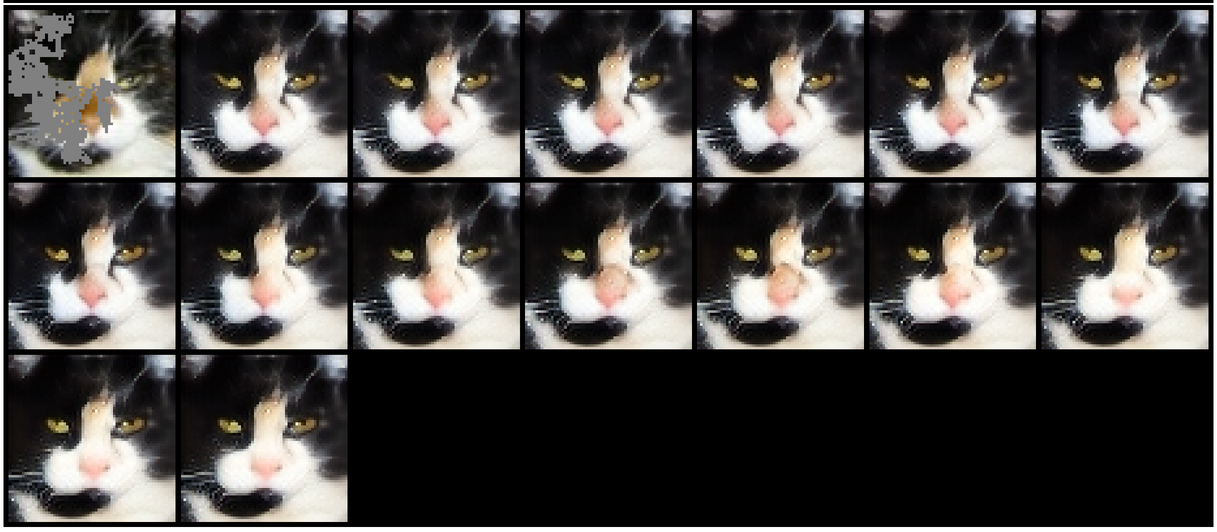
During the training process, I noticed that the trends of training loss and validation loss are quite similar. I think this is because I implemented several techniques to prevent overfitting, such as dropout layers and weight decay.
I also want to discuss that, although the masked positions in training are randomly selected, the test cases apply masks to specific regions. As seen in the results in the following sections, masked pixels near the boundaries of these regions tend to become confident more quickly, while those in the center require more iterations to converge. I think allowing the mask to be regional during training might help improve the model.

