

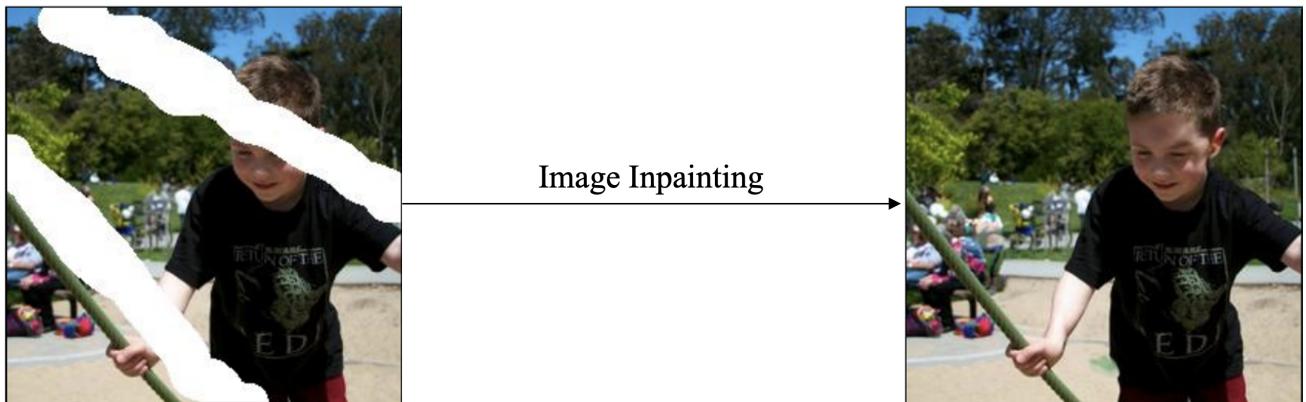
Exercise: Image Inpainting using Denoising Diffusion Probabilistic Model

Quoc-Thai Nguyen và Quang-Vinh Dinh

PR-Team: Đăng-Nhã Nguyễn, Minh-Châu Phạm và Hoàng-Nguyễn Vũ

Ngày 25 tháng 3 năm 2024

Phần I. Giới thiệu

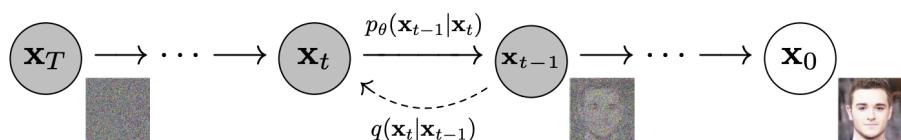


Hình 1: Ví dụ minh họa Image Inpainting sử dụng mô hình Denoising Diffusion Probabilistic Model.

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 II. Image Inpainting using Denoising Diffusion Probabilistic Model

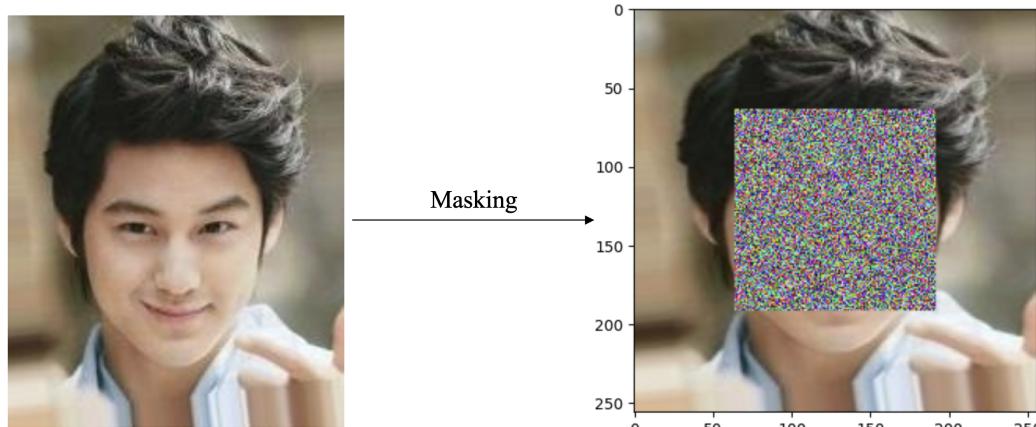
Trong phần này chúng ta sẽ huấn luyện mô hình Denoising Diffusion Probabilistic Model (DDP) ([paper](#)) để giải quyết bài toán Image Inpainting dựa vào bộ dữ liệu [CelebA](#)

Nội dung thực nghiệm bao gồm 5 phần:

- Data Preparing: Chuẩn bị dữ liệu CelebA cho huấn luyện mô hình
- Model: Xây dựng mô hình DDP
- Loss, Metric: Xây dựng hàm mất mát và độ đo đánh giá
- Trainer: Xây dựng các hàm để huấn luyện mô hình
- 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 hóa các ảnh đầu vào.



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

```

1 # load the dataset
2 import os
3 import numpy as np
4 import torch
5 from PIL import Image
6 from torchvision import transforms
7 from torch.utils.data import Dataset
8
9
10 file_names = os.listdir('./img_align_celeba')
11 img_paths = ['./img_align_celeba/' + file_name for file_name in file_names]
12

