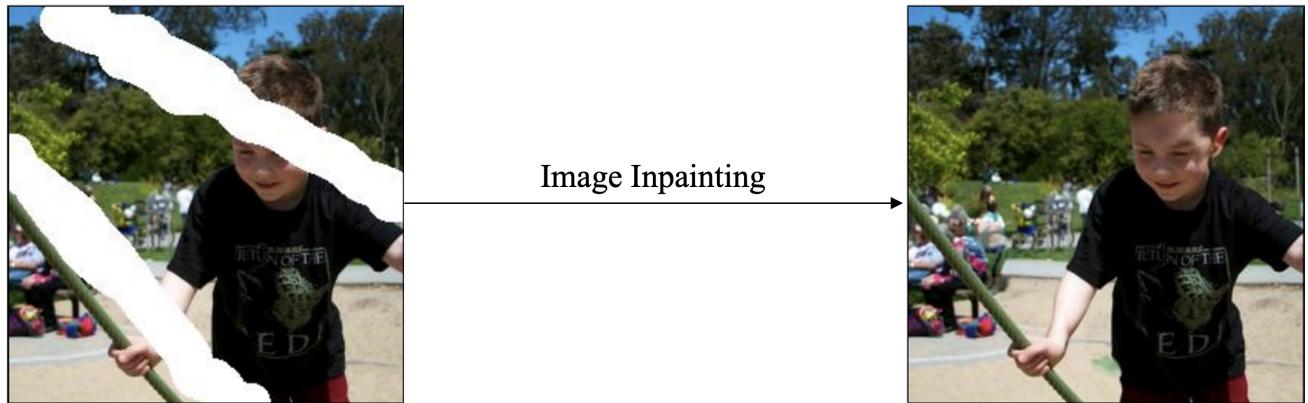


Exercise: Image Inpainting using Denoising Diffusion Probabilistic Model

Quoc-Thai Nguyen, Quang-Hien Ho và Quang-Vinh Dinh

Ngày 22 tháng 3 năm 2025

Phần 1. Giới thiệu

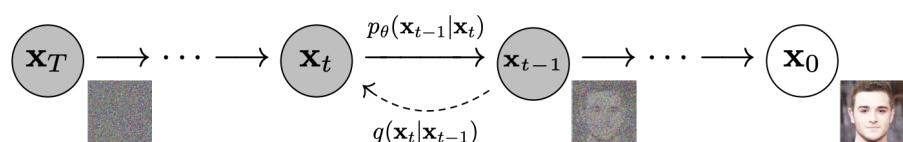


Hình 1: Ví dụ minh họa Image Inpainting sử dụng mô hình Denoising Diffusion Probabilistic Model.

Diffusion Models là một trong những mô hình sinh ngày càng được ứng dụng rộng rãi cho nhiều ứng dụng khác nhau như: Image Inpainting (Chỉnh sửa hình ảnh khuyết thiếu), Image Colorization (Tô màu hình ảnh),... Diffusion Models có nhiệm vụ tạo ra một phân phối cho dữ liệu đầu vào và xấp xỉ phân phối của dữ liệu được sinh ra với phân phối của dữ liệu gốc, từ đó giúp mô hình có thể sinh ra hình ảnh mới.

Trong phần này chúng ta sẽ tìm hiểu về ứng dụng Image Inpainting. Và sử dụng mô hình Denoising Diffusion Probabilistic Model để huấn luyện mô hình hoàn thiện các hình ảnh bị khuyết thiếu.

Diffusion Models bao gồm 2 quá trình: Forward Diffusion Process và Reverse Diffusion Process được mô tả như hình sau:



Hình 2: Minh họa quá trình hoạt động của Diffusion Models.

- (a) Forward Diffusion Process (FDP): Từ một điểm dữ liệu đầu vào x_0 thuộc một phân phối biết trước $x_0 \sim q(x)$, FDP sẽ thêm từ từ lần lượt theo thời gian một lượng nhỏ nhiễu được lấy mẫu từ phân phối Gauss $\epsilon \sim \mathcal{N}(\mu, \sigma^2)$ tạo ra các mẫu chứa nhiễu x_1, x_2, \dots, x_T , với T (steps) số bước thêm nhiễu vào.
- (b) Reverse Diffusion Process (RDP): Tại thời điểm x_T là mẫu chứa nhiễu, RDP sẽ tiến hành huấn luyện mô hình để khử nhiễu, khôi phục lại dữ liệu hình ảnh đầu vào. Mô hình được sử dụng trong quá trình decode khử nhiễu là UNet kết hợp với cơ chế Attention.

