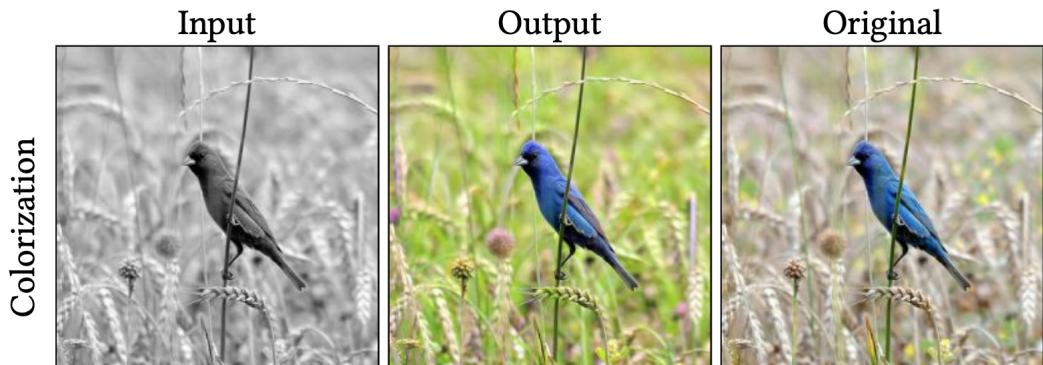


# Diffusion-based Image Colorization

Tien-Huy Nguyen, Khanh Duong and Nhu-Tai Do

Ngày 7 tháng 4 năm 2024

## Phần I: Giới thiệu



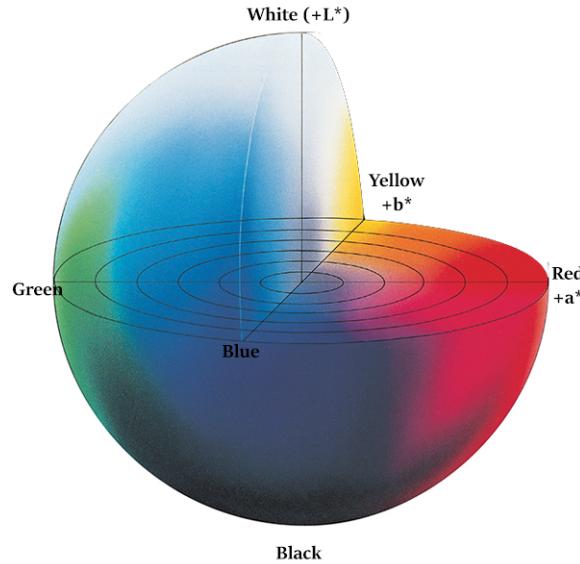
Hình 1: Ví dụ minh họa cho bài toán Image Colorization.

Sự bùng nổ của mô hình **Diffusion** trong những năm gần đây đã tạo ra nhiều bước ngoặt trong vấn đề sinh dữ liệu, đặc biệt là vấn đề tái tạo ảnh. Theo đó, ở quá trình forward, Diffusion thêm nhiễu vào ảnh một cách có hệ thống, biến ảnh thành một nhiễu tuân theo phân phối Gauss. Sau đó, ở quá trình ngược lại, mô hình này học cách dự đoán nhiễu và tiến hành khử nhiễu dần dần từ một nhiễu chuẩn để tạo ra ảnh mới.

Trong bài báo mang tên [Palette: Image-to-Image Diffusion Models](#), mô hình Diffusion đã thực sự chứng minh được khả năng tổng quát mạnh mẽ của mình, khi cùng lúc giải quyết 4 vấn đề khó trong lĩnh vực dịch ảnh chỉ với một mô hình duy nhất, bao gồm: Colorization, Inpainting, Uncropping, và JPEG restoration.

Trong dự án này, chúng ta sẽ cùng tìm hiểu ứng dụng của mô hình Diffusion trong bài toán tô màu ảnh, bằng cách sử dụng những ý tưởng cơ bản của công bố nêu trên.

**Image Colorization** là quá trình dự đoán màu cho các ảnh đen trắng, giúp tái tạo lại hình ảnh thực tế từ dữ liệu đơn sắc, mang lại trải nghiệm hình ảnh phong phú và sống động. Với đầu vào là một ảnh xám, biểu thị cường độ sáng của ảnh, mô hình sẽ học cách ước tính các kênh màu của ảnh, tạo ra một hình ảnh hợp lý và hài hòa về mặt thị giác.



Hình 2: Không gian màu *Lab*.

Trong dự án này, chúng ta sẽ tiếp tục sử dụng không gian màu *Lab* cho xử lý dữ liệu. Theo đó, mô hình của chúng ta sẽ nhận đầu vào là kênh *L* như tám ảnh gray-scale, đại diện cho độ sáng, và sử dụng kênh *ab* như ground truth của mô hình.

Đối với vấn đề tô màu cho ảnh, có một thuật ngữ được gọi là *Multimodal Predictions*. Thuật ngữ này đề cập đến việc một pixel có thể có nhiều kết quả màu hợp lý thay vì chỉ một dự đoán nhất định. Điều này dẫn đến việc, để đánh giá một mô hình Image Colorization, ta không chỉ sử dụng thước đo định lượng, mà cần phải xem xét đến cảm nhận thị giác và những kết quả mang tính định tính khác.