```

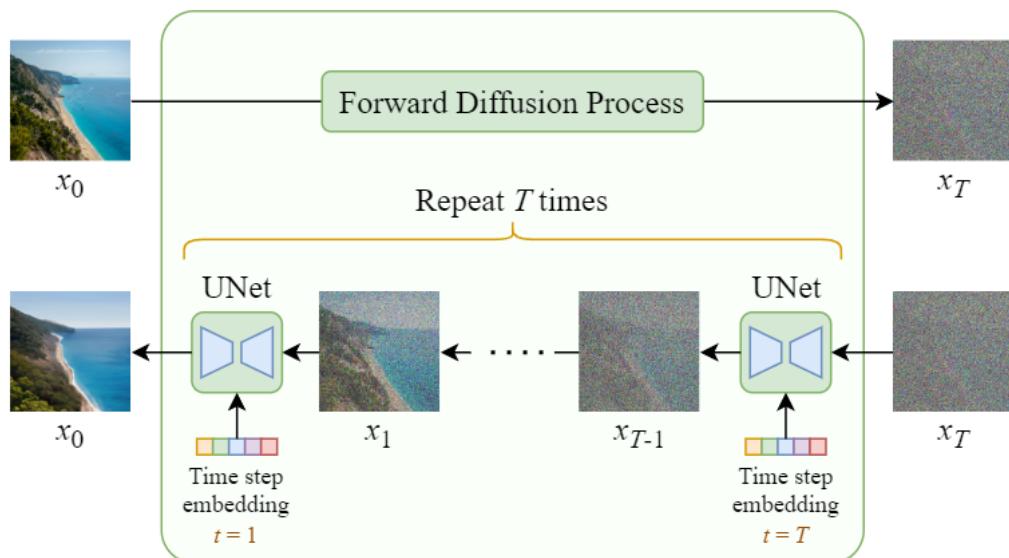
```
13 # train: valid split
14 num_train = 150000
15 train_imgpaths = img_paths[: num_train]
16 val_imgpaths = img_paths[num_train :]
17
18 # generate mask image
19 def bbox2mask(img_shape, bbox, dtype='uint8'):
20     """
21     Generate mask in ndarray from bbox.
22     bbox (tuple[int]): Configuration tuple, (top, left, height, width)
23     """
24
25     height, width = img_shape[:2]
26
27     mask = np.zeros((height, width, 1), dtype=dtype)
28     mask[bbox[0]:bbox[0] + bbox[2], bbox[1]:bbox[1] + bbox[3], :] = 1
29
30     return mask
31
32 # build dataset
33 class InpaintingDataset():
34     def __init__(self, img_paths, mask_mode, image_size=[256, 256]):
35         self.img_paths = img_paths
36         self.tfs = transforms.Compose([
37             transforms.Resize((image_size[0], image_size[1])),
38             transforms.ToTensor(),
39             transforms.Normalize(mean=[0.5, 0.5, 0.5], std=[0.5, 0.5, 0.5])
40         ])
41         self.mask_mode = mask_mode
42         self.image_size = image_size
43
44     def __getitem__(self, index):
45         img_path = self.img_paths[index]
46         img = Image.open(img_path).convert('RGB')
47         img = self.tfs(img)
48         mask = self.get_mask()
49         cond_image = img*(1. - mask) + mask*torch.randn_like(img)
50         mask_img = img*(1. - mask) + mask
51         return {
52             'gt_image': img,
53             'cond_image': cond_image,
54             'mask_image': mask_img,
55             'mask': mask,
56             'path': img_path
57         }
58
59     def __len__(self):
60         return len(self.img_paths)
61
62     def get_mask(self):
63         if self.mask_mode == 'center':
64             h, w = self.image_size
65             mask = bbox2mask(self.image_size, (h//4, w//4, h//2, w//2))
66
67         else:
68             raise NotImplementedError(
69                 f'Mask mode {self.mask_mode} has not been implemented.')
70         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
4
5 class Block(nn.Module):
6     def __init__(self, in_ch, out_ch, time_emb_dim, up=False):
7         super().__init__()
8         self.time_mlp = nn.Linear(time_emb_dim, out_ch)
9         if up:
10             self.conv1 = nn.Conv2d(2*in_ch, out_ch, 3, padding=1)
11             self.transform = nn.ConvTranspose2d(out_ch, out_ch, 4, 2, 1)
12         else:
13             self.conv1 = nn.Conv2d(in_ch, out_ch, 3, padding=1)
14             self.transform = nn.Conv2d(out_ch, out_ch, 4, 2, 1)
15             self.conv2 = nn.Conv2d(out_ch, out_ch, 3, padding=1)
16             self.bnrm1 = nn.BatchNorm2d(out_ch)
17             self.bnrm2 = nn.BatchNorm2d(out_ch)
18             self.relu = nn.ReLU()
19
20     def forward(self, x, t, ):
21         # First Conv
22         h = self.bnrm1(self.relu(self.conv1(x)))

```

```

23     # Time embedding
24     time_emb = self.relu(self.time_mlp(t))
25     # Extend last 2 dimensions
26     time_emb = time_emb[(..., ) + (None, ) * 2]
27     # Add time channel
28     h = h + time_emb
29     # Second Conv
30     h = self.bn2d(self.relu(self.conv2(h)))
31     # Down or Upsample
32     return self.transform(h)
33
34 class SinusoidalPositionEmbeddings(nn.Module):
35     def __init__(self, dim):
36         super().__init__()
37         self.dim = dim
38
39     def forward(self, time):
40         device = time.device
41         half_dim = self.dim // 2
42         embeddings = math.log(10000) / (half_dim - 1)
43         embeddings = torch.exp(torch.arange(half_dim, device=device) * -embeddings)
44         embeddings = time[:, None] * embeddings[None, :]
45         embeddings = torch.cat((embeddings.sin(), embeddings.cos()), dim=-1)
46         # TODO: Double check the ordering here
47         return embeddings
48
49
50 class SimpleUnet(nn.Module):
51     """
52     A simplified variant of the Unet architecture.
53     """
54     def __init__(self):
55         super().__init__()
56         image_channels = 3
57         down_channels = (64, 128, 256, 512, 1024)
58         up_channels = (1024, 512, 256, 128, 64)
59         out_dim = 3
60         time_emb_dim = 32
61
62         # Time embedding
63         self.time_mlp = nn.Sequential(
64             SinusoidalPositionEmbeddings(time_emb_dim),
65             nn.Linear(time_emb_dim, time_emb_dim),
66             nn.ReLU()
67         )
68
69         # Initial projection
70         self.conv0 = nn.Conv2d(image_channels, down_channels[0], 3, padding=1)
71
72         # Downsample
73         self.downs = nn.ModuleList([Block(down_channels[i], down_channels[i+1], \
74                                         time_emb_dim) \
75                                         for i in range(len(down_channels)-1)])
76         # Upsample
77         self.ups = nn.ModuleList([Block(up_channels[i], up_channels[i+1], \
78                                     time_emb_dim, up=True) \
79                                     for i in range(len(up_channels)-1)])
80
81         # Edit: Corrected a bug found by Jakub C (see YouTube comment)
82         self.output = nn.Conv2d(up_channels[-1], out_dim, 1)

