Lab 5 Report

Deep Learning

Name: Kai-Jie Lin
Student ID 110652019
May 16, 2024

1 Introduction

In this lab, I implemented multi-head attention module and train the transformer model to predict latent token of images. After training transformer, I implemented iterative decoding for inpainting tasks. I compare the FID score with different settings of mask scheduling. Finally, I got best FID score of 27.68.

2 Implementation Details

2.1 Multi-Head Self-Attention

First create linear layer for query, key, and value. The dimensions of key, query, value are for all heads.

```
self.n_head = num_heads
self.dim = dim
# key, query, value dimension for all heads
self.key = nn.Linear(dim, dim)
self.query = nn.Linear(dim, dim)
self.value = nn.Linear(dim, dim)
# regularization
self.attn_drop = nn.Dropout(attn_drop)
# output projection
self.proj = nn.Linear(dim, dim)
```

Then input x into linear layers to get query, key and value. We need reshape the size of query, key, value to (batch size, token number, number of heads, embeded dimensions // number of heads). We can calculate the attention score following the formula. $Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$, where d_k is embeded dimensions // number of heads. Here we will do a dropout before multiply the attention score with value. Finally we can re-assemble all heads outputs and input into projection layer.

```
B, T, D = x.size()
k = self.key(x).view(B, T, self.n_head, D // self.n_head).transpose(1, 2)
q = self.query(x).view(B, T, self.n_head, D // self.n_head).transpose(1, 2)
v = self.value(x).view(B, T, self.n_head, D // self.n_head).transpose(1, 2)
att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1)))
att = nn.functional.softmax(att, dim=-1)
att = self.attn_drop(att)
y = att @ v

y = y.transpose(1, 2).contiguous().view(B, T, D) # re-assemble all head outputs side by side
return self.proj(y)
```

2.2 The details of stage2 training

Model traning forward:

First we input the image into the VQGAN encoder to get the latent tokens. Then we create a mask using bernoulli distribution with given ratio. Mask the latent tokens and input it into transformer. We can get the probability of each kinds of tokens. Finally, we create a ground truth logits with one hot encoding from the ground truth latent tokens. Return the logits and ground truth logits for further training process.

```python
def forward(self, x, ratio):
    _, z_indices=self.encode_to_z(x) #ground truth
    z_indices = z_indices.view(-1, self.num_image_tokens)
    # apply mask to the ground truth
    mask = torch.bernoulli(torch.ones_like(z_indices) * ratio)
    z_indices_input = torch.where(mask == 1, torch.tensor(self.mask_token_id).to(mask.device), z_indices)
    logits = self.transformer(z_indices_input)  #transformer predict the probability of tokens
    logits = logits[..., :self.mask_token_id]
    ground_truth = torch.zeros(
        z_indices.shape[0],
        z_indices.shape[1],
        self.mask_token_id).to(z_indices.device).scatter_(2, z_indices.unsqueeze(-1), 1)
    return logits, ground_truth
```

MVTM traning:

We use the logits and ground truth logits to calculate the cross entropy loss.

```python
def train_one_epoch(self, train_loader, epoch):
    self.model.train()
    losses = []

    for x in (pbar := tqdm(train_loader, ncols=120)):
        x = x.to(self.device)
        self.optim.zero_grad()
        ratio = np.random.rand()
        y_pred, y = self.model(x, ratio)
        loss = F.cross_entropy(y_pred, y)
        loss.backward()
        self.optim.step()
        losses.append(loss.detach().item())
        self.tqdm_bar(epoch, f'Train', pbar, loss.detach().cpu(), lr=self.scheduler.get_last_lr()[0])

    self.scheduler.step()
    return np.mean(losses)
```

Optimizer Configuration and Learning Rate Scheduler:

Following the trick from MinGPT, we are separating out all parameters of the model into two buckets: those that will experience weight decay for regularization and those that won't (biases, and layernorm/embedding weights). We are then returning the PyTorch optimizer object. Learning rate warm-up (in which the learning rate is gradually increased during the early stages of training) is usually used in transformer traning. The learning rate is increased linearly from 0 to R over first T_R time steps so that: $lr[t] = R\frac{t}{T_r}$.

```python
def configure_optimizers(self):
    # separate out all parameters to those that will and won't experience regularizing weight decay
    decay = set()
    no_decay = set()
    whitelist_weight_modules = (torch.nn.Linear, )
    blacklist_weight_modules = (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 # full param name

            if pn.endswith('bias'):
                # all biases will not be decayed
                no_decay.add(fpn)
            elif pn.endswith('weight') and isinstance(m, whitelist_weight_modules):
                # weights of whitelist modules will be weight decayed
                decay.add(fpn)
            elif pn.endswith('weight') and isinstance(m, blacklist_weight_modules):
                # weights of blacklist modules will NOT be weight decayed
                no_decay.add(fpn)

    # special case the position embedding parameter in the root GPT module as not decayed
    no_decay.add('pos_emb')

    # validate that we considered every parameter
    param_dict = {pn: p for pn, p in self.model.transformer.named_parameters()}
    inter_params = decay & no_decay
    union_params = decay | no_decay
    assert len(inter_params) == 0, "parameters %s made it into both decay/no_decay sets!" % (str(inter_params), )
    assert len(param_dict.keys() - union_params) == 0, "parameters %s were not separated into either decay/no_decay set!" \
                                                % (str(param_dict.keys() - union_params), )

    # create the pytorch optimizer object
    optim_groups = [
        {"params": [param_dict[pn] for pn in sorted(list(decay))], "weight_decay": 0.01},
        {"params": [param_dict[pn] for pn in sorted(list(no_decay))], "weight_decay": 0.0},
    ]
    optimizer = torch.optim.AdamW(optim_groups, lr=self.learning_rate, betas=(0.9, 0.95))
    scheduler = torch.optim.lr_scheduler.LambdaLR(optimizer,lambda steps: min((steps+1)/self.args.warmup_steps, 1))
    return optimizer, scheduler
```

2.3 Iterative Decoding

Mask Scheduling Functions:

Implemented mask scheduling functions $\gamma(\frac{t}{T})$ for iterative decoding. Here are linear, cosine and square functions. When training, we just sample the ratio from uniform distribution.

```python
if mode == "linear":
    def f(ratio):
        return 1 - ratio
    return f
elif mode == "cosine":
    def f(ratio):
        return math.cos(math.pi * ratio / 2)
    return f
elif mode == "square":
    def f(ratio):
        return 1 - ratio ** 2
    return f
```

Iterative Decoding:

In one iterative decoding, we first input the image into the VQGAN encoder to get the latent tokens. Then we mask the token value with given latent mask. We input the masked tokens into the transformer to get the probability of each kinds of tokens. We can get the most likely token and the probability of the token. Calculate confidence using predicted probabilities add temperature annealing gumbel noise. $Confidence = p_z + temperature \cdot (-\ln(-\ln(p)))$, where p_z is the predicted probability of the token and p is sampled from uniform distribution. We mask the tokens with n lowest confidence, other tokens are either predicted tokens or ground truth tokens. $n = \lceil \gamma(\frac{t}{T})N \rceil$, where N is number of masked tokens of input image. Note that if ratio is less than 10^{-8}, then I set it to zero to ensure last output mask is unmask. Finally, we return predicted latent tokens and the mask of next iteration.

```python
def inpainting(self, x , ratio, mask_b):
    _, z_indices=self.encode_to_z(x)
    z_indices_input = torch.where(mask_b == 1, torch.tensor(self.mask_token_id).to(mask_b.device), z_indices)
    logits = self.transformer(z_indices_input)
    #Apply softmax to convert logits into a probability distribution across the last dimension.
    logits = torch.nn.functional.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.randn(1, device=z_indices_predict_prob.device) # gumbel noise
    g = -torch.log(-torch.log(torch.rand(1, device=z_indices_predict_prob.device))) # gumbel noise
    temperature = self.choice_temperature * (1 - ratio)
    confidence = z_indices_predict_prob + temperature * g

    #hint: If mask is False, the probability should be set to infinity, so that the tokens are not affected by the transformer's prediction
    #sort the confidence for the rank
    confidence = torch.where(mask_b == 0, torch.tensor(float('inf')).to(mask_b.device), confidence)
    ratio = 0 if ratio < 1e-8 else ratio
    n = math.ceil(mask_b.sum() * ratio)
    _, idx_to_mask = torch.topk(confidence, n, largest=False)
    #define how much the iteration remain predicted tokens by mask scheduling
    #At the end of the decoding process, add back the original token values that were not masked to the predicted tokens
    mask_bc=torch.zeros_like(mask_b).scatter_(1, idx_to_mask, 1)
    torch.bitwise_and(mask_bc, mask_b, out=mask_bc)
    return z_indices_predict, mask_bc
```