Phần 2. Image Inpainting using Denoising Diffusion Probabilistic Model

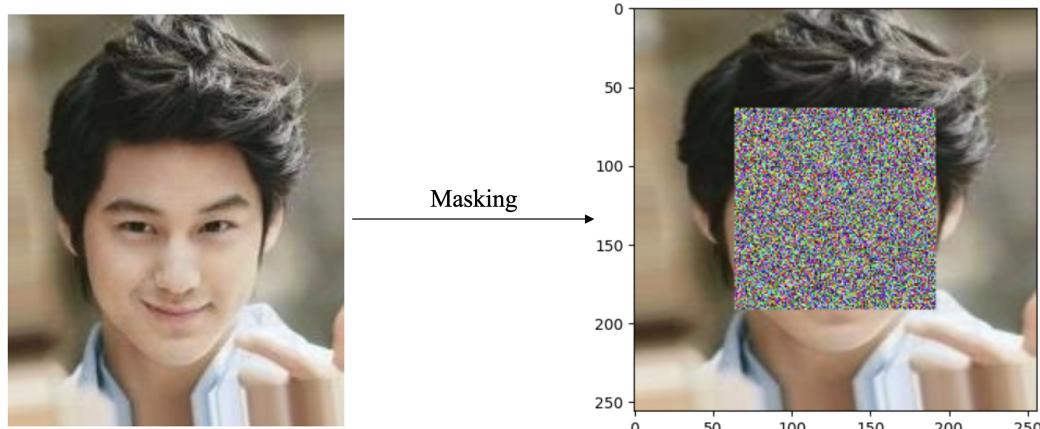
Trong phần này chúng ta sẽ huấn luyện mô hình Denoising Diffusion Probabilistic Model (DDP) ([paper](#)) để giải quyết bài toán Image Inpainting dựa vào bộ dữ liệu [CelebA](#)

Nội dung thực nghiệm bao gồm 5 phần:

- (a) Data Preparing: Chuẩn bị dữ liệu CelebA cho huấn luyện mô hình
- (b) Model: Xây dựng mô hình DDP
- (c) Loss, Metric: Xây dựng hàm mất mát và độ đo đánh giá
- (d) Trainer: Xây dựng các hàm để huấn luyện mô hình
- (e) Inference: Minh họa kết quả đạt được sau khi huấn luyện mô hình

1. Data Preparing

Bộ dữ liệu **CelebA** bao gồm hơn 200,000 ảnh khuôn mặt, vì vậy để huấn luyện mô hình Image Inpainting, trong phần này chúng ta sẽ tạo ra các ảnh mask. Có nhiều cách khác nhau để tạo ra ảnh mask từ ảnh gốc như: mask các điểm ảnh trung tâm, mask các điểm ảnh ở các góc hoặc mask các điểm ảnh ngẫu nhiên trên bức ảnh gốc. Để đơn giản, chúng ta sẽ chọn mask tại các vị trí điểm ảnh trung tâm của bức ảnh dưới dạng hình chữ nhật. Bên cạnh đó, chúng ta cũng thực hiện một số bước tiền xử lý như thay đổi kích thước các ảnh thành 256x256, chuẩn hoá các ảnh đầu vào.



Hình 3: Minh họa quá trình tạo ra ảnh mask từ ảnh gốc.

```

1 # Define mask
2 import os
3 import numpy as np
4
5 file_names = os.listdir('./img_align_celeba')
6 img_paths = ['./img_align_celeba/' + file_name for file_name in file_names]
7
8 # train: valid split
9 num_train = 150000
10 train_imgpaths = img_paths[: num_train]
11 val_imgpaths = img_paths[num_train :]
12
13 # generate mask image
14 def bbox2mask(img_shape, bbox, dtype='uint8'):
15     """
16     Generate mask in ndarray from bbox.
17     bbox (tuple[int]): Configuration tuple, (top, left, height, width)
18     """
19     height, width = img_shape[:2]
20     mask = np.zeros((height, width, 1), dtype=dtype)
21     mask[bbox[0]:bbox[0] + bbox[2], bbox[1]:bbox[1] + bbox[3], :] = 1
22     return mask