Một thước đo định tính đã và đang được sử dụng để giải quyết vấn đề này được gọi là *colorization Turing test* (hay *Fool rate*). Theo đó, người tham gia sẽ cần phải phân biệt những tám ảnh gốc và những tám ảnh giả được tạo ra từ mô hình AI. Tỷ lệ người tham gia bị đánh lừa bởi mức độ chân thật của ảnh giả càng cao, chứng tỏ mô hình thu được càng tốt.

Chính vì vậy, cần thiết phải có một công cụ giúp người huấn luyện có thể quan sát một cách trực quan những kết quả hình ảnh được tạo ra trong quá trình huấn luyện một mô hình Image Colorization. Thật may mắn, **Weights & Biases** (wandb), một công cụ cho phép theo dõi và hiển thị kết quả số liệu, hình ảnh, đã và đang ngày càng phổ biến và dễ sử dụng.

Trong phần này, chúng ta sẽ tập trung vào việc xây dựng một mô hình dựa trên Diffusion để giải quyết vấn đề Image Colorization, cùng với đó là áp dụng công cụ *wandb* để hỗ trợ quá trình huấn luyện. Input và output của chương trình như sau:

1. **Input:** Ảnh xám G (L channel).
2. **Output:** Trưởng ảnh màu C (ab channels).

## Phần II: Nội dung

Trong phần này, chúng ta sẽ triển khai mô hình Diffusion-based Image Colorization dựa trên ý tưởng cơ bản của bài báo [Palette: Image-to-Image Diffusion Models](#) để học cách biến đổi một hình ảnh xám đầu vào thành một hình ảnh màu hợp lý. Cụ thể, ta sẽ xây dựng chương trình dựa trên bộ dữ liệu CelebA ([Large-scale CelebFaces Attributes](#)), một tập dữ liệu lớn được sử dụng rộng rãi trong lĩnh vực nhận dạng khuôn mặt và phân loại hình ảnh, chứa hơn 200,000 hình ảnh của nhiều người nổi tiếng từ các bộ phim, truyền hình và âm nhạc.



Hình 3: Ảnh minh họa cho CelebA dataset

Theo đó, nội dung thực nghiệm sẽ trình bày với các thành phần như sau:

- Data Preparation: Chuẩn bị dữ liệu cho tập huấn luyện.
- Models: Xây dựng mô hình UNet và mô hình Diffusion cho Colorization.
- Loss, Metrics: Xây dựng hàm mất mát và độ đo đánh giá cho mô hình.
- Trainer: Xây dựng class Trainer dành riêng cho huấn luyện
- Inference: Minh họa kết quả đạt được sau khi huấn luyện mô hình.

### 1. Data Preparation

Đầu tiên, chúng ta cần chuẩn bị bộ dữ liệu CelebA. Bạn có thể tải bộ dữ liệu và giải nén tại [đây](#).

Khai báo các thư viện:

```

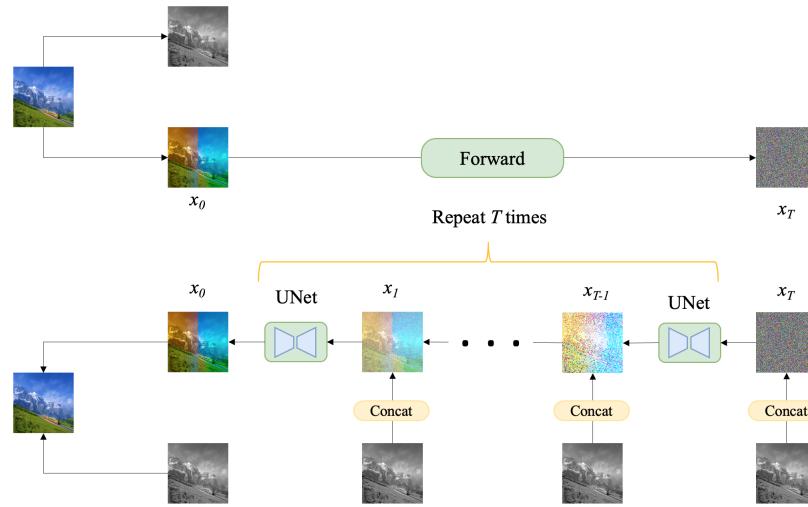
1 import glob
2 import torch
3 import cv2
4 import numpy as np
5 from torchvision import transforms
6 from torch.utils.data import Dataset
7 from torch.utils.data import DataLoader
8
9 # Load the paths of the images and split into train set and valid set
10 img_paths = glob.glob('./img_align_celeba/*.jpg')
11 num_train, num_val = 200, 20 # demo with small data
12 train_imgpaths = img_paths[:num_train]
13 val_imgpaths = img_paths[num_train : num_train + num_val]
14
15

