DLP Lab7 Report

▼ 資科工碩　穆冠蓁　310551178

Report

Introduction

We try to implement a conditional Denoising Diffusion Probabilistic Models(DDPM) to generate images based on multi-lablel condition.There is totally 24 labels in this task.

Implementation details

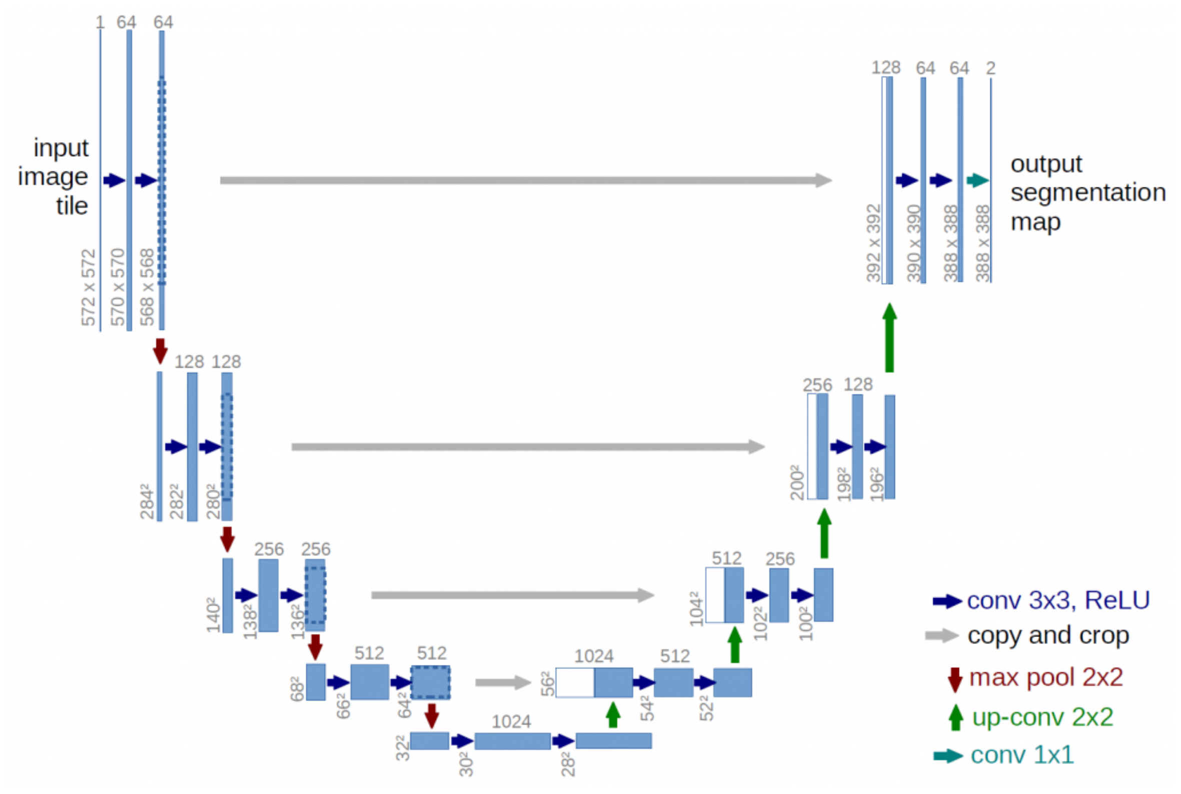
- Describe how you implement your model, including your choice of DDPM, UNet architectures, noise schedule, and loss functions.

  1. Type of DDPM

     I implement my DDPM on pixel space.Which include noise scheduler ,sampling timesteps, and also sampling images.

     ```
     class Diffusion:
         def __init__(self, noise_steps=1000, beta_start=1e-4, beta_end=0.02, img_size=256, device="cuda"):
             self.noise_steps = noise_steps
             self.beta_start = beta_start
             self.beta_end = beta_end

             self.beta = self.prepare_noise_schedule().to(device)
             self.alpha = 1. - self.beta
             self.alpha_hat = torch.cumprod(self.alpha, dim=0)

             self.img_size = img_size
             self.device = device

         def prepare_noise_schedule(self):
             return torch.linspace(self.beta_start, self.beta_end, self.noise_steps)

         def noise_images(self, x, t):
             sqrt_alpha_hat = torch.sqrt(self.alpha_hat[t])[:, None, None, None]
             sqrt_one_minus_alpha_hat = torch.sqrt(1 - self.alpha_hat[t])[:, None, None, None]
             Ɛ = torch.randn_like(x)
             return sqrt_alpha_hat * x + sqrt_one_minus_alpha_hat * Ɛ, Ɛ

         def sample_timesteps(self, n):
             return torch.randint(low=1, high=self.noise_steps, size=(n,))