```

```

1 import torch
2 from PIL import Image
3 from torchvision import transforms
4 from torch.utils.data import Dataset
5
6 # build dataset
7 class InpaintingDataset(Dataset):
8     def __init__(self, img_paths, mask_mode, image_size=[256, 256]):
9         self.img_paths = img_paths
10        self.tfs = transforms.Compose([
11            transforms.Resize((image_size[0], image_size[1])),
12            transforms.ToTensor(),
13            transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
14        ])
15        self.mask_mode = mask_mode
16        self.image_size = image_size
17
18    def __getitem__(self, index):
19        img_path = self.img_paths[index]
20        img = Image.open(img_path).convert('RGB')
21        img = self.tfs(img)
22        mask = self.get_mask()
23        cond_image = img*(1. - mask) + mask*torch.randn_like(img)
24        mask_img = img*(1. - mask) + mask
25        return {
26            'gt_image': img,
27            'cond_image': cond_image,
28            'mask_image': mask_img,
29            'mask': mask,
30            'path': img_path
31        }
32
33    def __len__(self):
34        return len(self.img_paths)
35
36    def get_mask(self):
37        if self.mask_mode == 'center':
38            h, w = self.image_size
39            mask = bbox2mask(self.image_size, (h//4, w//4, h//2, w//2))
40
41        else:
42            raise NotImplementedError(
43                f'Mask mode {self.mask_mode} has not been implemented.')
44        return torch.from_numpy(mask).permute(2,0,1)

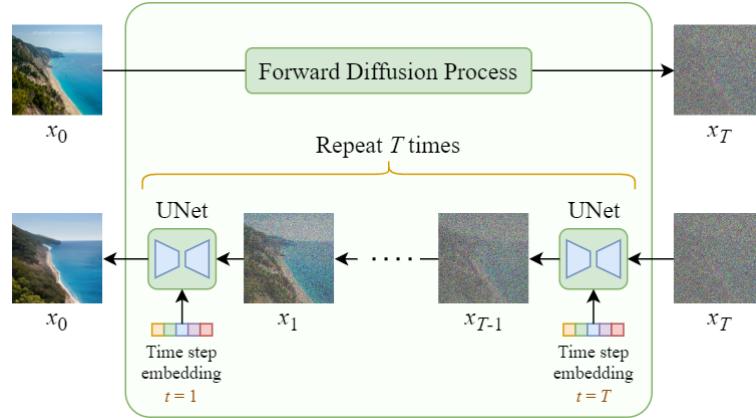
```

2. Model

Trong phần này chúng ta sẽ xây dựng mô hình cho quá trình RDP. Mô hình UNet sẽ được sử dụng làm mô hình khử nhiễu qua mỗi bước thời gian. Quá trình FDP sẽ thêm lần lượt theo bước thời gian T nhiễu Gauss vào ảnh đầu vào, vì vậy ở bước RDP chúng ta sẽ bổ sung thêm không gian embedding của bước thời gian vào mô hình UNet dựa vào hàm Sinusoidal Positional Embedding.

Phần xây dựng mô hình sẽ bao gồm 2 phần. Phần 1, chúng ta sẽ xây dựng mô hình UNet cơ bản chỉ bao gồm kiến trúc các Block của mô hình UNet gốc kết hợp với embedding của bước thời gian. Ở phần 2, chúng ta sẽ sử dụng mô hình UNet kết hợp thêm cơ chế Attention và Adaptive Group Normalization giúp cải tiến mô hình được giới thiệu trong bài [Diffusion Models Beat GAN on Image Synthesis](#)

2.1. Basic UNet Model



Hình 4: Minh họa quá trình RDP sử dụng kết hợp embedding bước thời gian vào mô hình UNet.

```

1 from torch import nn
2 import math
3 class Block(nn.Module):
4     def __init__(self, in_ch, out_ch, time_emb_dim, up=False):
5         super().__init__()
6         self.time_mlp = nn.Linear(time_emb_dim, out_ch)
7         if up:
8             self.conv1 = nn.Conv2d(2*in_ch, out_ch, 3, padding=1)
9             self.transform = nn.ConvTranspose2d(out_ch, out_ch, 4, 2, 1)
10        else:
11            self.conv1 = nn.Conv2d(in_ch, out_ch, 3, padding=1)
12            self.transform = nn.Conv2d(out_ch, out_ch, 4, 2, 1)
13            self.conv2 = nn.Conv2d(out_ch, out_ch, 3, padding=1)
14            self.bnrm1 = nn.BatchNorm2d(out_ch)
15            self.bnrm2 = nn.BatchNorm2d(out_ch)
16            self.relu = nn.ReLU()
17
18    def forward(self, x, t, ):
19        h = self.bnrm1(self.relu(self.conv1(x)))
20        time_emb = self.relu(self.time_mlp(t)) # Time embedding
21        time_emb = time_emb[(..., ) + (None, ) * 2]
22        h = h + time_emb # Add time channel
23        h = self.bnrm2(self.relu(self.conv2(h)))
24        return self.transform(h) # Down or Upsample