```

```
16 # Build ColorDataset
17 class ColorDataset():
18     def __init__(self, img_paths, data_len=2880, image_size=(128, 128)):
19         if data_len > 0:
20             self.img_paths = img_paths[:int(data_len)]
21         else:
22             self.img_paths = img_paths
23         self.tfs = transforms.Resize((image_size[0], image_size[1]))
24
25     def __getitem__(self, index):
26         img_path = self.img_paths[index]
27         arr_img_bgr = cv2.imread(img_path)
28
29         # Convert BGR to LAB
30         arr_img_lab = cv2.cvtColor(arr_img_bgr, cv2.COLOR_BGR2LAB)
31
32         # Normalize from [0..255] to [-1..1]
33         arr_img_lab = ((arr_img_lab * 2.0) / 255.0) - 1.0
34
35         # Resize the image to image_size=(128, 128)
36         tens_img_lab = torch.tensor(arr_img_lab.transpose(2, 0, 1),
37                                     dtype=torch.float32)
38
39         # Divide the image into gray input and color output
40         original_img_l = tens_img_lab[:1, :, :]
41         tens_img_lab = self.tfs(tens_img_lab)
42         tens_img_l = tens_img_lab[:1, :, :]
43         tens_img_ab = tens_img_lab[1:, :, :]
44         return original_img_l, tens_img_l, tens_img_ab
45
46     def __len__(self):
47         return len(self.img_paths)
48
49 # Create Dataset
50 train_dataset = ColorDataset(train_imgpaths, num_train)
51 val_dataset = ColorDataset(val_imgpaths, num_val)
52
53 # Create DataLoader
54 BATCH_SIZE = 4
55 train_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE,
56                           shuffle=True, drop_last=True)
57 val_loader = DataLoader(val_dataset, batch_size=BATCH_SIZE)
```

## 2. Models

Trong phần này, chúng ta sẽ tận dụng một kiến trúc UNet phổ biến trong các bài toán liên quan đến Diffusion, sử dụng nhiều kỹ thuật như Attention Mechanism, Adaptive Group Normalization và được giới thiệu trong một công bố mang tên [Diffusion Models Beat GAN on Image Synthesis](#).



Hình 4: Minh họa quá trình đảo ngược của mô hình Color Diffusion

Ở quá trình đảo ngược, mô hình UNet nhận đầu vào bao gồm: timestep embedding  $t$  và một ảnh 3 chiều (được hợp thành bởi đầu ra của mô hình UNet tại thời điểm  $t + 1$  và thành phần điều kiện là kênh màu xám). Sau đó, mô hình học cách dự đoán và trả về một nhiễu Gauss cho timestep  $t - 1$ . Kết quả này sau đó được kết hợp với trường màu ab tại timestep  $t$  để tính toán ra  $ab$  channels tại timestep  $t - 1$ . Quá trình này được lặp lại cho đến khi timestep  $t = 0$ . Lúc này ta nhận được trường màu  $ab$  đã được khử nhiễu. Kết hợp với kênh màu xám, ta thu được một ảnh màu hoàn thiện.

### UNet model:

```

1 import math
2 import numpy as np
3 import torch
4 import torch.nn as nn
5
6
7 class GroupNorm32(nn.GroupNorm):
8     def forward(self, x):
9         return super().forward(x.float()).type(x.dtype)
10
11
12 def zero_module(module):
13     """
14     Zero out the parameters of a module and return it.
15     """
16     for p in module.parameters():
17         p.detach().zero_()
18     return module
19
20
21 def scale_module(module, scale):
22     """
23     Scale the parameters of a module and return it.
24     """
25     for p in module.parameters():
26         p.detach().mul_(scale)