4 Experiment Score

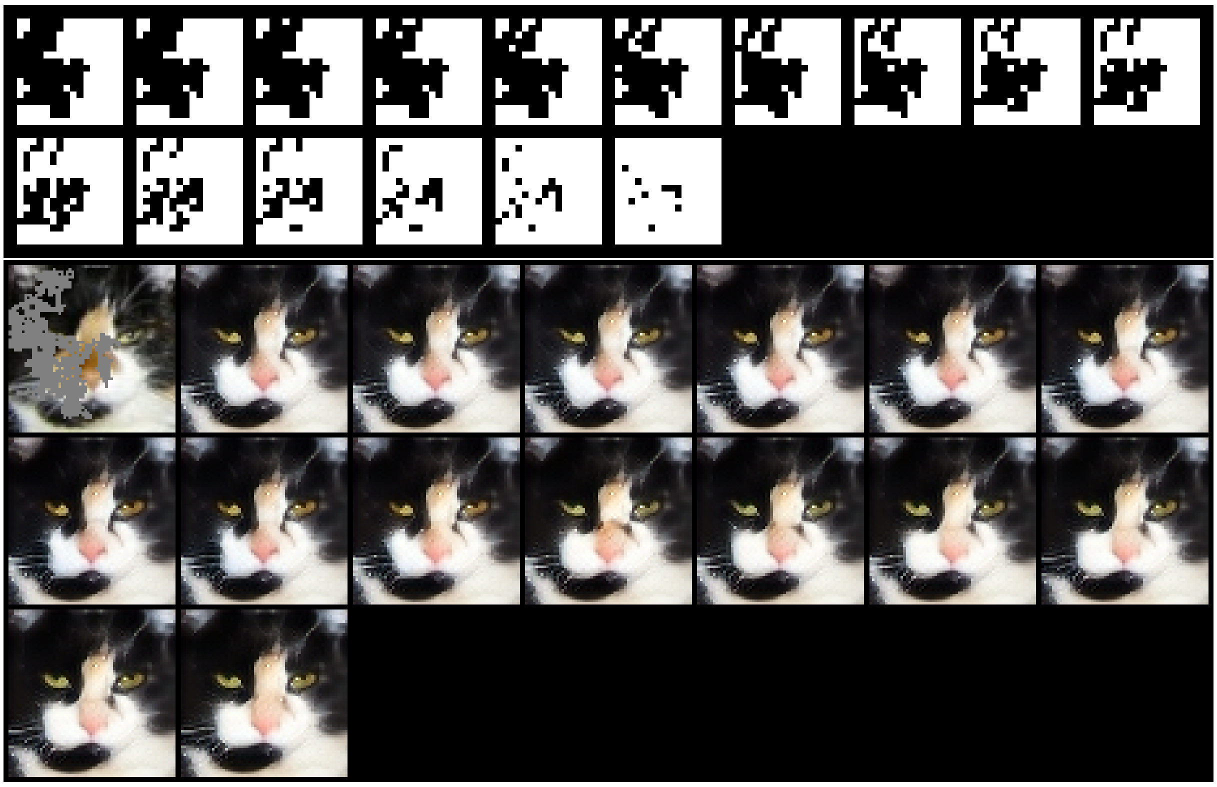
4.1 Show iterative decoding (T = 15)