```

```

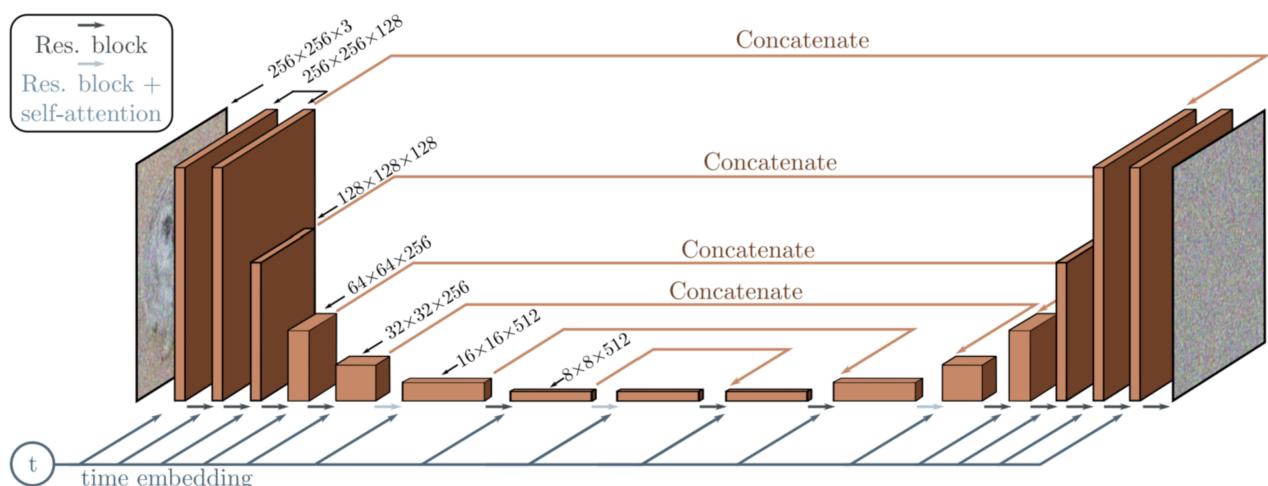
83
84     def forward(self, x, timestep):
85         # Embedd time
86         t = self.time_mlp(timestep)
87         # Initial conv
88         x = self.conv0(x)
89         # Unet
90         residual_inputs = []
91         for down in self.downs:
92             x = down(x, t)
93             residual_inputs.append(x)
94         for up in self.ups:
95             residual_x = residual_inputs.pop()
96             # Add residual x as additional channels
97             x = torch.cat((x, residual_x), dim=1)
98             x = up(x, t)
99         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.

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.

```

1 # Reference: https://github.com/openai/guided-diffusion
2 """
3 Various utilities for neural networks.
4 """
5 import math
6 import numpy as np
7 import torch
8 import torch.nn as nn
9
10
11 class GroupNorm32(nn.GroupNorm):
12     def forward(self, x):
13         return super().forward(x.float()).type(x.dtype)
14
15
16 def zero_module(module):
17     """

```

```
18     Zero out the parameters of a module and return it.
19     """
20     for p in module.parameters():
21         p.detach().zero_()
22     return module
23
24
25 def scale_module(module, scale):
26     """
27     Scale the parameters of a module and return it.
28     """
29     for p in module.parameters():
30         p.detach().mul_(scale)
31     return module
32
33
34 def mean_flat(tensor):
35     """
36     Take the mean over all non-batch dimensions.
37     """
38     return tensor.mean(dim=list(range(1, len(tensor.shape))))
39
40
41 def normalization(channels):
42     """
43     Make a standard normalization layer.
44
45     :param channels: number of input channels.
46     :return: an nn.Module for normalization.
47     """
48     return GroupNorm32(32, channels)
49
50
51
52 def checkpoint(func, inputs, params, flag):
53     """
54     Evaluate a function without caching intermediate activations, allowing for
55     reduced memory at the expense of extra compute in the backward pass.
56
57     :param func: the function to evaluate.
58     :param inputs: the argument sequence to pass to 'func'.
59     :param params: a sequence of parameters 'func' depends on but does not
60                     explicitly take as arguments.
61     :param flag: if False, disable gradient checkpointing.
62     """
63
64     if flag:
65         args = tuple(inputs) + tuple(params)
66         return CheckpointFunction.apply(func, len(inputs), *args)
67     else:
68         return func(*inputs)
69
70
71 class CheckpointFunction(torch.autograd.Function):
72     @staticmethod
73     def forward(ctx, run_function, length, *args):
74         ctx.run_function = run_function
75         ctx.input_tensors = list(args[:length])
76         ctx.input_params = list(args[length:])
77         with torch.no_grad():
78             output_tensors = ctx.run_function(*ctx.input_tensors)
```

```

78         return output_tensors
79
80     @staticmethod
81     def backward(ctx, *output_grads):
82         ctx.input_tensors = [x.detach().requires_grad_(True) for x in ctx.
83         input_tensors]
84         with torch.enable_grad():
85             # Fixes a bug where the first op in run_function modifies the
86             # Tensor storage in place, which is not allowed for detach()'d
87             # Tensors.
88             shallow_copies = [x.view_as(x) for x in ctx.input_tensors]
89             output_tensors = ctx.run_function(*shallow_copies)
90             input_grads = torch.autograd.grad(
91                 output_tensors,
92                 ctx.input_tensors + ctx.input_params,
93                 output_grads,
94                 allow_unused=True,
95             )
96             del ctx.input_tensors
97             del ctx.input_params
98             del output_tensors
99             return (None, None) + input_grads
100
101 def count_flops_attn(model, _x, y):
102     """
103     A counter for the 'thop' package to count the operations in an
104     attention operation.
105     Meant to be used like:
106         macs, params = thop.profile(
107             model,
108             inputs=(inputs, timestamps),
109             custom_ops={QKVAAttention: QKVAAttention.count_flops},
110         )
111     """
112     b, c, *spatial = y[0].shape
113     num_spatial = int(np.prod(spatial))
114     # We perform two matmuls with the same number of ops.
115     # The first computes the weight matrix, the second computes
116     # the combination of the value vectors.
117     matmul_ops = 2 * b * (num_spatial ** 2) * c
118     model.total_ops += torch.DoubleTensor([matmul_ops])
119
120
121 def gamma_embedding(gammas, dim, max_period=10000):
122     """
123     Create sinusoidal timestep embeddings.
124     :param gammas: a 1-D Tensor of N indices, one per batch element.
125             These may be fractional.
126     :param dim: the dimension of the output.
127     :param max_period: controls the minimum frequency of the embeddings.
128     :return: an [N x dim] Tensor of positional embeddings.
129     """
130     half = dim // 2
131     freqs = torch.exp(
132         -math.log(max_period) * torch.arange(start=0, end=half, dtype=torch.float32) /
133         half
134     ).to(device=gammas.device)
135     args = gammas[:, None].float() * freqs[None]
136     embedding = torch.cat([torch.cos(args), torch.sin(args)], dim=-1)

```