```

```
27     return module
28
29
30 def mean_flat(tensor):
31     """
32     Take the mean over all non-batch dimensions.
33     """
34     return tensor.mean(dim=list(range(1, len(tensor.shape))))
35
36
37 def normalization(channels):
38     """
39     Make a standard normalization layer.
40
41     :param channels: number of input channels.
42     :return: an nn.Module for normalization.
43     """
44     return GroupNorm32(32, channels)
45
46
47
48 def checkpoint(func, inputs, params, flag):
49     """
50     Evaluate a function without caching intermediate activations, allowing for
51     reduced memory at the expense of extra compute in the backward pass.
52
53     :param func: the function to evaluate.
54     :param inputs: the argument sequence to pass to 'func'.
55     :param params: a sequence of parameters 'func' depends on but does not
56                     explicitly take as arguments.
57     :param flag: if False, disable gradient checkpointing.
58     """
59     if flag:
60         args = tuple(inputs) + tuple(params)
61         return CheckpointFunction.apply(func, len(inputs), *args)
62     else:
63         return func(*inputs)
64
65
66 class CheckpointFunction(torch.autograd.Function):
67     @staticmethod
68     def forward(ctx, run_function, length, *args):
69         ctx.run_function = run_function
70         ctx.input_tensors = list(args[:length])
71         ctx.input_params = list(args[length:])
72         with torch.no_grad():
73             output_tensors = ctx.run_function(*ctx.input_tensors)
74         return output_tensors
75
76     @staticmethod
77     def backward(ctx, *output_grads):
78         ctx.input_tensors = [x.detach().requires_grad_(True) for x in ctx.
79         input_tensors]
80         with torch.enable_grad():
81             # Fixes a bug where the first op in run_function modifies the
82             # Tensor storage in place, which is not allowed for detach()'d
83             # Tensors.
84             shallow_copies = [x.view_as(y) for x in ctx.input_tensors]
85             output_tensors = ctx.run_function(*shallow_copies)
86             input_grads = torch.autograd.grad(
```

```

86         output_tensors ,
87         ctx.input_tensors + ctx.input_params ,
88         output_grads ,
89         allow_unused=True ,
90     )
91     del ctx.input_tensors
92     del ctx.input_params
93     del output_tensors
94     return (None, None) + input_grads
95
96
97 def count_flops_attn(model, _x, y):
98     """
99     A counter for the 'thop' package to count the operations in an
100    attention operation.
101    Meant to be used like:
102        macs, params = thop.profile(
103            model,
104            inputs=(inputs, timestamps),
105            custom_ops={QKVAttention: QKVAttention.count_flops},
106        )
107    """
108    b, c, *spatial = y[0].shape
109    num_spatial = int(np.prod(spatial))
110    # We perform two matmuls with the same number of ops.
111    # The first computes the weight matrix, the second computes
112    # the combination of the value vectors.
113    matmul_ops = 2 * b * (num_spatial ** 2) * c
114    model.total_ops += torch.DoubleTensor([matmul_ops])
115
116
117 def gamma_embedding(gammas, dim, max_period=10000):
118     """
119     Create sinusoidal timestep embeddings.
120     :param gammas: a 1-D Tensor of N indices, one per batch element.
121             These may be fractional.
122     :param dim: the dimension of the output.
123     :param max_period: controls the minimum frequency of the embeddings.
124     :return: an [N x dim] Tensor of positional embeddings.
125     """
126     half = dim // 2
127     freqs = torch.exp(
128         -math.log(max_period) * torch.arange(start=0, end=half, dtype=torch.
129         float32) / half
130     ).to(device=gammas.device)
131     args = gammas[:, None].float() * freqs[None]
132     embedding = torch.cat([torch.cos(args), torch.sin(args)], dim=-1)
133     if dim % 2:
134         embedding = torch.cat([embedding, torch.zeros_like(embedding[:, :1])], dim=-1)
135     return embedding
136
137
138 import math
139 import torch
140 import torch.nn as nn
141 import torch.nn.functional as F
142
143 from abc import abstractmethod
144
145 class SiLU(nn.Module):
146     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__(self,
102                 channels,
103                 emb_channels,
104                 dropout,
105                 out_channel=None,
106                 use_conv=False,
107                 use_scale_shift_norm=False,
108                 use_checkpoint=False,
109                 up=False,
110                 down=False,
111                 ):
112        super().__init__()
113        self.channels = channels
114        self.emb_channels = emb_channels
115        self.dropout = dropout
116        self.out_channel = out_channel or channels
117        self.use_conv = use_conv
118        self.use_checkpoint = use_checkpoint
119        self.use_scale_shift_norm = use_scale_shift_norm
120
121        self.in_layers = nn.Sequential(
122            normalization(channels),
123            SiLU(),
124            nn.Conv2d(channels, self.out_channel, 3, padding=1),
125        )
126
127        self.updown = up or down
128
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.
144             out_channel,
145             ),
146     )
147     self.out_layers = nn.Sequential(
148         normalization(self.out_channel),
149         SiLU(),
150         nn.Dropout(p=dropout),
151         zero_module(
152             nn.Conv2d(self.out_channel, self.out_channel, 3, padding=1)
153             ),
154     )
155
156     if self.out_channel == channels:
157         self.skip_connection = nn.Identity()
158     elif use_conv:
159         self.skip_connection = nn.Conv2d(
160             channels, self.out_channel, 3, padding=1
161             )
162     else:
163         self.skip_connection = nn.Conv2d(channels, self.out_channel, 1)
164
165     def forward(self, x, emb):
166         """
167             Apply the block to a Tensor, conditioned on a embedding.
168             :param x: an [N x C x ...] Tensor of features.
169             :param emb: an [N x emb_channels] Tensor of embeddings.
170             :return: an [N x C x ...] Tensor of outputs.
171         """
172
173         return checkpoint(
174             self._forward, (x, emb), self.parameters(), self.use_checkpoint
175         )
176
177     def _forward(self, x, emb):
178         if self.updown:
179             in_rest, in_conv = self.in_layers[:-1], self.in_layers[-1]
180             h = in_rest(x)
181             h = self.h_upd(h)
182             x = self.x_upd(x)
183             h = in_conv(h)
184         else:
185             h = self.in_layers(x)
186             emb_out = self.emb_layers(emb).type(h.dtype)
187             while len(emb_out.shape) < len(h.shape):
188                 emb_out = emb_out[..., None]
189             if self.use_scale_shift_norm:
190                 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__(
206         self,
207         channels,
208         num_heads=1,
209         num_head_channels=-1,
210         use_checkpoint=False,
211         use_new_attention_order=False,
212     ):
213         super().__init__()
214         self.channels = channels
215         if num_head_channels == -1:
216             self.num_heads = num_heads
217         else:
218             assert (
219                 channels % num_head_channels == 0
220             ), f"q,k,v channels {channels} is not divisible by num_head_channels {num_head_channels}"
221             self.num_heads = channels // num_head_channels
222         self.use_checkpoint = use_checkpoint
223         self.norm = normalization(channels)
224         self.qkv = nn.Conv1d(channels, channels * 3, 1)
225         if use_new_attention_order:
226             # split qkv before split heads
227             self.attention = QKVAttention(self.num_heads)
228         else:
229             # split heads before split qkv
230             self.attention = QKVAttentionLegacy(self.num_heads)
231
232         self.proj_out = zero_module(nn.Conv1d(channels, channels, 1))
233
234     def forward(self, x):
235         return checkpoint(self._forward, (x,), self.parameters(), True)
236
237     def _forward(self, x):
238         b, c, *spatial = x.shape
239         x = x.reshape(b, c, -1)
240         qkv = self.qkv(self.norm(x))
241         h = self.attention(qkv)
242         h = self.proj_out(h)
243         return (x + h).reshape(b, c, *spatial)
244
245 class QKVAttentionLegacy(nn.Module):
246     """