```

```

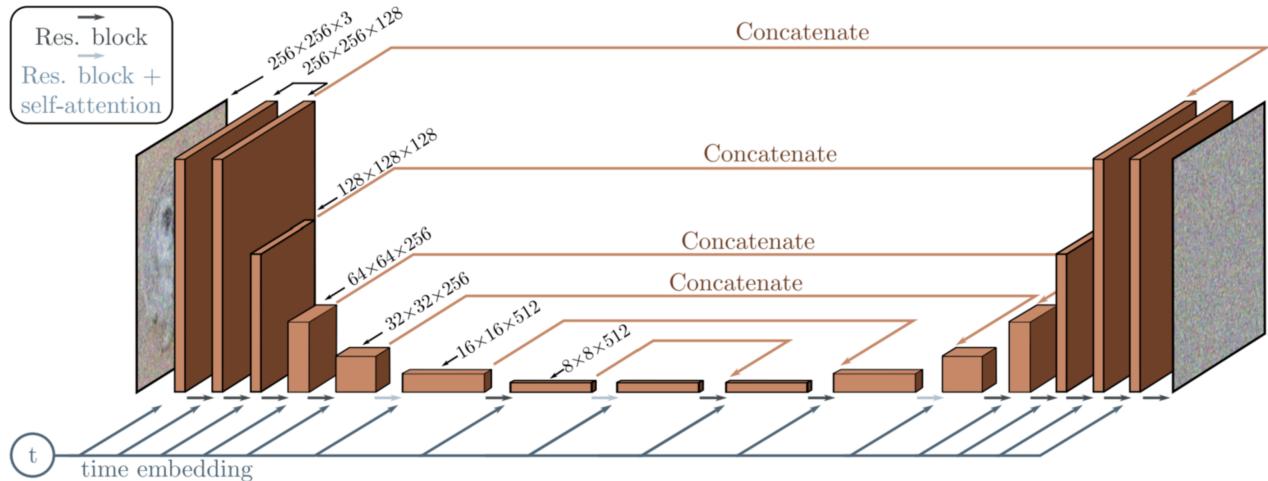
1 class SinusoidalPositionEmbeddings(nn.Module):
2     def __init__(self, dim):
3         super().__init__()
4         self.dim = dim
5     def forward(self, time):
6         device = time.device
7         half_dim = self.dim // 2
8         embeddings = math.log(10000) / (half_dim - 1)
9         embeddings = torch.exp(torch.arange(half_dim, device=device) * -
embeddings)
10        embeddings = time[:, None] * embeddings[None, :]
11        embeddings = torch.cat((embeddings.sin(), embeddings.cos()), dim=-1)
12        return embeddings
13
14 class SimpleUnet(nn.Module):
15     def __init__(self):
16         super().__init__()
17         image_channels = 3
18         down_channels = (64, 128, 256, 512, 1024)
19         up_channels = (1024, 512, 256, 128, 64)
20         out_dim = 3
21         time_emb_dim = 32
22         self.time_mlp = nn.Sequential(
23             SinusoidalPositionEmbeddings(time_emb_dim),
24             nn.Linear(time_emb_dim, time_emb_dim), nn.ReLU()
25         )
26         self.conv0 = nn.Conv2d(image_channels, down_channels[0], 3, padding=1)
27         self.downs = nn.ModuleList([Block(down_channels[i], down_channels[i+1],
time_emb_dim) for i in range(len(down_channels)-1)])
28         self.ups = nn.ModuleList([Block(up_channels[i], up_channels[i+1],
time_emb_dim, up=True) for i in range(len(up_channels)-1)])
29         self.output = nn.Conv2d(up_channels[-1], out_dim, 1)
30     def forward(self, x, timestep):
31         t = self.time_mlp(timestep)
32         x = self.conv0(x)
33         residual_inputs = []
34         for down in self.downs:
35             x = down(x, t)
36             residual_inputs.append(x)
37         for up in self.ups:
38             residual_x = residual_inputs.pop()
39             x = torch.cat((x, residual_x), dim=1)
40             x = up(x, t)
41         return self.output(x)

```

2.2. Improved UNet Model

Trong phần này chúng ta sẽ xây dựng mô hình UNet kết hợp với cơ chế Attention và Adaptive Group Normalization. Có thể tải về mã nguồn model [tại đây](#)

Kết trúc mô hình UNet sử dụng kết hợp với cơ chế Attention được mô tả như sau:



Hình 5: Mô hình UNet kết hợp cơ chế Attention trong DDP.

2.3. Gaussian Diffusion Model

Trong phần này chúng ta định nghĩa mô hình Diffusion Model đầy đủ với 2 bước: FDP ước lượng bước thời gian β dựa vào hàm linear, các giá trị xác suất để tính $q(x_t|x_0)$ và $q(x_{t-1}|x_t, x_0)$, thuật toán huấn luyện và lấy mẫu.

- Xây dựng hàm tạo các tham số

```

1 from tqdm import tqdm
2 from functools import partial
3
4 def make_beta_schedule(schedule, n_timestep, linear_start=1e-5, linear_end=1e-2):
5     if schedule == 'linear':
6         betas = np.linspace(
7             linear_start, linear_end, n_timestep, dtype=np.float64
8         )
9     else:
10        raise NotImplementedError(schedule)
11    return betas
12
13 def get_index_from_list(vals, t, x_shape=(1,1,1,1)):
14    """
15        Returns a specific index t of a passed list of values vals
16        while considering the batch dimension.
17    """
18    batch_size, *_ = t.shape
19    out = vals.gather(-1, t)
20    return out.reshape(batch_size, *((1,) * (len(x_shape) - 1))).to(device)