```
136     if dim % 2:
137         embedding = torch.cat([embedding, torch.zeros_like(embedding[:, :1])], dim=-1)
138     return embedding
1
1 import math
2 import torch
3 import torch.nn as nn
4 import torch.nn.functional as F
5
6 from abc import abstractmethod
7
8 class SiLU(nn.Module):
9     def forward(self, x):
10        return x * torch.sigmoid(x)
11
12 class EmbedBlock(nn.Module):
13     """
14     Any module where forward() takes embeddings as a second argument.
15     """
16
17     @abstractmethod
18     def forward(self, x, emb):
19         """
20         Apply the module to 'x' given 'emb' embeddings.
21         """
22
23 class EmbedSequential(nn.Sequential, EmbedBlock):
24     """
25     A sequential module that passes embeddings to the children that
26     support it as an extra input.
27     """
28
29     def forward(self, x, emb):
30         for layer in self:
31             if isinstance(layer, EmbedBlock):
32                 x = layer(x, emb)
33             else:
34                 x = layer(x)
35         return x
36
37 class Upsample(nn.Module):
38     """
39     An upsampling layer with an optional convolution.
40     :param channels: channels in the inputs and outputs.
41     :param use_conv: a bool determining if a convolution is applied.
42     """
43
44
45     def __init__(self, channels, use_conv, out_channel=None):
46         super().__init__()
47         self.channels = channels
48         self.out_channel = out_channel or channels
49         self.use_conv = use_conv
50         if use_conv:
51             self.conv = nn.Conv2d(self.channels, self.out_channel, 3, padding=1)
52
53     def forward(self, x):
54         assert x.shape[1] == self.channels
55         x = F.interpolate(x, scale_factor=2, mode="nearest")
56         if self.use_conv:
57             x = self.conv(x)
```

```
58         return x
59
60     class Downsample(nn.Module):
61         """
62             A downsampling layer with an optional convolution.
63             :param channels: channels in the inputs and outputs.
64             :param use_conv: a bool determining if a convolution is applied.
65         """
66
67         def __init__(self, channels, use_conv, out_channel=None):
68             super().__init__()
69             self.channels = channels
70             self.out_channel = out_channel or channels
71             self.use_conv = use_conv
72             stride = 2
73             if use_conv:
74                 self.op = nn.Conv2d(
75                     self.channels, self.out_channel, 3, stride=stride, padding=1
76                 )
77             else:
78                 assert self.channels == self.out_channel
79                 self.op = nn.AvgPool2d(kernel_size=stride, stride=stride)
80
81         def forward(self, x):
82             assert x.shape[1] == self.channels
83             return self.op(x)
84
85
86     class ResBlock(EmbedBlock):
87         """
88             A residual block that can optionally change the number of channels.
89             :param channels: the number of input channels.
90             :param emb_channels: the number of embedding channels.
91             :param dropout: the rate of dropout.
92             :param out_channel: if specified, the number of out channels.
93             :param use_conv: if True and out_channel is specified, use a spatial
94                 convolution instead of a smaller 1x1 convolution to change the
95                 channels in the skip connection.
96             :param use_checkpoint: if True, use gradient checkpointing on this module.
97             :param up: if True, use this block for upsampling.
98             :param down: if True, use this block for downsampling.
99         """
100
101        def __init__(
102            self,
103            channels,
104            emb_channels,
105            dropout,
106            out_channel=None,
107            use_conv=False,
108            use_scale_shift_norm=False,
109            use_checkpoint=False,
110            up=False,
111            down=False,
112        ):
113            super().__init__()
114            self.channels = channels
115            self.emb_channels = emb_channels
116            self.dropout = dropout
117            self.out_channel = out_channel or channels
```

```
118     self.use_conv = use_conv
119     self.use_checkpoint = use_checkpoint
120     self.use_scale_shift_norm = use_scale_shift_norm
121
122     self.in_layers = nn.Sequential(
123         normalization(channels),
124         SiLU(),
125         nn.Conv2d(channels, self.out_channel, 3, padding=1),
126     )
127
128     self.updown = up or down
129
130     if up:
131         self.h_upd = Upsample(channels, False)
132         self.x_upd = Upsample(channels, False)
133     elif down:
134         self.h_upd = Downsample(channels, False)
135         self.x_upd = Downsample(channels, False)
136     else:
137         self.h_upd = self.x_upd = nn.Identity()
138
139     self.emb_layers = nn.Sequential(
140         SiLU(),
141         nn.Linear(
142             emb_channels,
143             2 * self.out_channel if use_scale_shift_norm else self.out_channel,
144         ),
145     )
146     self.out_layers = nn.Sequential(
147         normalization(self.out_channel),
148         SiLU(),
149         nn.Dropout(p=dropout),
150         zero_module(
151             nn.Conv2d(self.out_channel, self.out_channel, 3, padding=1)
152         ),
153     )
154
155     if self.out_channel == channels:
156         self.skip_connection = nn.Identity()
157     elif use_conv:
158         self.skip_connection = nn.Conv2d(
159             channels, self.out_channel, 3, padding=1
160         )
161     else:
162         self.skip_connection = nn.Conv2d(channels, self.out_channel, 1)
163
164     def forward(self, x, emb):
165         """
166             Apply the block to a Tensor, conditioned on a embedding.
167             :param x: an [N x C x ...] Tensor of features.
168             :param emb: an [N x emb_channels] Tensor of embeddings.
169             :return: an [N x C x ...] Tensor of outputs.
170         """
171         return checkpoint(
172             self._forward, (x, emb), self.parameters(), self.use_checkpoint
173         )
174
175     def _forward(self, x, emb):
176         if self.updown:
177             in_rest, in_conv = self.in_layers[:-1], self.in_layers[-1]
```