```

```
247     A module which performs QKV attention. Matches legacy QKVAttention + input/
248     ouput 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 QKVAttention(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)
```

```
305
306     @staticmethod
307     def count_flops(model, _x, y):
308         return count_flops_attn(model, _x, y)
309
310 class UNet(nn.Module):
311     """
312         The full UNet model with attention and embedding.
313         :param in_channel: channels in the input Tensor, for image colorization :
314             Y_channels + X_channels .
315         :param inner_channel: base channel count for the model.
316         :param out_channel: channels in the output Tensor.
317         :param res_blocks: number of residual blocks per downsample.
318         :param attn_res: a collection of downsample rates at which
319             attention will take place. May be a set, list, or tuple.
320             For example, if this contains 4, then at 4x downsampling, attention
321             will be used.
322         :param dropout: the dropout probability.
323         :param channel_mults: channel multiplier for each level of the UNet.
324         :param conv_resample: if True, use learned convolutions for upsampling and
325             downsampling.
326         :param use_checkpoint: use gradient checkpointing to reduce memory usage.
327         :param num_heads: the number of attention heads in each attention layer.
328         :param num_heads_channels: if specified, ignore num_heads and instead use
329             a fixed channel width per attention head.
330         :param num_heads_upsample: works with num_heads to set a different number
331             of heads for upsampling. Deprecated.
332         :param use_scale_shift_norm: use a FiLM-like conditioning mechanism.
333         :param resblock_updown: use residual blocks for up/downsampling.
334         :param use_new_attention_order: use a different attention pattern for
335             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,
355         use_scale_shift_norm=True,
356         resblock_updown=True,
357         use_new_attention_order=False,
358     ):
359
360         super().__init__()
361
362         if num_heads_upsample == -1:
363             num_heads_upsample = num_heads
```

```
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     ch = out_ch
437     input_block_chans.append(ch)
438     ds *= 2
439     self._feature_size += ch
440
441     self.middle_block = EmbedSequential(
442         ResBlock(
443             ch,
444             cond_embed_dim,
445             dropout,
446             use_checkpoint=use_checkpoint,
447             use_scale_shift_norm=use_scale_shift_norm,
448         ),
449         AttentionBlock(
450             ch,
451             use_checkpoint=use_checkpoint,
452             num_heads=num_heads,
453             num_head_channels=num_head_channels,
454             use_new_attention_order=use_new_attention_order,
455         ),
456         ResBlock(
457             ch,
458             cond_embed_dim,
459             dropout,
460             use_checkpoint=use_checkpoint,
461             use_scale_shift_norm=use_scale_shift_norm,
462         ),
463     ),
464     self._feature_size += ch
465
466     self.output_blocks = nn.ModuleList([])
467     for level, mult in list(enumerate(channel_mults))[:-1]:
468         for i in range(res_blocks + 1):
469             ich = input_block_chans.pop()
470             layers = [
471                 ResBlock(
472                     ch + ich,
473                     cond_embed_dim,
474                     dropout,
475                     out_channel=int(inner_channel * mult),
476                     use_checkpoint=use_checkpoint,
477                     use_scale_shift_norm=use_scale_shift_norm,
478                 )
479             ]
480             ch = int(inner_channel * mult)
481             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)

```

## Color Diffusion Model

```

1 def make_beta_schedule(schedule, n_timestep, linear_start=1e-5, linear_end=1e-2):
2     if schedule == 'linear':
3         betas = np.linspace(

```

```

4         linear_start, linear_end, n_timestep, dtype=np.float64
5     )
6     else:
7         raise NotImplementedError(schedule)
8     return betas
9
10 def get_index_from_list(vals, t, x_shape=(1,1,1,1)):
11     """
12     Returns a specific index t of a passed list of values vals
13     while considering the batch dimension.
14     """
15     batch_size, *_ = t.shape
16     out = vals.gather(-1, t)
17     return out.reshape(batch_size, *((1,) * (len(x_shape) - 1))).to(device)
18
19 from tqdm import tqdm
20 from functools import partial
21
22 class ColorDiffusion(nn.Module):
23     def __init__(self, unet_config, beta_schedule, **kwargs):
24         super(ColorDiffusion, 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_recipm1_gammas', to_torch(np.sqrt(1. / gammas
42             - 1)))
43
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
46         # the 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
52             * np.sqrt(alphas) / (1. - gammas)))
53
53     def set_loss(self, loss_fn):
54         self.loss_fn = loss_fn
55
56     def predict_start_from_noise(self, y_t, t, noise):
57         return (
58             get_index_from_list(self.sqrt_recip_gammas, t, y_t.shape) * y_t -
59             get_index_from_list(self.sqrt_recipm1_gammas, t, y_t.shape) * noise
60         )
61
62     def q_posterior(self, y_0_hat, y_t, t):
63
64