```

- Xây dựng hàm xác định các giá trị xác suất cho quá trình forward và reverse process.

```

1  class InpaintingGaussianDiffusion(nn.Module):
2      def __init__(self, unet_config, beta_schedule, **kwargs):
3          super(InpaintingGaussianDiffusion, self).__init__(**kwargs)
4          self.denoise_fn = UNet(**unet_config)
5          self.beta_schedule = beta_schedule
6      def set_new_noise_schedule(self, device):
7          to_torch = partial(torch.tensor, dtype=torch.float32, device=device)
8          betas = make_beta_schedule(**self.beta_schedule)
9          alphas = 1. - betas
10         timesteps, = betas.shape
11         self.num_timesteps = int(timesteps)
12         gammas = np.cumprod(alphas, axis=0) # alphas_cumprod
13         gammas_prev = np.append(1., gammas[:-1])
14         self.register_buffer("gammas", to_torch(gammas)) # q(x_t | x_{t-1})
15         self.register_buffer("sqrt_recip_gammas", to_torch(np.sqrt(1. / gammas)))
16         self.register_buffer("sqrt_recipm1_gammas", to_torch(np.sqrt(1. / gammas
- 1)))
17         posterior_variance = betas * (1. - gammas_prev) / (1. - gammas) # q(x_{t
-1} | x_t, x_0)
18         self.register_buffer("posterior_log_variance_clipped", to_torch(np.log(np
. maximum(posterior_variance, 1e-20))))
19         self.register_buffer("posterior_mean_coef1", to_torch(betas * np.sqrt(
gammas_prev) / (1. - gammas)))
20         self.register_buffer("posterior_mean_coef2", to_torch((1. - gammas_prev)
* np.sqrt(alphas) / (1. - gammas)))
21
22     def set_loss(self, loss_fn):
23         self.loss_fn = loss_fn
24     def predict_start_from_noise(self, y_t, t, noise):
25         return (
26             get_index_from_list(self.sqrt_recip_gammas, t, y_t.shape) * y_t -
27             get_index_from_list(self.sqrt_recipm1_gammas, t, y_t.shape) * noise
28         )
29     def q_posterior(self, y_0_hat, y_t, t):
30         # q(x_{t-1} | x_t, x_0)
31         posterior_mean = (
32             get_index_from_list(self.posterior_mean_coef1, t, y_t.shape) *
y_0_hat +
33             get_index_from_list(self.posterior_mean_coef2, t, y_t.shape) * y_t
34         )
35         posterior_log_variance_clipped = get_index_from_list(
36             self.posterior_log_variance_clipped, t, y_t.shape
37         )
38         return posterior_mean, posterior_log_variance_clipped

```

```
1     def p_mean_variance(self, y_t, t, clip_denoised: bool, y_cond=None):
2         noise_level = get_index_from_list(self.gammas, t, x_shape=(1, 1)).to(y_t.
device)
3         y_0_hat = self.predict_start_from_noise(
4             y_t, t=t, noise=self.denoise_fn(torch.cat([y_cond, y_t], dim=1),
noise_level))
5         if clip_denoised:
6             y_0_hat.clamp_(-1., 1.)
7         model_mean, posterior_log_variance = self.q_posterior(
8             y_0_hat=y_0_hat, y_t=y_t, t=t)
9         return model_mean, posterior_log_variance
10
11    def q_sample(self, y_0, sample_gammas, noise=None):
12        noise = noise if noise is not None else torch.randn_like(y_0)
13        return (
14            sample_gammas.sqrt() * y_0 +
15            (1 - sample_gammas).sqrt() * noise
16        )
17
18    def forward(self, y_0, y_cond=None, mask=None, noise=None):
19        # sampling from p(gamma)
20        b, *_ = y_0.shape
21        t = torch.randint(1, self.num_timesteps, (b,), device=y_0.device).long()
22
23        gamma_t1 = get_index_from_list(self.gammas, t-1, x_shape=(1, 1))
24        sqrt_gamma_t2 = get_index_from_list(self.gammas, t, x_shape=(1, 1))
25        sample_gammas = (sqrt_gamma_t2-gamma_t1) * torch.rand((b, 1), device=y_0.
device) + gamma_t1
26        sample_gammas = sample_gammas.view(b, -1)
27
28        noise = noise if noise is not None else torch.randn_like(y_0)
29        y_noisy = self.q_sample(
30            y_0=y_0, sample_gammas=sample_gammas.view(-1, 1, 1, 1), noise=noise)
31
32        if mask is not None:
33            noise_hat = self.denoise_fn(torch.cat([y_cond, y_noisy*mask+(1.-mask)
*y_0], dim=1), sample_gammas)
34            loss = self.loss_fn(mask*noise, mask*noise_hat)
35
36        else:
37            noise_hat = self.denoise_fn(torch.cat([y_cond, y_noisy], dim=1),
sample_gammas)
38            loss = self.loss_fn(noise, noise_hat)
39        return loss
```

```

1  @torch.no_grad()
2  def p_sample(self, y_t, t, clip_denoised=True, y_cond=None):
3      model_mean, model_log_variance = self.p_mean_variance(
4          y_t=y_t, t=t, clip_denoised=clip_denoised, y_cond=y_cond)
5      noise = torch.randn_like(y_t) if any(t>0) else torch.zeros_like(y_t)
6      return model_mean + noise * (0.5 * model_log_variance).exp()
7  @torch.no_grad()
8  def restoration(self, y_cond, y_t=None, y_0=None, mask=None, sample_num=8):
9      b, *_ = y_cond.shape
10
11     sample_inter = (self.num_timesteps//sample_num)
12
13     y_t = y_t if y_t is not None else torch.randn_like(y_cond)
14     ret_arr = y_t
15     for i in reversed(range(0, self.num_timesteps)):
16         t = torch.full((b,), i, device=y_cond.device, dtype=torch.long)
17         y_t = self.p_sample(y_t, t, y_cond=y_cond)
18         if mask is not None:
19             y_t = y_0*(1.-mask) + mask*y_t
20         if i % sample_inter == 0:
21             ret_arr = torch.cat([ret_arr, y_t], dim=0)
22     return y_t, ret_arr