```

178         h = in_rest(x)
179         h = self.h_upd(h)
180         x = self.x_upd(x)
181         h = in_conv(h)
182     else:
183         h = self.in_layers(x)
184         emb_out = self.emb_layers(emb).type(h.dtype)
185         while len(emb_out.shape) < len(h.shape):
186             emb_out = emb_out[..., None]
187         if self.use_scale_shift_norm:
188             out_norm, out_rest = self.out_layers[0], self.out_layers[1:]
189             scale, shift = torch.chunk(emb_out, 2, dim=1)
190             h = out_norm(h) * (1 + scale) + shift
191             h = out_rest(h)
192         else:
193             h = h + emb_out
194             h = self.out_layers(h)
195         return self.skip_connection(x) + h
196
197 class AttentionBlock(nn.Module):
198     """
199     An attention block that allows spatial positions to attend to each other.
200     Originally ported from here, but adapted to the N-d case.
201     https://github.com/hojonathanho/diffusion/blob/1
202     e0dceb3b3495bbe19116a5e1b3596cd0706c543/diffusion_tf/models/unet.py#L66.
203     """
204
205     def __init__(self,
206                  channels,
207                  num_heads=1,
208                  num_head_channels=-1,
209                  use_checkpoint=False,
210                  use_new_attention_order=False,
211                  ):
212         super().__init__()
213         self.channels = channels
214         if num_head_channels == -1:
215             self.num_heads = num_heads
216         else:
217             assert (
218                 channels % num_head_channels == 0
219             ), f"q,k,v channels {channels} is not divisible by num_head_channels {num_head_channels}"
220         self.num_heads = channels // num_head_channels
221         self.use_checkpoint = use_checkpoint
222         self.norm = normalization(channels)
223         self.qkv = nn.Conv1d(channels, channels * 3, 1)
224         if use_new_attention_order:
225             # split qkv before split heads
226             self.attention = QKVAttention(self.num_heads)
227         else:
228             # split heads before split qkv
229             self.attention = QKVAttentionLegacy(self.num_heads)
230
231         self.proj_out = zero_module(nn.Conv1d(channels, channels, 1))
232
233     def forward(self, x):
234         return checkpoint(self._forward, (x,), self.parameters(), True)
235

```

```
236     def _forward(self, x):
237         b, c, *spatial = x.shape
238         x = x.reshape(b, c, -1)
239         qkv = self.qkv(self.norm(x))
240         h = self.attention(qkv)
241         h = self.proj_out(h)
242         return (x + h).reshape(b, c, *spatial)
243
244
245 class QKVAAttentionLegacy(nn.Module):
246     """
247     A module which performs QKV attention. Matches legacy QKVAAttention + input/output
248     heads shaping
249     """
250
251     def __init__(self, n_heads):
252         super().__init__()
253         self.n_heads = n_heads
254
255     def forward(self, qkv):
256         """
257         Apply QKV attention.
258         :param qkv: an [N x (H * 3 * C) x T] tensor of Qs, Ks, and Vs.
259         :return: an [N x (H * C) x T] tensor after attention.
260         """
261         bs, width, length = qkv.shape
262         assert width % (3 * self.n_heads) == 0
263         ch = width // (3 * self.n_heads)
264         q, k, v = qkv.reshape(bs * self.n_heads, ch * 3, length).split(ch, dim=1)
265         scale = 1 / math.sqrt(math.sqrt(ch))
266         weight = torch.einsum(
267             "bct,bcs->bts", q * scale, k * scale
268         ) # More stable with f16 than dividing afterwards
269         weight = torch.softmax(weight.float(), dim=-1).type(weight.dtype)
270         a = torch.einsum("bts,bcs->bct", weight, v)
271         return a.reshape(bs, -1, length)
272
273     @staticmethod
274     def count_flops(model, _x, y):
275         return count_flops_attn(model, _x, y)
276
277 class QKVAAttention(nn.Module):
278     """
279     A module which performs QKV attention and splits in a different order.
280     """
281
282     def __init__(self, n_heads):
283         super().__init__()
284         self.n_heads = n_heads
285
286     def forward(self, qkv):
287         """
288         Apply QKV attention.
289         :param qkv: an [N x (3 * H * C) x T] tensor of Qs, Ks, and Vs.
290         :return: an [N x (H * C) x T] tensor after attention.
291         """
292         bs, width, length = qkv.shape
293         assert width % (3 * self.n_heads) == 0
294         ch = width // (3 * self.n_heads)
```

```

295     q, k, v = qkv.chunk(3, dim=1)
296     scale = 1 / math.sqrt(math.sqrt(ch))
297     weight = torch.einsum(
298         "bct,bcs->bts",
299         (q * scale).view(bs * self.n_heads, ch, length),
300         (k * scale).view(bs * self.n_heads, ch, length),
301     ) # More stable with f16 than dividing afterwards
302     weight = torch.softmax(weight.float(), dim=-1).type(weight.dtype)
303     a = torch.einsum("bts,bcs->bct", weight, v.reshape(bs * self.n_heads, ch,
304     length))
305     return a.reshape(bs, -1, length)
306
307     @staticmethod
308     def count_flops(model, _x, y):
309         return count_flops_attn(model, _x, y)
310
311     class UNet(nn.Module):
312         """
313             The full UNet model with attention and embedding.
314             :param in_channel: channels in the input Tensor, for image colorization :
315                 Y_channels + X_channels .
316             :param inner_channel: base channel count for the model.
317             :param out_channel: channels in the output Tensor.
318             :param res_blocks: number of residual blocks per downsample.
319             :param attn_res: a collection of downsample rates at which
320                 attention will take place. May be a set, list, or tuple.
321                 For example, if this contains 4, then at 4x downsampling, attention
322                 will be used.
323             :param dropout: the dropout probability.
324             :param channel_mults: channel multiplier for each level of the UNet.
325             :param conv_resample: if True, use learned convolutions for upsampling and
326                 downsampling.
327             :param use_checkpoint: use gradient checkpointing to reduce memory usage.
328             :param num_heads: the number of attention heads in each attention layer.
329             :param num_heads_channels: if specified, ignore num_heads and instead use
330                 a fixed channel width per attention head.
331             :param num_heads_upsample: works with num_heads to set a different number
332                 of heads for upsampling. Deprecated.
333             :param use_scale_shift_norm: use a FiLM-like conditioning mechanism.
334             :param resblock_updown: use residual blocks for up/downsampling.
335             :param use_new_attention_order: use a different attention pattern for potentially
336                 increased efficiency.
337         """
338
339         def __init__(
340             self,
341             image_size,
342             in_channel,
343             inner_channel,
344             out_channel,
345             res_blocks,
346             attn_res,
347             dropout=0,
348             channel_mults=(1, 2, 4, 8),
349             conv_resample=True,
350             use_checkpoint=False,
351             use_fp16=False,
352             num_heads=1,
353             num_head_channels=-1,
354             num_heads_upsample=-1,

```