```



```

99         # sampling from p(gamma)
100        b, *_ = y_0.shape
101        t = torch.randint(1, self.num_timesteps, (b,), device=y_0.device).long()
102        gamma_t1 = get_index_from_list(self.gammas, t-1, x_shape=(1, 1))
103        sqrt_gamma_t2 = get_index_from_list(self.gammas, t, x_shape=(1, 1))
104        sample_gammas = (sqrt_gamma_t2-gamma_t1) * torch.rand((b, 1), device=y_0.
device) + gamma_t1
105        sample_gammas = sample_gammas.view(b, -1)
106
107        noise = noise if noise is not None else torch.randn_like(y_0)
108        y_noisy = self.q_sample(
109            y_0=y_0, sample_gammas=sample_gammas.view(-1, 1, 1, 1), noise=noise)
110
111        noise_hat = self.denoise_fn(torch.cat([y_cond, y_noisy], dim=1),
112        sample_gammas)
113        loss = self.loss_fn(noise, noise_hat)
114        return loss
115
116
117 unet_config = {
118     "in_channel": 3,
119     "out_channel": 2,
120     "inner_channel": 64,
121     "channel_mults": [1, 2, 4, 8],
122     "attn_res": [16],
123     "num_head_channels": 32,
124     "res_blocks": 2,
125     "dropout": 0.2,
126     "image_size": 128
127 }
128
129 beta_schedule = {
130     "schedule": "linear",
131     "n_timestep": 20,
132     "linear_start": 1e-4,
133     "linear_end": 0.09
134 }
135
136 colordiff_model = ColorDiffusion(unet_config, beta_schedule)

```

### 3. Loss and Metrics

Trong phần này, chúng ta thiết lập hàm mất mát cho mô hình Color Diffusion. Trong đó,  $L2$  norm, còn được biết đến là *Mean Square Error*, được dùng làm hàm mất mát do sự đa dạng đáng kể về mặt kết quả so với *Mean Absolute Error* ( $L1$  norm), được kiểm chứng qua quá trình thực nghiệm bởi [Saharia, C.](#) và các cộng sự. Ngoài ra, *MAE* cũng đóng vai trò như một độ đo để đánh giá mô hình trong dự án này.

```

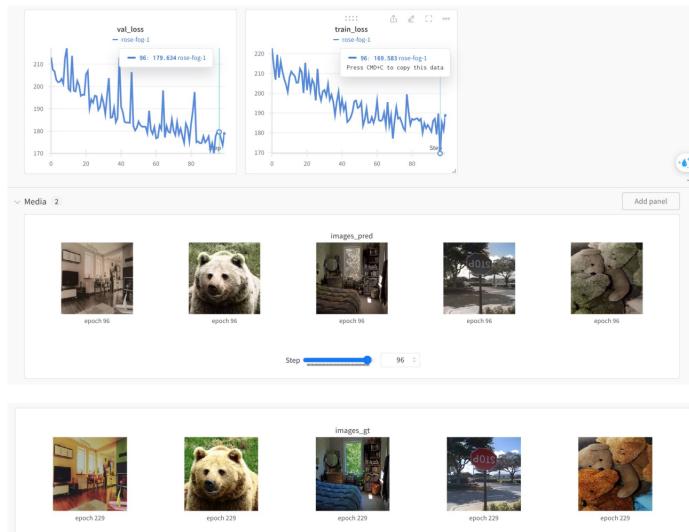
1 import torch.nn.functional as F
2
3 def mse_loss(output, target):
4     return F.mse_loss(output, target)
5
6
7 def mae(input, target):
8     with torch.no_grad():
9         loss = nn.L1Loss()
10        output = loss(input, target)
11        return output

```

#### 4. Trainer

Ở giai đoạn huấn luyện, chúng ta sẽ xây dựng class Trainer dành cho việc huấn luyện mô hình Diffusion. Ngoài ra, chúng ta cũng sẽ sử dụng một công cụ theo dõi được sử dụng phổ biến trong việc huấn luyện các mô hình học máy, được gọi là **wandb**. Theo đó, các kết quả bao gồm cả thông số mất mát của mô hình và hình ảnh mà mô hình dự đoán cũng có thể được trực quan hóa, giúp người dùng dễ dàng kiểm soát quy trình huấn luyện, từ đó đưa ra những chiến lược thích hợp và kịp thời.

Để có thể sử dụng wandb, bạn có thể đăng ký và nhận API cá nhân thông qua [wandb.ai](https://wandb.ai).



Hình 5: Ảnh minh họa cho việc sử dụng **wandb**.

Khởi tạo class Trainer

```

1 import time
2
3 class Trainer():
4     def __init__(self, model, optimizers, train_loader,
5                  val_loader, epochs, sample_num,
6                  device, save_model, use_wandb=False):
7
8         self.model = model.to(device)
9         self.optimizer = torch.optim.Adam(list(filter(
10             lambda p: p.requires_grad, self.model.parameters()
11         )), **optimizers)
12         self.model.set_loss(mse_loss)
13         self.model.set_new_noise_schedule(device)
14         self.sample_num = sample_num
15         self.train_loader = train_loader
16         self.val_loader = val_loader
17         self.device = device
18         self.epochs = epochs
19         self.save_model = save_model
20         self.use_wandb = use_wandb
21
22     def train_step(self):
23         self.model.train()
24         losses = []