```

3. Loss. Metric

Trong phần này chúng ta xây dựng hàm mất mát và độ đo đánh giá mô hình

```

1 import torch.nn.functional as F
2
3 def mse_loss(output, target):
4     return F.mse_loss(output, target)
5
6
7 def mae(input, target):
8     with torch.no_grad():
9         loss = nn.L1Loss()
10        output = loss(input, target)
11    return output

```

4. Trainer

Trong phần này chúng ta xây dựng hàm huấn luyện mô hình.

```
1 import time
2 class Trainer():
3     def __init__(self, model, optimizers, train_loader, val_loader, epochs,
4                  sample_num, device, save_model):
5         self.model = model.to(device)
6         self.optimizer = torch.optim.Adam(list(filter(lambda p: p.requires_grad,
7                                                       self.model.parameters())), **optimizers)
8         self.model.set_loss(mse_loss)
9         self.model.set_new_noise_schedule(device)
10        self.sample_num = sample_num
11        self.train_loader = train_loader
12        self.val_loader = val_loader
13        self.device = device
14        self.epochs = epochs
15        self.save_model = save_model + "/best_model.pth"
16    def train_step(self):
17        losses = []
18        for batch in tqdm(self.train_loader):
19            gt_image = batch['gt_image'].to(self.device)
20            cond_image = batch['cond_image'].to(self.device)
21            mask = batch['mask'].to(self.device)
22            mask_image = batch['mask_image'].to(self.device)
23            batch_size = len(batch['path'])
24            self.optimizer.zero_grad()
25            loss = self.model(gt_image, cond_image, mask=mask)
26            loss.backward()
27            losses.append(loss.item())
28            self.optimizer.step()
29        return sum(losses)/len(losses)
30    def val_step(self):
31        losses, metrics = [], []
32        with torch.no_grad():
33            for batch in tqdm(self.val_loader):
34                gt_image = batch['gt_image'].to(self.device)
35                cond_image = batch['cond_image'].to(self.device)
36                mask = batch['mask'].to(self.device)
37                mask_image = batch['mask_image'].to(self.device)
38                loss = self.model(gt_image, cond_image, mask=mask)
39                output, visuals = self.model.restoration(cond_image, y_t=
40                cond_image, y_0=gt_image, mask=mask, sample_num=self.sample_num)
41                mae_score = mae(gt_image, output)
42                losses.append(loss.item())
43                metrics.append(mae_score.item())
44        return sum(losses)/len(losses), sum(metrics)/len(metrics)
```