```
353     use_scale_shift_norm=True,
354     resblock_updown=True,
355     use_new_attention_order=False,
356     ):
357
358     super().__init__()
359
360     if num_heads_upsample == -1:
361         num_heads_upsample = num_heads
362
363     self.image_size = image_size
364     self.in_channel = in_channel
365     self.inner_channel = inner_channel
366     self.out_channel = out_channel
367     self.res_blocks = res_blocks
368     self.attn_res = attn_res
369     self.dropout = dropout
370     self.channel_mults = channel_mults
371     self.conv_resample = conv_resample
372     self.use_checkpoint = use_checkpoint
373     self.dtype = torch.float16 if use_fp16 else torch.float32
374     self.num_heads = num_heads
375     self.num_head_channels = num_head_channels
376     self.num_heads_upsample = num_heads_upsample
377
378     cond_embed_dim = inner_channel * 4
379     self.cond_embed = nn.Sequential(
380         nn.Linear(inner_channel, cond_embed_dim),
381         SiLU(),
382         nn.Linear(cond_embed_dim, cond_embed_dim),
383     )
384
385     ch = input_ch = int(channel_mults[0] * inner_channel)
386     self.input_blocks = nn.ModuleList(
387         [EmbedSequential(nn.Conv2d(in_channel, ch, 3, padding=1))]
388     )
389     self._feature_size = ch
390     input_block_chans = [ch]
391     ds = 1
392     for level, mult in enumerate(channel_mults):
393         for _ in range(res_blocks):
394             layers = [
395                 ResBlock(
396                     ch,
397                     cond_embed_dim,
398                     dropout,
399                     out_channel=int(mult * inner_channel),
400                     use_checkpoint=use_checkpoint,
401                     use_scale_shift_norm=use_scale_shift_norm,
402                 )
403             ]
404             ch = int(mult * inner_channel)
405             if ds in attn_res:
406                 layers.append(
407                     AttentionBlock(
408                         ch,
409                         use_checkpoint=use_checkpoint,
410                         num_heads=num_heads,
411                         num_head_channels=num_head_channels,
412                         use_new_attention_order=use_new_attention_order,
```

```
413             )
414         )
415         self.input_blocks.append(EmbedSequential(*layers))
416         self._feature_size += ch
417         input_block_chans.append(ch)
418     if level != len(channel_mults) - 1:
419         out_ch = ch
420         self.input_blocks.append(
421             EmbedSequential(
422                 ResBlock(
423                     ch,
424                     cond_embed_dim,
425                     dropout,
426                     out_channel=out_ch,
427                     use_checkpoint=use_checkpoint,
428                     use_scale_shift_norm=use_scale_shift_norm,
429                     down=True,
430                 )
431                 if resblock_updown
432                 else Downsample(
433                     ch, conv_resample, out_channel=out_ch
434                 )
435             )
436         )
437         ch = out_ch
438         input_block_chans.append(ch)
439         ds *= 2
440         self._feature_size += ch
441
442     self.middle_block = EmbedSequential(
443         ResBlock(
444             ch,
445             cond_embed_dim,
446             dropout,
447             use_checkpoint=use_checkpoint,
448             use_scale_shift_norm=use_scale_shift_norm,
449         ),
450         AttentionBlock(
451             ch,
452             use_checkpoint=use_checkpoint,
453             num_heads=num_heads,
454             num_head_channels=num_head_channels,
455             use_new_attention_order=use_new_attention_order,
456         ),
457         ResBlock(
458             ch,
459             cond_embed_dim,
460             dropout,
461             use_checkpoint=use_checkpoint,
462             use_scale_shift_norm=use_scale_shift_norm,
463         ),
464     )
465     self._feature_size += ch
466
467     self.output_blocks = nn.ModuleList([])
468     for level, mult in list(enumerate(channel_mults))[:-1]:
469         for i in range(res_blocks + 1):
470             ich = input_block_chans.pop()
471             layers = [
472                 ResBlock(
```

```

473             ch + ich,
474             cond_embed_dim,
475             dropout,
476             out_channel=int(inner_channel * mult),
477             use_checkpoint=use_checkpoint,
478             use_scale_shift_norm=use_scale_shift_norm,
479         )
480     ]
481     ch = int(inner_channel * mult)
482     if ds in attn_res:
483         layers.append(
484             AttentionBlock(
485                 ch,
486                 use_checkpoint=use_checkpoint,
487                 num_heads=num_heads_upsample,
488                 num_head_channels=num_head_channels,
489                 use_new_attention_order=use_new_attention_order,
490             )
491         )
492     if level and i == res_blocks:
493         out_ch = ch
494         layers.append(
495             ResBlock(
496                 ch,
497                 cond_embed_dim,
498                 dropout,
499                 out_channel=out_ch,
500                 use_checkpoint=use_checkpoint,
501                 use_scale_shift_norm=use_scale_shift_norm,
502                 up=True,
503             )
504             if resblock_updown
505             else Upsample(ch, conv_resample, out_channel=out_ch)
506         )
507         ds /= 2
508     self.output_blocks.append(EmbedSequential(*layers))
509     self._feature_size += ch
510
511     self.out = nn.Sequential(
512         normalization(ch),
513         SiLU(),
514         zero_module(nn.Conv2d(input_ch, out_channel, 3, padding=1)),
515     )
516
517     def forward(self, x, gammas):
518         """
519             Apply the model to an input batch.
520             :param x: an [N x 2 x ...] Tensor of inputs (B&W)
521             :param gammas: a 1-D batch of gammas.
522             :return: an [N x C x ...] Tensor of outputs.
523         """
524         hs = []
525         gammas = gammas.view(-1, )
526         emb = self.cond_embed(gamma_embedding(gammas, self.inner_channel))
527
528         h = x.type(torch.float32)
529         for module in self.input_blocks:
530             h = module(h, emb)
531             hs.append(h)
532         h = self.middle_block(h, emb)

```

```
533     for module in self.output_blocks:  
534         h = torch.cat([h, hs.pop()], dim=1)  
535         h = module(h, emb)  
536         h = h.type(x.dtype)  
537     return self.out(h)
```

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.