In inference, we do the iterative decoding for T times and return the final predicted latent tokens. Note the input image and mask should be replaced by predicted ones every time.

```python
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(image, ratio, mask_bc) #mask_bc: mask in latent domain

    #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
    image = decoded_img
    imga[step+1]=dec_img_ori #get decoded image
```

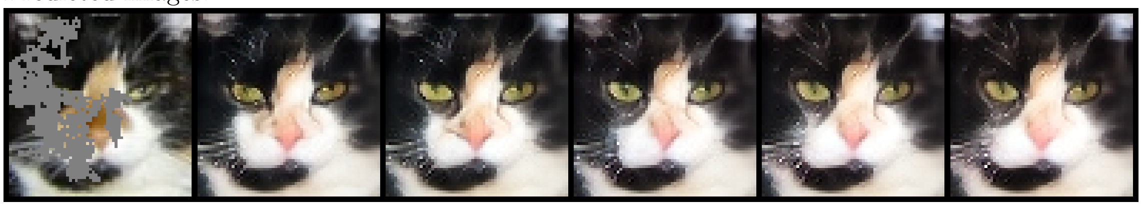
3 Experimental Results

3.1 The best testing fid

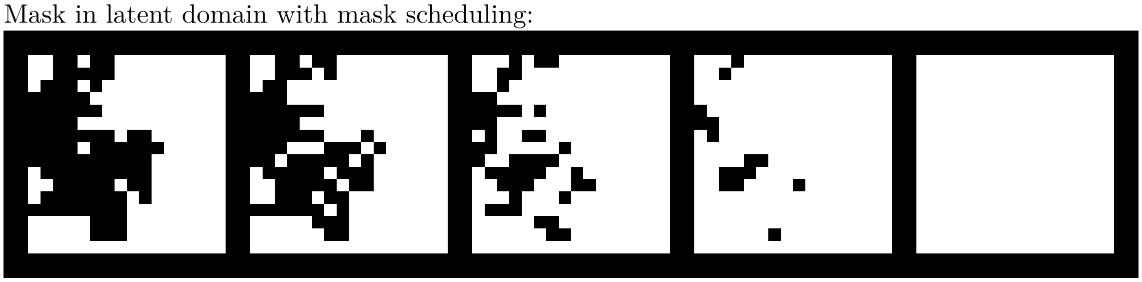
The best testing fid is 27.68.

```
(seq) hungyh@cpu2:~/kjl/dlp/Lab5/faster-pytorch-fid$ python fid_score_gpu.py --predicted-path /home/hungyh/kjl/dlp/Lab5/Result/test_results
100%|                                                                  | 15/15 [00:01<00:00, 13.30it/s]
100%|                                                                  | 15/15 [00:00<00:00, 19.12it/s]
27.681387706732664
```

Predicted images:



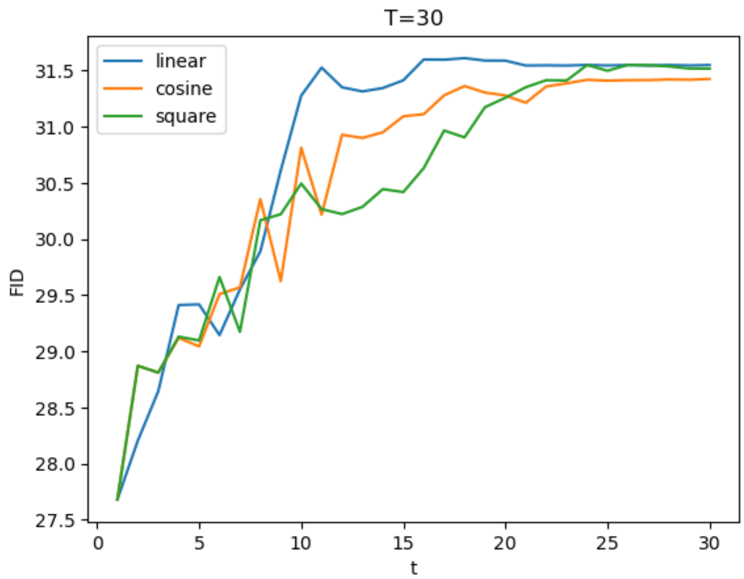
Note: FID score is calculated using second image above.
Mask in latent domain with mask scheduling:



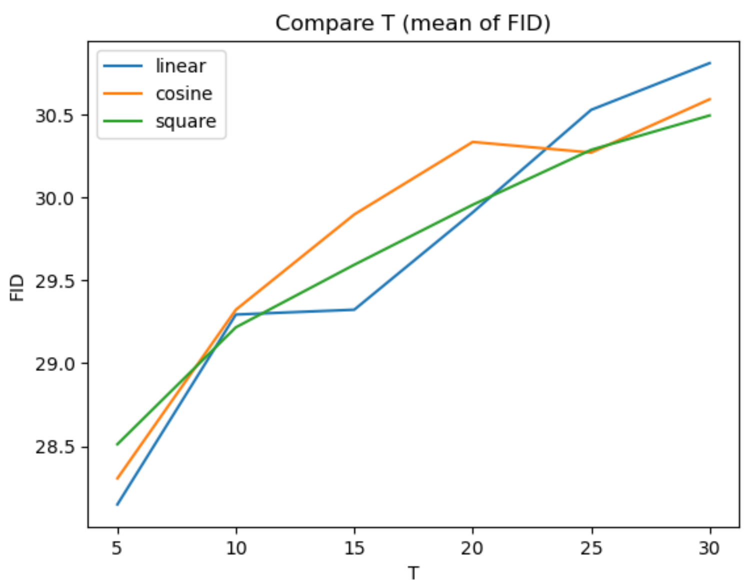
Traning Setting and Mask Scheduling:
I trained transformer with batch size = 64, learning rate = 0.0005, warmup steps = 50, number of epochs = 100, dropout rate = 0.1. In mask scheduling, I set total iteration to 5, sweet spot to 1 and gamma function is cosine.

3.2 Comparison figures with different mask scheduling parameters setting

We first compare the FID score with different gamma functions and different sweet spot. T is total iteration and t is sweet spot.



We can see that the FID score increased with the increase of t. The performance would be poor if doing iterative decoding too many times. In early iterations, the linear function gives the best performance. The square function gives the best performance in middle iterations. The cosine function gives the best performance in late iterations.
We then compare the FID score with different gamma functions and different total iteration.

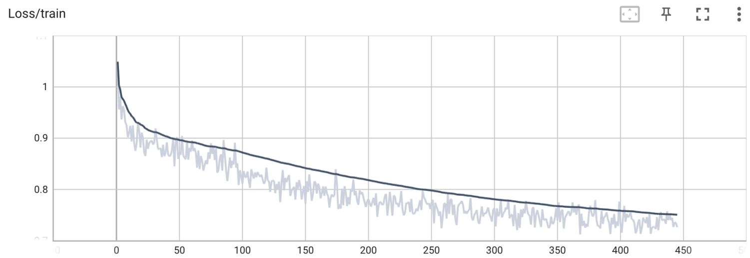


We can conclude that the combination of smaller T and linear function would give best performance.

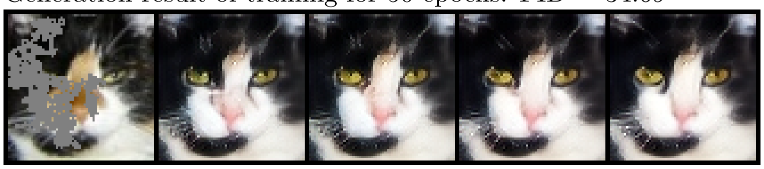
4 Discussion

Transformer Traning and Generation Performance:

Training Loss:



Generation result of training for 50 epochs: FID = 34.69



Generation result of training for 100 epochs: FID = 28.65

Generation result of training for 150 epochs: FID = 30.89

Generation result of training for 200 epochs: FID = 366.64

The best performance comes from training for 100 epochs. The performance would be poor if training too many epochs. It seems like transformer would overfit to training data when training too much epochs.