```

```

25         for original_gray, gray, color in tqdm(self.train_loader):
26             cond_gray = gray.to(self.device)
27             gt_color = color.to(self.device)
28
29             self.optimizer.zero_grad()
30
31             loss = self.model(gt_color, cond_gray)
32             loss.backward()
33             losses.append(loss.item())
34             self.optimizer.step()
35         return sum(losses)/len(losses)
36
37     def val_step(self, epoch):
38         self.model.eval()
39         losses, metrics = [], []
40         pred_images = []
41         gt_images = []
42
43         with torch.no_grad():
44             for i, (original_gray, gray, color) in tqdm(enumerate(self.val_loader
45             )):
46                 cond_gray = gray.to(self.device)
47                 gt_color = color.to(self.device)
48                 loss = self.model(gt_color, cond_gray)
49
50                 output, visuals = self.model.restoration(
51                     cond_gray, sample_num=self.sample_num)
52                 if i == 0:
53                     for i in range(output.shape[0]):
54                         # Show predicted image
55                         pred_bgr_image = self.show_wandb_image(original_gray[i],
56                         output[i].detach().cpu())
57                         pred_wandb_image = wandb.Image(pred_bgr_image, caption=f"\"
58                         epoch {epoch}\")"
59                         pred_images.append(pred_wandb_image)
60
61                         # Show ground truth image
62                         gt_bgr_image = self.show_wandb_image(original_gray[i],
63                         color[i])
64                         gt_wandb_image = wandb.Image(gt_bgr_image, caption=f"\"
65                         epoch {epoch}\")"
66                         gt_images.append(gt_wandb_image)
67
68                         mae_score = mae(gt_color, output)
69                         losses.append(loss.item())
70                         metrics.append(mae_score.item())
71             return sum(losses)/len(losses), sum(metrics)/len(metrics), pred_images,
72             gt_images
73
74     # Postprocess the image before logging to WandB
75     def show_wandb_image(self, img_l, img_ab, is_save=False):
76         img_l = img_l.permute(1, 2, 0).numpy()
77         img_ab = img_ab.permute(1, 2, 0).numpy()
78         img_ab = cv2.resize(img_ab, (img_l.shape[1], img_l.shape[0]),
79             interpolation=cv2.INTER_LINEAR)
80         arr_lab = np.concatenate([img_l, img_ab], axis=2)
81         arr_lab = (arr_lab + 1.0) * 255 / 2
82         arr_lab = np.clip(arr_lab, 0, 255).astype(np.uint8)
83         arr_bgr = cv2.cvtColor(arr_lab, cv2.COLOR_LAB2RGB)
84         return arr_bgr

```

```

78
79
80     def train(self):
81         best_mae = 100000
82         for epoch in range(self.epochs):
83             epoch_start_time = time.time()
84             train_loss = self.train_step()
85             val_loss, val_mae, pred_images, gt_images = self.val_step(epoch)
86
87             # Log the results to WandB
88             if self.use_wandb:
89                 wandb.log({
90                 "train_loss": train_loss,
91                 "val_loss": val_loss,
92                 "val_mae": val_mae,
93                 "pred_images": pred_images,
94                 "gt_images": gt_images
95             })
96
97             if val_mae < best_mae:
98                 torch.save(self.model.state_dict(), self.save_model)
99             # Print loss, acc end epoch
100            print("-" * 59)
101            print(
102                "| End of epoch {:3d} | Time: {:.5.2f}s | Train Loss {:.8.3f} "
103                "| Valid Loss {:.8.3f} | Valid MAE {:.8.3f} ".format(
104                    epoch+1, time.time() - epoch_start_time,
105                    train_loss, val_loss, val_mae
106                )
107            )
108            print("-" * 59)
109            self.model.load_state_dict(torch.load(self.save_model))

```

Tải thư viện **wandb**.

```

1 !pip install wandb
2 import wandb

```

Khởi tạo các tham số và Trainer object.

```

1 epochs = 3
2 sample_num = 8
3 save_model = './save_model/best_model.pth'
4 optimizers = { "lr": 5e-5, "weight_decay": 0}
5 device = "cuda" if torch.cuda.is_available() else "cpu"
6 use_wandb = True
7
8 trainer = Trainer(
9     colordiff_model, optimizers,
10    train_loader, val_loader,
11    epochs, sample_num,
12    device, save_model,
13    use_wandb
14 )

```

Đăng nhập **wandb** thông qua API được lấy từ tài khoản **wandb** cá nhân.

```

1 if use_wandb:
2     wandb_api="387da1f220b55f23dec29347d30650c011d7exxx"
3     wandb.login(key=wandb_api)

```

Khởi tạo một **wandb** session.

```

1 if use_wandb:
2     wandb.init(
3         # set the wandb project where this run will be logged
4         project="my-diff-color",
5
6         # track hyperparameters and run metadata
7         config={
8             "learning_rate": optimizers["lr"],
9             "weight_decay": optimizers["weight_decay"],
10            "architecture": "UNet",
11            "dataset": "CelebA",
12            "epochs": epochs,
13            "save_model": save_model,
14            "sample_num": sample_num
15        }
16    )

```

Tiến hành huấn luyện. Lúc này, nếu **wandb** được sử dụng thông qua việc cài đặt tham số **use\_wandb = True**, các thông tin về loss, metrics và hình ảnh sẽ được đẩy lên giao diện **wandb**, giúp người dùng có thể dễ dàng theo dõi và đánh giá tiến độ huấn luyện.

```
1 trainer.train()
```