```

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)
21
22 class InpaintingGaussianDiffusion(nn.Module):
23     def __init__(self, unet_config, beta_schedule, **kwargs):
24         super(InpaintingGaussianDiffusion, self).__init__(**kwargs)
25         self.denoise_fn = UNet(**unet_config)
26         self.beta_schedule = beta_schedule
27
28     def set_new_noise_schedule(self, device):
29         to_torch = partial(torch.tensor, dtype=torch.float32, device=device)
30         betas = make_beta_schedule(**self.beta_schedule)
31         alphas = 1. - betas
32         timesteps, = betas.shape
33         self.num_timesteps = int(timesteps)
34
35         gammas = np.cumprod(alphas, axis=0) # alphas_cumprod
36         gammas_prev = np.append(1., gammas[:-1])
37
38         # calculations for diffusion q(x_t | x_{t-1}) and others
39         self.register_buffer('gammas', to_torch(gammas))
40         self.register_buffer('sqrt_recip_gammas', to_torch(np.sqrt(1. / gammas)))
41         self.register_buffer('sqrt_recip1_gammas', to_torch(np.sqrt(1. / gammas - 1)))
42
43         # calculations for posterior q(x_{t-1} | x_t, x_0)
44         posterior_variance = betas * (1. - gammas_prev) / (1. - gammas)
45         # below: log calculation clipped because the posterior variance is 0 at the
46         # beginning of the diffusion chain
47         self.register_buffer('posterior_log_variance_clipped', to_torch(np.log(np.
48             maximum(posterior_variance, 1e-20))))
49         self.register_buffer('posterior_mean_coef1', to_torch(betas * np.sqrt(
50             gammas_prev) / (1. - gammas)))
51         self.register_buffer('posterior_mean_coef2', to_torch((1. - gammas_prev) * np.
52             sqrt(alphas) / (1. - gammas)))
53
54     def set_loss(self, loss_fn):

```

```

51         self.loss_fn = loss_fn
52
53     def predict_start_from_noise(self, y_t, t, noise):
54         return (
55             get_index_from_list(self.sqrt_recip_gammas, t, y_t.shape) * y_t -
56             get_index_from_list(self.sqrt_recipm1_gammas, t, y_t.shape) * noise
57         )
58
59     def q_posterior(self, y_0_hat, y_t, t):
60         """
61             Compute the mean and variance of the diffusion posterior:
62
63             q(x_{t-1} | x_t, x_0)
64
65             """
66             posterior_mean = (
67                 get_index_from_list(self.posterior_mean_coef1, t, y_t.shape) * y_0_hat +
68                 get_index_from_list(self.posterior_mean_coef2, t, y_t.shape) * y_t
69             )
70             posterior_log_variance_clipped = get_index_from_list(
71                 self.posterior_log_variance_clipped, t, y_t.shape
72             )
73             return posterior_mean, posterior_log_variance_clipped
74
75     def p_mean_variance(self, y_t, t, clip_denoised: bool, y_cond=None):
76         noise_level = get_index_from_list(self.gammas, t, x_shape=(1, 1)).to(y_t.
device)
77         y_0_hat = self.predict_start_from_noise(
78             y_t, t=t, noise=self.denoise_fn(torch.cat([y_cond, y_t], dim=1),
noise_level))
79
80         if clip_denoised:
81             y_0_hat.clamp_(-1., 1.)
82
83         model_mean, posterior_log_variance = self.q_posterior(
84             y_0_hat=y_0_hat, y_t=y_t, t=t)
85         return model_mean, posterior_log_variance
86
87     def q_sample(self, y_0, sample_gammas, noise=None):
88         noise = noise if noise is not None else torch.randn_like(y_0)
89         return (
90             sample_gammas.sqrt() * y_0 +
91             (1 - sample_gammas).sqrt() * noise
92         )
93
94         @torch.no_grad()
95     def p_sample(self, y_t, t, clip_denoised=True, y_cond=None):
96         model_mean, model_log_variance = self.p_mean_variance(
97             y_t=y_t, t=t, clip_denoised=clip_denoised, y_cond=y_cond)
98         noise = torch.randn_like(y_t) if any(t>0) else torch.zeros_like(y_t)
99         return model_mean + noise * (0.5 * model_log_variance).exp()
100
101        @torch.no_grad()
102    def restoration(self, y_cond, y_t=None, y_0=None, mask=None, sample_num=8):
103        b, *_ = y_cond.shape
104
105        sample_inter = (self.num_timesteps//sample_num)
106
107        y_t = y_t if y_t is not None else torch.randn_like(y_cond)
108        ret_arr = y_t

```

```

109     for i in reversed(range(0, self.num_timesteps)):
110         t = torch.full((b,), i, device=y_cond.device, dtype=torch.long)
111         y_t = self.p_sample(y_t, t, y_cond=y_cond)
112         if mask is not None:
113             y_t = y_0*(1.-mask) + mask*y_t
114         if i % sample_inter == 0:
115             ret_arr = torch.cat([ret_arr, y_t], dim=0)
116     return y_t, ret_arr
117
118 def forward(self, y_0, y_cond=None, mask=None, noise=None):
119     # sampling from p(gammas)
120     b, *_ = y_0.shape
121     t = torch.randint(1, self.num_timesteps, (b,), device=y_0.device).long()
122     gamma_t1 = get_index_from_list(self.gammas, t-1, x_shape=(1, 1))
123     sqrt_gamma_t2 = get_index_from_list(self.gammas, t, x_shape=(1, 1))
124     sample_gammas = (sqrt_gamma_t2-gamma_t1) * torch.rand((b, 1), device=y_0.
device) + gamma_t1
125     sample_gammas = sample_gammas.view(b, -1)
126
127     noise = noise if noise is not None else torch.randn_like(y_0)
128     y_noisy = self.q_sample(
129         y_0=y_0, sample_gammas=sample_gammas.view(-1, 1, 1, 1), noise=noise)
130
131     if mask is not None:
132         noise_hat = self.denoise_fn(torch.cat([y_cond, y_noisy*mask+(1.-mask)*y_0
], dim=1), sample_gammas)
133         loss = self.loss_fn(mask*noise, mask*noise_hat)
134     else:
135         noise_hat = self.denoise_fn(torch.cat([y_cond, y_noisy], dim=1),
sample_gammas)
136         loss = self.loss_fn(noise, noise_hat)
137     return loss

```

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
3 class Trainer():
4     def __init__(self, model, optimizers, train_loader, val_loader, epochs, sample_num
, device, save_model):
5         self.model = model.to(device)
6         self.optimizer = torch.optim.Adam(list(filter(
7             lambda p: p.requires_grad, self.model.parameters()
8         )), **optimizers)
9         self.model.set_loss(mse_loss)
10        self.model.set_new_noise_schedule(device)