         def sample(self, model, n, labels, cfg_scale=3):
             logging.info(f"Sampling {n} new images....")
             model.eval()
             with torch.no_grad():
                 x = torch.randn((n, 3, self.img_size, self.img_size)).to(self.device)
                 for i in tqdm(reversed(range(1, self.noise_steps)), position=0):
                     t = (torch.ones(n) * i).long().to(self.device)
                     predicted_noise = model(x, t, labels)
                     if cfg_scale > 0:
                         uncond_predicted_noise = model(x, t, None)
                         predicted_noise = torch.lerp(uncond_predicted_noise, predicted_noise, cfg_scale)
                     alpha = self.alpha[t][:, None, None, None]
                     alpha_hat = self.alpha_hat[t][:, None, None, None]
                     beta = self.beta[t][:, None, None, None]
                     if i > 1:
                         noise = torch.randn_like(x)
                     else:
                         noise = torch.zeros_like(x)
                     x = 1 / torch.sqrt(alpha) * (x - ((1 - alpha) / (torch.sqrt(1 - alpha_hat))) * predicted_noise) + torch.sqrt(beta)
             model.train()
             x = (x.clamp(-1, 1) + 1) / 2
             x = (x * 255).type(torch.uint8)
             return x
     ```

  2. Unet architecture



Our Unet is design in 3 downsampling blocks , 3 bottlenecks and 3 upsampling blocks.

In donwsampling part, included Down sampling function and followed Selfattention methods.

```
def __init__(self, c_in=3, c_out=3, time_dim=256, device="cuda"):
    super().__init__()
    self.device = device
    self.time_dim = time_dim
    self.inc = DoubleConv(c_in, 64)
    self.down1 = Down(64, 128)
    self.sa1 = SelfAttention(128, 32)
    self.down2 = Down(128, 256)
    self.sa2 = SelfAttention(256, 16)
    self.down3 = Down(256, 256)
    self.sa3 = SelfAttention(256, 8)

    self.bot1 = DoubleConv(256, 512)
    self.bot2 = DoubleConv(512, 512)
    self.bot3 = DoubleConv(512, 256)

    self.up1 = Up(512, 128)
    self.sa4 = SelfAttention(128, 16)
    self.up2 = Up(256, 64)
    self.sa5 = SelfAttention(64, 32)
    self.up3 = Up(128, 64)
    self.sa6 = SelfAttention(64, 64)
    self.outc = nn.Conv2d(64, c_out, kernel_size=1)
```

Our downsampling and upsampling blocks include one upsampling(downsampling)and one convolution layer and at last on embedding layers.

```
class Down(nn.Module):
    def __init__(self, in_channels, out_channels, emb_dim=256):
        super().__init__()
        self.maxpool_conv = nn.Sequential(
            nn.MaxPool2d(2),
            DoubleConv(in_channels, in_channels, residual=True),
            DoubleConv(in_channels, out_channels),
        )

        self.emb_layer = nn.Sequential(
            nn.SiLU(),
            nn.Linear(
                emb_dim,
                out_channels
            ),
        )

    def forward(self, x, t):
        x = self.maxpool_conv(x)
        emb = self.emb_layer(t)[:, :, None, None].repeat(1, 1, x.shape[-2], x.shape[-1])
        return x + emb


class Up(nn.Module):
    def __init__(self, in_channels, out_channels, emb_dim=256):
        super().__init__()

        self.up = nn.Upsample(scale_factor=2, mode="bilinear", align_corners=True)
        self.conv = nn.Sequential(
            DoubleConv(in_channels, in_channels, residual=True),
            DoubleConv(in_channels, out_channels, in_channels // 2),
        )

        self.emb_layer = nn.Sequential(
            nn.SiLU(),
            nn.Linear(
                emb_dim,
                out_channels
            ),
        )

    def forward(self, x, skip_x, t):
        x = self.up(x)
        x = torch.cat([skip_x, x], dim=1)
        x = self.conv(x)
        emb = self.emb_layer(t)[:, :, None, None].repeat(1, 1, x.shape[-2], x.shape[-1])
        return x + emb
```

Beside we use a special label embeddings label_emb to embed our one hot vector condition.

```
self.label_emb = nn.Sequential(
                nn.Linear(num_classes, 128),
                nn.ReLU(),
                nn.Linear(128, 128),
                nn.ReLU(),
                nn.Linear(128, time_dim)
            )
```

3. noise schedule

   I chose Simplified noise prediction method here.

```
def noise_images(self, x, t):
    sqrt_alpha_hat = torch.sqrt(self.alpha_hat[t])[:, None, None, None]
    sqrt_one_minus_alpha_hat = torch.sqrt(1 - self.alpha_hat[t])[:, None, None, None]
    Ɛ = torch.randn_like(x)
    return sqrt_alpha_hat * x + sqrt_one_minus_alpha_hat * Ɛ, Ɛ
```

   So we add noise to the original image, the noise image is represent as $\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon$

4. loss functions

   The loss function can write as

$$L_{t-1} = \mathbb{E}_{x_0,\epsilon}[\|\epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\epsilon)\|_2^2]$$

5. Dataloader

   I have written a dataset to read the label and image out which is called `iclevr_dataset.`

   It include `__len__`, `get_seq`, `label_to_onehot`, `get_label`, `__getitem__`.