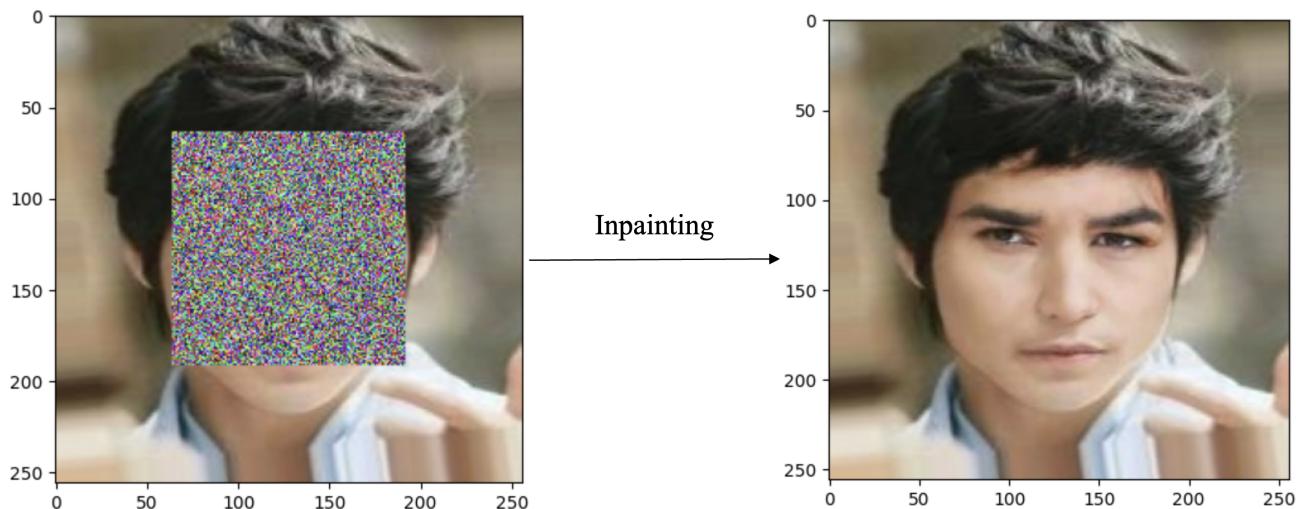
```
1  def train(self):
2      best_mae = 100000
3      for epoch in range(self.epochs):
4          epoch_start_time = time.time()
5          train_loss = self.train_step()
6          val_loss, val_mae = self.val_step()
7          if val_mae < best_mae:
8              torch.save(self.model.state_dict(), self.save_model)
9          # Print loss, acc end epoch
10         print("-" * 59)
11         print(
12             "| End of epoch {:3d} | Time: {:5.2f}s | Train Loss {:8.3f} "
13             "| Valid Loss {:8.3f} | Valid MAE {:8.3f} ".format(
14                 epoch+1, time.time() - epoch_start_time, train_loss, val_loss
15                 , val_mae
16                 )
17         )
18         self.model.load_state_dict(torch.load(self.save_model))
19
20 epochs = 200 # 5
21 sample_num = 8
22 save_model = "./save_model"
23 optimizers = { "lr": 5e-5, "weight_decay": 0}
24 device = "cuda" if torch.cuda.is_available() else "cpu"
25
26 unet_config = {
27     "in_channel": 6, "out_channel": 3, "inner_channel": 64,
28     "channel_mults": [1, 2, 4, 8], "attn_res": [16], "num_head_channels": 32, "
29     res_blocks": 2, "dropout": 0.2, "image_size": 256
30 }
31 beta_schedule = {
32     "schedule": "linear", "n_timestep": 20, "linear_start": 1e-4, "linear_end": "
33     0.09
34 }
35 inpainting_model = InpaintingGaussianDiffusion(unet_config, beta_schedule)
36
37 trainer = Trainer(
38     inpainting_model, optimizers, train_loader, val_loader, epochs, sample_num,
39     device, save_model
40 )
```

5. Inference

Sau quá trình huấn luyện, checkpoint của mô hình tốt nhất có thể tải về [tại đây](#) và sử dụng để sinh ảnh mới từ ảnh mask.

```
1 # load model
2 inpainting_model = InpaintingGaussianDiffusion(unet_config, beta_schedule)
3 inpainting_model.set_new_noise_schedule(device)
4 load_state = torch.load('./save_model/best_model_200.pth')
5 inpainting_model.load_state_dict(load_state, strict=True)
6 inpainting_model.eval().to(device)
7
8 # test image
9 test_imgpath = img_paths[16]
10 test_dataset = InpaintingDataset([test_imgpath], mask_mode='center')
11 test_sample = next(iter(test_dataset))
12
13 def inference(model, test_sample):
14     with torch.no_grad():
15         output, visuals = model.restoration(
16             test_sample['cond_image'].unsqueeze(0).to(device),
17             y_t=test_sample['cond_image'].unsqueeze(0).to(device),
18             y_0=test_sample['cond_image'].unsqueeze(0).to(device),
19             mask=test_sample['mask'].unsqueeze(0).to(device)
20         )
21     return output, visuals
22
23 output, visuals = inference(inpainting_model, test_sample)
24
25 # show result
26 def show_tensor_image(image, show=True):
27     reverse_transforms = transforms.Compose([
28         transforms.Lambda(t: (t + 1) / 2),
29         transforms.Lambda(t: t.permute(1, 2, 0)), # CHW to HWC
30         transforms.Lambda(t: t * 255.),
31         transforms.Lambda(t: t.numpy().astype(np.uint8)),
32         transforms.ToPILImage(),
33     ])
34
35 # Take first image of batch
36 if len(image.shape) == 4:
37     image = image[0, :, :, :]
38 if show:
39     plt.imshow(reverse_transforms(image))
40 else
```

Kết quả thử nghiệm



Hình 6: Kết quả thực nghiệm mô hình sau khi huấn luyện.

6. Deployment

Triển khai mô hình trên streamlit. Tham khảo về [code](#) và [demo](#).

Image Inpainting using Denoising Diffusion Probabilistic Model

Model: DDPM - Repaint. Dataset: CelebA-HQ

How would you like to give the input?

Upload Image File

Please upload an image



Drag and drop file here

Limit 200MB per file • JPG, PNG, JPEG

Browse files



example.png 117.2KB



Hình 7: Triển khai ứng dụng trên Streamlit.

Phần 3. Câu hỏi trắc nghiệm

Câu hỏi 1 Mục tiêu của Forward Diffusion Process trong mô hình Diffusion là gì?