```

```

11     self.sample_num = sample_num
12     self.train_loader = train_loader
13     self.val_loader = val_loader
14     self.device = device
15     self.epochs = epochs
16     self.save_model = save_model + "/best_model.pth"
17
18     def train_step(self):
19         self.model.train()
20         losses = []
21         for batch in tqdm(self.train_loader):
22             gt_image = batch['gt_image'].to(self.device)
23             cond_image = batch['cond_image'].to(self.device)
24             mask = batch['mask'].to(self.device)
25             mask_image = batch['mask_image'].to(self.device)
26             batch_size = len(batch['path'])
27
28             self.optimizer.zero_grad()
29
30             loss = self.model(gt_image, cond_image, mask=mask)
31             loss.backward()
32             losses.append(loss.item())
33             self.optimizer.step()
34         return sum(losses)/len(losses)
35
36     def val_step(self):
37         self.model.eval()
38         losses, metrics = [], []
39         with torch.no_grad():
40             for batch in tqdm(self.val_loader):
41                 gt_image = batch['gt_image'].to(self.device)
42                 cond_image = batch['cond_image'].to(self.device)
43                 mask = batch['mask'].to(self.device)
44                 mask_image = batch['mask_image'].to(self.device)
45                 loss = self.model(gt_image, cond_image, mask=mask)
46
47                 output, visuals = self.model.restoration(
48                     cond_image, y_t=cond_image, y_0=gt_image, mask=mask, sample_num=
49 self.sample_num)
50                 mae_score = mae(gt_image, output)
51
52                 losses.append(loss.item())
53                 metrics.append(mae_score.item())
54         return sum(losses)/len(losses), sum(metrics)/len(metrics)
55
56     def train(self):
57         best_mae = 100000
58         for epoch in range(self.epochs):
59             epoch_start_time = time.time()
60             train_loss = self.train_step()
61             val_loss, val_mae = self.val_step()
62             if val_mae < best_mae:
63                 torch.save(self.model.state_dict(), self.save_model)
64             # Print loss, acc end epoch
65             print("-" * 59)
66             print(
67                 "| End of epoch {:3d} | Time: {:.5.2f}s | Train Loss {:.8.3f} "
68                 "| Valid Loss {:.8.3f} | Valid MAE {:.8.3f} ".format(
69                     epoch+1, time.time() - epoch_start_time, train_loss, val_loss,
70                     val_mae

```

```

69
70
71     )
72     print("-" * 59)
73     self.model.load_state_dict(torch.load(self.save_model))
74
75 epochs = 200 # 5
76 sample_num = 8
77 save_model = './save_model'
78 optimizers = { "lr": 5e-5, "weight_decay": 0}
79 device = "cuda" if torch.cuda.is_available() else "cpu"
80
81 unet_config = {
82     "in_channel": 6,
83     "out_channel": 3,
84     "inner_channel": 64,
85     "channel_mults": [1, 2, 4, 8],
86     "attn_res": [16],
87     "num_head_channels": 32,
88     "res_blocks": 2,
89     "dropout": 0.2,
90     "image_size": 256
91 }
92
93 beta_schedule = {
94     "schedule": "linear",
95     "n_timestep": 20,
96     "linear_start": 1e-4,
97     "linear_end": 0.09
98 }
99 inpainting_model = InpaintingGaussianDiffusion(unet_config, beta_schedule)
100
101 trainer = Trainer(
102     inpainting_model, optimizers, train_loader, val_loader, epochs, sample_num, device
103     , save_model
104 )

```

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.

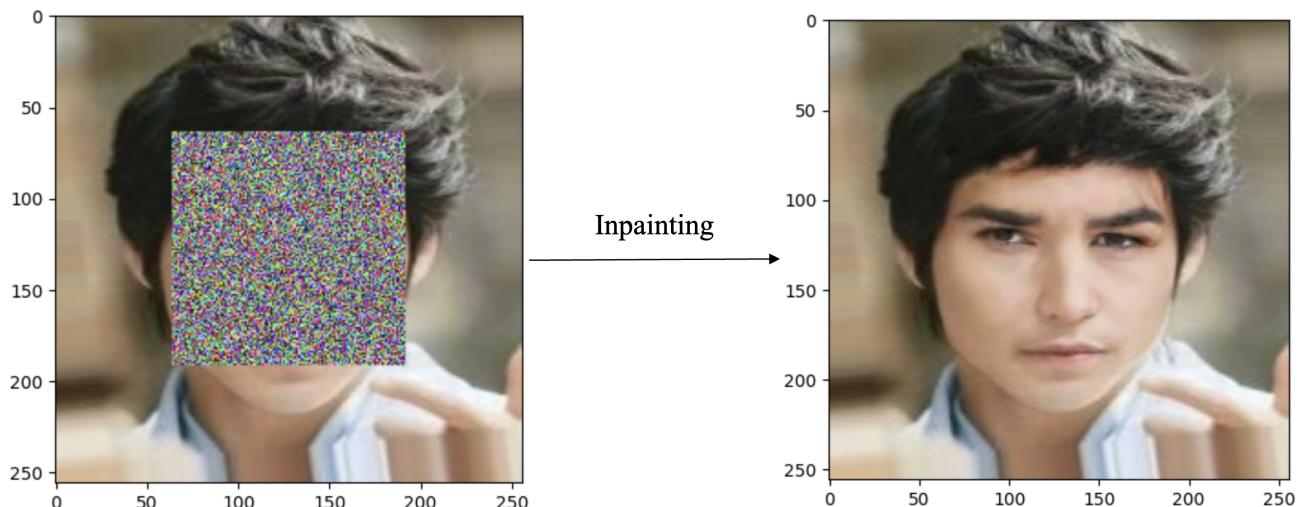
```

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(lambda t: (t + 1) / 2),
29         transforms.Lambda(lambda t: t.permute(1, 2, 0)), # CHW to HWC
30         transforms.Lambda(lambda t: t * 255.),
31         transforms.Lambda(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.

Phần 4. 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

- *Hết* -