Deep Learning HW3

110550085 房天越

1. Introduction

In this project, we use MaskGIT to do the training for a bidirectional transformer. There are three steps that we need to finish. First, we implement the logic for the Multihead Attention Layer. Second, we implement the training and evaluation strategy for the model. Third, we implement the decoding part for inpainting. Finally, we use FID to determine the impact of using different decoding ways.

2. Implementation Details

A. Multi Head Attention

First, we create linear layers for query, key, and value.

The dimensions of them are for all heads.

```
self.dim = dim
self.num_heads = num_heads
self.head_dim = dim // num_heads

self.q_proj = nn.Linear(dim, dim)
self.k_proj = nn.Linear(dim, dim)
self.v_proj = nn.Linear(dim, dim)

self.attn_drop = nn.Dropout(attn_drop)
self.out_proj = nn.Linear(dim, dim)
```

Second, we put x into the layers to get q, k, v. After reshaping, we calculate the attention score using the formula given on the spec.

We also need to do dropout before multiplying the attention score with value.

Finally, we merge the outputs of all heads and input into projection layer.

```
B, N, C = x.shape # C == dim
q = self.q_proj(x).view(B, N, self.num_heads, self.head_dim).transpose(1, 2)
k = self.k_proj(x).view(B, N, self.num_heads, self.head_dim).transpose(1, 2)
v = self.v_proj(x).view(B, N, self.num_heads, self.head_dim).transpose(1, 2)

attn_score = torch.matmul(q, k.transpose(-2, -1)) / math.sqrt(self.head_dim)
attn_probs = torch.nn.functional.softmax(attn_score, dim=-1)
attn_probs = self.attn_drop(attn_probs)

y = torch.matmul(attn_probs, v)
y = y.transpose(1, 2).contiguous().view(B, N, C)

output = self.out_proj(y)
return output
```

B. Stage 2 training

1. Encode to z

In this function, token is generated with the image through VQGAN.

```python
def encode_to_z(self, x):

    z_map, z_indices, loss = self.vqgan.encode(x)
    z_indices = z_indices.view(z_map.shape[0], -1)
    return z_map, z_indices
```

2. MVTM

   In this part, first we encode the input data x to get a z_indices, which is the ground truth. Then, we construct the mask token, and use Bernoulli to sample them and put into the transformer. The new indices are messed up with ground truth token and mask tokens. And the logits are predicted with the new indices using the transformer.

```python
def forward(self, x):

    # z_indices=ground truth
    # logits=transformer predict the probability of tokens

    _, z_indices = self.encode_to_z(x)
    mask_tk = torch.ones(z_indices.shape, device = z_indices.device).long()*self.mask_token_id
    mask = torch.bernoulli(0.5*torch.ones(z_indices.shape, device = z_indices.device)).long()

    new_ind = mask*mask_tk + (1-mask)*z_indices
    logits = self.transformer(new_ind)

    return logits, z_indices
```

3. Training Logic

   In the training part, we use logits and ground truth logits to calculate the loss, the loss function we used is Cross Entropy.

```python
def train_one_epoch(self, train_loader, epoch, args):
    #pass
    train_bar = tqdm(enumerate(train_loader))
    losses = []

    self.model.train()
    for i, data in train_bar:
        data = data.to(args.device)
        logits, z_indices = self.model(data)

        loss = F.cross_entropy(logits.view(-1, logits.shape[-1]), z_indices.view(-1))
        loss.backward()
        losses.append(loss.item())
        if i % args.accum_grad == 0:
            self.optim.step()
            self.optim.zero_grad()
        train_bar.set_description_str(f"Epoch {epoch} | Iteration {i+1} / {len(train_loader)} | Loss: {np.mean(losses):.4f}")

    train_loss = np.mean(losses)
    self.writer.add_scalar("Loss/train", train_loss, epoch)
    return train_loss
```

C. Inference for Inpainting Task

I. Gamma Functions

Here are the functions we use for mask scheduling.

```python
if mode == "linear":
    #raise Exception('TODO2 step1-2!')
    return lambda r : 1 - r
elif mode == "cosine":
    #raise Exception('TODO2 step1-2!')
    return lambda r : np.cos(r * np.pi / 2)
elif mode == "square":
    #raise Exception('TODO2 step1-2!')
    return lambda r : 1 - r**2
else:
    raise NotImplementedError
```

II. Iterative Decoding

In this part, first, we replace masked positions in z_indices with mask_token_id, then feed the masked indices to the transformer. Then, we compute logits with it, and apply softmax to get probability distribution.

After that, we generate z_indices_predict to ensure no mask_token_id remains, and keep original tokens for unmasked positions.

Then, we compute confidence score by determining masking ratio and the number of tokens to unmask, and apply temperature annealing and Gumbel noise for diversity.

Finally, we calculate a confidence threshold and mask out tokens with confidence below the threshold.

```python
@torch.no_grad()
def inpainting(self, z_indices, mask, mask_num, ratio, mask_func):


    masked_z_indices  = mask * self.mask_token_id + (~mask) * z_indices
    logits = self.transformer(masked_z_indices)

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

    #FIND MAX probability for each token value
    while True:
        z_indices_predict = torch.distributions.categorical.Categorical(logits=logits).sample()
        if torch.all(z_indices_predict != self.mask_token_id):
            break

    z_indices_predict = mask * z_indices_predict + (~mask) * z_indices
    z_indices_predict_prob = prob_dist.gather(-1, z_indices_predict.unsqueeze(-1)).squeeze(-1)
    z_indices_predict_prob = torch.where(mask, z_indices_predict_prob, torch.zeros_like(z_indices_predict_prob) + torch.inf)

    mask_r = self.gamma_func(mask_func)(ratio)
    mask_l = torch.floor(mask_r * mask_num).long()

    #predicted probabilities add temperature annealing gumbel noise as confidence
    eps = 1e-6
    g = -torch.log(-torch.log(torch.rand_like(z_indices_predict_prob).clamp(eps, 1 - eps)))

    temperature = self.choice_temperature * (1 - mask_r)
    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
    sort_conf = torch.sort(confidence, dim=-1)[0]
    #define how much the iteration remain predicted tokens by mask scheduling
    ##At the end of the decoding process, add back the original(non-masked) token values

    cut_off = sort_conf[:, mask_l].unsqueeze(-1)
    mask_bc=(confidence < cut_off )
```

3. Experiment Score

Part 1. Prove your code implementation is correct

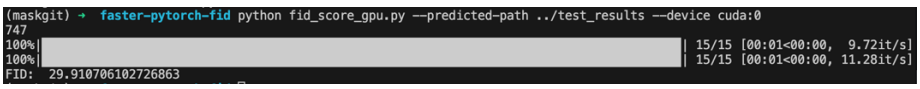
Here are the iterative decoding results of cosine, linear, and square, respectively. We can see for cosine and square, the unmasked parts change slowly at first, but fast at last, while for linear, the unmasked parts change equally.

And here are the predicted images for test_0:

Part 2. Best FID Score

This is the final FID score I got, using Gamma function “square”.

The performance is square > cosine > linear. However, they both did well and the FID scores were all way lower than 40.



Here are some hyperparameters I used:

Learning Rate: 1e-4.

Epochs of training: 100, got the best model for valid at epoch = 91.

Batch-size: 10.

Sweet-spot: 10.

Total-iter: 10.