1. cosine

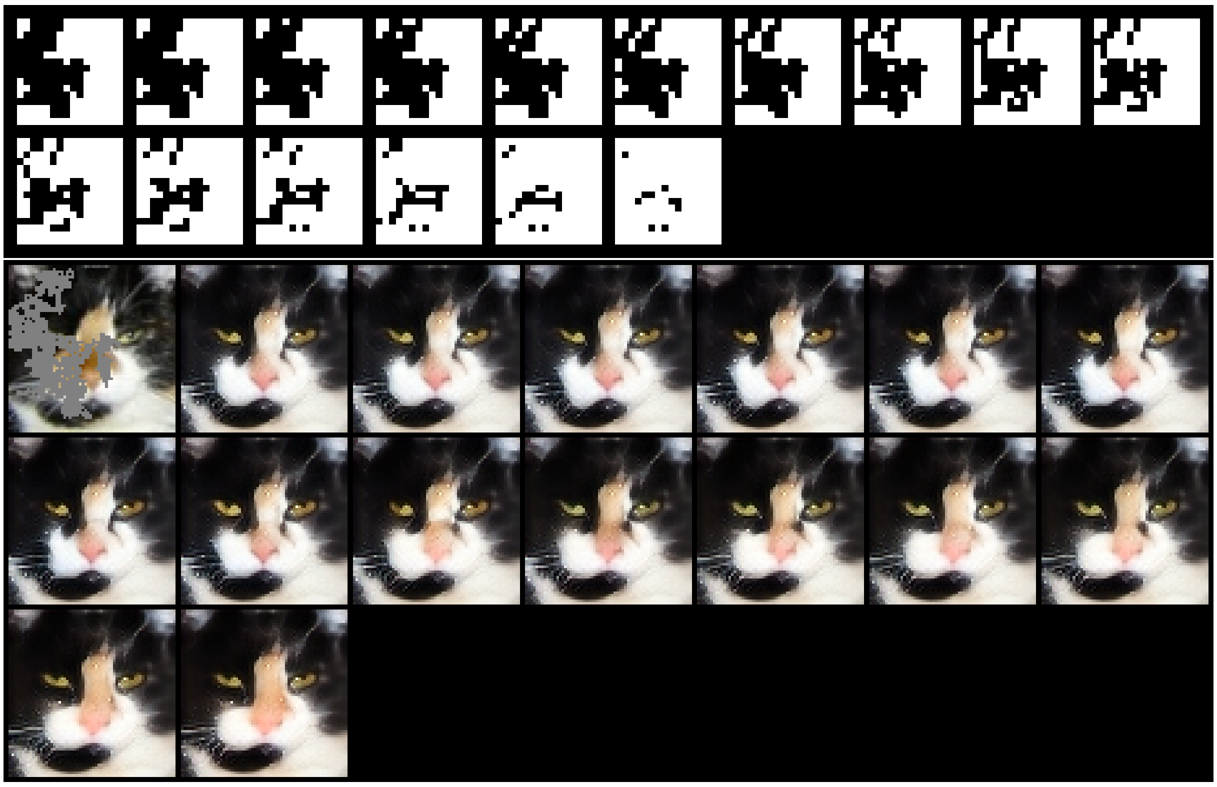




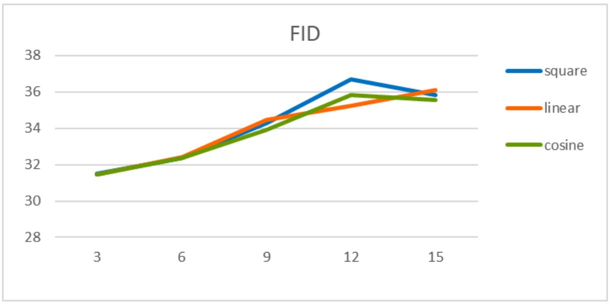
2. linear



3. square



4. FID score

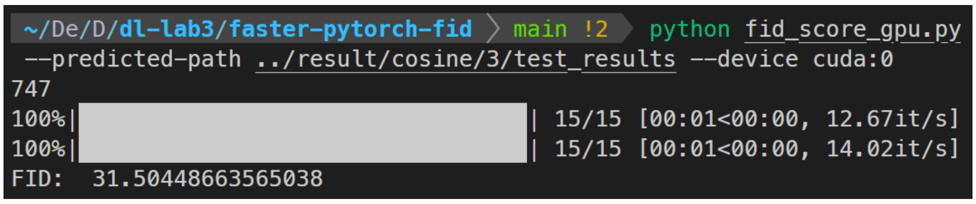


To my surprise, as the number of iterations t increases, the FID score also becomes higher, indicating poorer performance.

4.2 The Best FID Score

4.2.1 Screenshot

The parameters of best FID are: T = 15, t = 3, mask-func = cosine

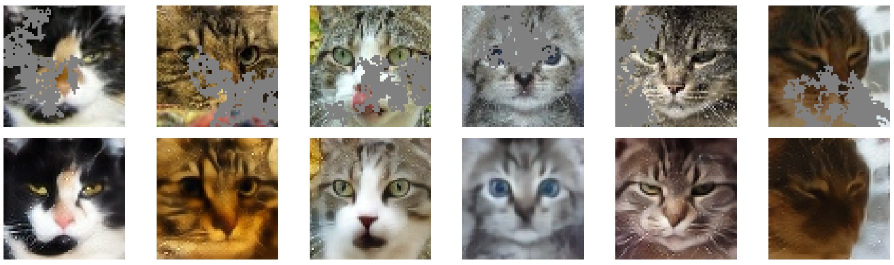


```
~/De/D/dl-lab3/faster-pytorch-fid  main !2  python fid_score_gpu.py
 --predicted-path ../result/cosine/3/test_results --device cuda:0
747
100%|                                                      | 15/15 [00:01<00:00, 12.67it/s]
100%|                                                      | 15/15 [00:01<00:00, 14.02it/s]
FID:  31.50448663565038
```

4.2.2 Masked Images v.s MaskGIT Inpainting Results

Masked

MaskGit result



4.2.3 The setting about training strategy, mask scheduling parameters

- epochs = 100
- learning-rate = 5e-4
- betas = (0.9, 0.95)
- weight-decay = 0.01
- warmup-steps = 50