- a) Thêm một lượng nhỏ nhiễu được lấy mẫu từ phân phối Gauss vào giá trị input x_0 lần lượt theo bước nhảy T với lịch trình phương sai β_1, \dots, β_T
- b) Khử nhiễu trong x_T để khôi phục giá trị đầu vào
- c) Cả 2 đáp án trên đều đúng
- d) Cả 2 đáp án trên đều sai

Câu hỏi 2 Mục tiêu của Reverse Diffusion Process trong mô hình Diffusion là gì?

- a) Thêm một lượng nhỏ nhiễu được lấy mẫu từ phân phối Gauss vào giá trị input x_0 lần lượt theo bước nhảy T với lịch trình phương sai β_1, \dots, β_T
- b) Khử nhiễu trong x_T để khôi phục giá trị đầu vào
- c) Cả 2 đáp án trên đều đúng
- d) Cả 2 đáp án trên đều sai

Câu hỏi 3 Dựa vào thực nghiệm trong phần 2, lịch trình phương sai β được tính dựa vào hàm nào sau đây?

- a) Linspace
- b) Sigmoid
- c) ReLU
- d) Tanh

Câu hỏi 4 Dựa vào thực nghiệm trong phần 2, phương pháp mask nào sau đây được sử dụng?

- a) Mask các vị trí điểm ảnh trung tâm theo hình chữ nhật
- b) Mask các vị trí điểm ảnh ở góc trái trên theo hình chữ nhật
- c) Mask các vị trí điểm ảnh ở góc trái dưới theo hình chữ nhật
- d) Mask các vị trí điểm ảnh trung tâm theo hình tròn

Câu hỏi 5 Dựa vào bài báo cáo nghiên cứu "Denoising Diffusion Probabilistic Model", công thức tính giá trị xác suất $q(x_t|x_0)$ là?

- a) $x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1 - \bar{\alpha}_t}\epsilon$
- b) $x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1 - \alpha_t}\epsilon$
- c) $x_t = \sqrt{1 - \bar{\alpha}_t}x_0 + \sqrt{1 - \alpha_t}\epsilon$
- d) $x_t = \sqrt{1 - \bar{\alpha}_t}x_0 + \sqrt{1 - \bar{\alpha}_t}\epsilon$

Câu hỏi 6 Dựa vào bài báo cáo nghiên cứu "Denoising Diffusion Probabilistic Model", công thức tính $\tilde{\beta}_t$ trong $q(x_{t-1}|x_t, x_0)$ là?

- a) $\tilde{\beta}_t = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}$
- b) $\tilde{\beta}_t = \frac{1-\bar{\alpha}_t}{1-\bar{\alpha}_{t-1}}\beta_t$
- c) $\tilde{\beta}_t = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t$
- d) $\tilde{\beta}_t = \frac{1}{1-\bar{\alpha}_t}\beta_t$

Câu hỏi 7 Dựa vào phần thực nghiệm 2, phương pháp embedding cho bước thời gian được sử dụng khi xây dựng mô hình Basic UNet là?

- a) ALiBi
- b) Rotary Embedding
- c) Conditional Positional Embedding
- d) Sinusoidal Positional Embedding

Câu hỏi 8 Dựa vào phần thực nghiệm 2, bộ dữ liệu huấn luyện mô hình nào được sử dụng?

- a) CelebA
- b) MNIST
- c) CIFAR10
- d) CIFAR100

Câu hỏi 9 Dựa vào phần thực nghiệm 2, hàm loss nào sau đây không được sử dụng?

- a) BCELoss
- b) MSELoss
- c) L1Loss
- d) CTCLoss

Câu hỏi 10 Dựa vào phần thực nghiệm 2, độ đo đánh giá mô hình nào sau đây không được sử dụng?

- a) F1
- b) Recall
- c) Precision
- d) MAE

Phần 4. Phụ lục

1. **Hint:** Dựa vào file tải về [Image-Inpainting-Denoising-Diffusion-Model](#) để hoàn thiện các đoạn code.
2. **Solution:** Các file code cài đặt hoàn chỉnh và phần trả lời nội dung trắc nghiệm có thể được tải về [tại đây](#) (Lưu ý: Sáng thứ 3 khi hết deadline phần project, admin mới copy các nội dung bài giải nêu trên vào đường dẫn).

3. **Rubric:**

Phần	Kiến Thức	Đánh Giá
1	<ul style="list-style-type: none">- Hiểu rõ bài toán Image Inpainting- Hiểu rõ mô hình Denoising Diffusion Probabilistic Model	<ul style="list-style-type: none">- Các bước thực hiện trong mô hình: Forward Diffusion Process thêm nhiễu và Reverse Diffusion Process để khử nhiễu.
2.	<ul style="list-style-type: none">- Hiểu rõ mô hình UNet sử dụng trong Diffusion Model- Các kỹ thuật cải tiến mô hình UNet	<ul style="list-style-type: none">- Xây dựng mô hình Image Inpainting dựa trên bộ dữ liệu CelebA- Cải tiến mô hình DCGAN với DistilBERT

- *Hết* -