   The `__init__` define the transformation(Resize ,Randomcrop, ToTensor, Normalize), open training file name and corresponding label in `train.json`, and label and corresponding number in `objects.json`.

```
def __init__(self, args):
    self.args = args
    default_transforms = torchvision.transforms.Compose([
        torchvision.transforms.Resize(80),  # args.image_size + 1/4 *args.image_size
        torchvision.transforms.RandomResizedCrop(self.args.image_size, scale=(0.8, 1.0)),
        torchvision.transforms.ToTensor(),
        torchvision.transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])
    self.root = '{}/iclevr/'.format(args.dataset_path)
    self.transform = default_transforms
    self.train_dict = json.load(open(args.dataset_path+'/train.json'))
    self.object_dict = json.load(open(args.dataset_path+'/objects.json'))
    self.dirs = list(self.train_dict.keys())
```

   `get_seq` read all the file store in `dirs` and do transformation.

```
def get_seq(self, idx):
    path = os.path.join(self.args.dataset_path, 'iclevr', self.dirs[idx])
    img = Image.open(path).convert('RGB')
    img = self.transform(img)
    return img
```

   Read all the label in training data and turn the label into one-hot vector.

```
# Turn label into one hot vector
def label_to_onehot(self, label, label_dict):
    onehot = np.zeros(24, dtype=np.float32)
    for l in label:
        onehot[label_dict[l]] = 1
    return onehot

def get_label(self, idx):
    label = self.train_dict[self.dirs[idx]]
    label = self.label_to_onehot(label, self.object_dict)
    return label
```

   Return all the image and label.

```
def __getitem__(self, idx):
    seq = self.get_seq(idx)
    label = self.get_label(idx)
    return seq, label
```

- Specify the hyperparameters (learning rate, epochs, etc.)

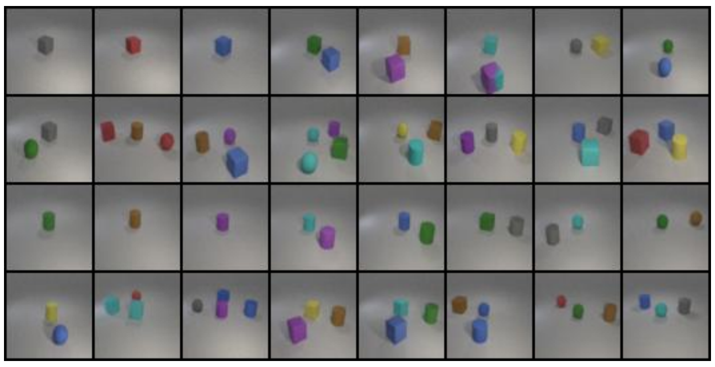
epochs	500
batch_size	14
image_size	64
learning rate	3e-4

Result and discussion

- Show your results based on the testing data

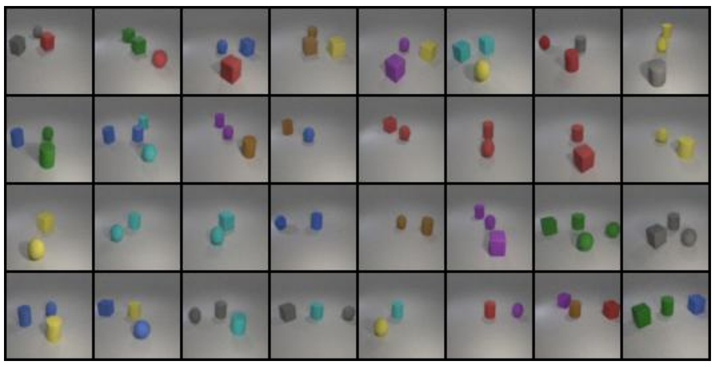
  Train in one layer of `label_emb` and 500 epochs.

  1. testing:



test: 0.8055555555555556

  2. new_testing :



new_test:0.8095238095238095

- Discuss the results of different model architectures. (20%) For example, what is the effect with or without some specific embedding methods, or what kind of prediction type is more effective in this case.

  In this section, our settings has change to train 400 epochs only.

  We implement different `label_emb` to compare the difference

Layer #	accuracy(test)	accuracy(new test)
2	**0.7916666666666666**	**0.7976190476190477**
3	0.7083333333333334	0.75
4	0.6527777777777778	0.7380952380952381
5	0.7361111111111112	0.7142857142857143

  From above result, we can see that if we increase the layer of label_embeddings can not significantly increase our accuracy, or maybe will decrease it. In my opinion, I think I will try to increase the training epoch next time.