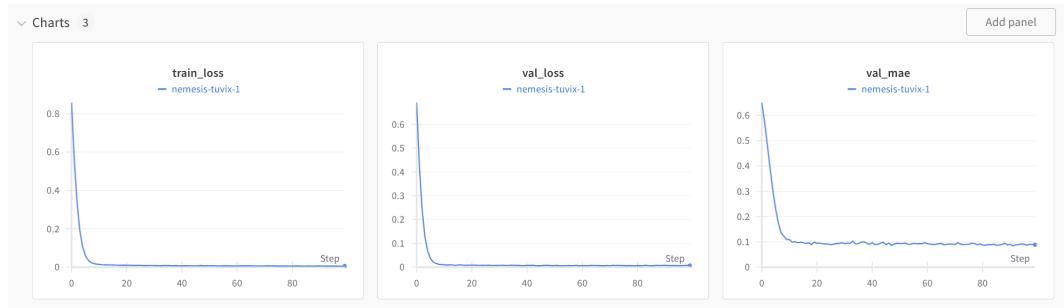
Sau khi hoàn thành huấn luyện, hãy kết thúc **wandb** session.

```

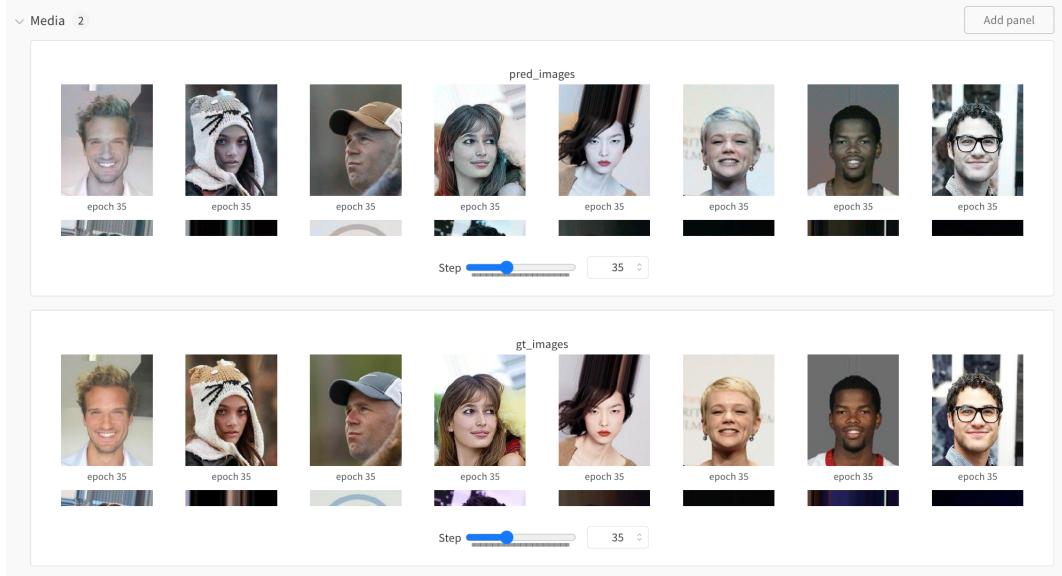
1 if use_wandb:
2     wandb.finish()

```

Theo dõi quá trình huấn luyện trên **wandb**.



Hình 6: Theo dõi kết quả loss và metrics theo từng epoch trên **wandb**.



Hình 7: Theo dõi quá trình tạo ảnh của mô hình theo từng epoch trên **wandb**.

## 5. Inference

Bạn có thể sử dụng checkpoint sẵn có để tiến hành quá trình suy luận thử nghiệm.

```

1 # Download the checkpoint
2 !gdown 1-OIcaofrE8cNbVUn1Ydo2WyShkxohv9r

1 # Load the model
2 colordiff_model = ColorDiffusion(unet_config, beta_schedule)
3 colordiff_model.set_new_noise_schedule(device)
4 load_state = torch.load('./best_model.pth')
5 colordiff_model.load_state_dict(load_state, strict=True)
6 colordiff_model.eval().to(device)
7
8
9 # Load original image
10 showed_img_idx = 55
11 img_path = img_paths[showed_img_idx]
12 img_bgr = cv2.imread(img_path)
13
14 img_lab = cv2.cvtColor(img, cv2.COLOR_BGR2LAB)
15 img_l = img_lab[:, :, 1]
16 plt.imshow(img_l, cmap='gray')
17 plt.axis(False)
18 plt.savefig("e1_gray.png")
19 plt.show()
20
21 img_rgb = cv2.cvtColor(img_bgr, cv2.COLOR_BGR2RGB)
22 plt.imshow(img_rgb)
23 plt.axis(False)
24 plt.savefig("e2_full_color.png")
25 plt.show()
26
27
28 # Infer
29 test_imgpath = img_paths[showed_img_idx]
30 test_dataset = ColorDataset([test_imgpath])
31 test_sample = next(iter(test_dataset))

```

```

32
33 def inference(model, test_sample):
34     with torch.no_grad():
35         output, visuals = model.restoration(
36             test_sample[1].unsqueeze(0).to(device)
37         )
38     return output, visuals
39
40 output, visuals = inference(colordiff_model, test_sample)
41
42
43 # Show the results
44 def show_tensor_image(img_l, img_ab, is_save=False):
45     img_l = img_l.permute(1, 2, 0).numpy()
46     img_ab = img_ab.permute(1, 2, 0).numpy()
47     img_ab = cv2.resize(img_ab, (img_l.shape[1], img_l.shape[0]), interpolation=
48         cv2.INTER_LINEAR)
49     arr_lab = np.concatenate([img_l, img_ab], axis=2)
50     arr_lab = (arr_lab + 1.0) * 255 / 2
51     arr_lab = np.clip(arr_lab, 0, 255).astype(np.uint8)
52     arr_bgr = cv2.cvtColor(arr_lab, cv2.COLOR_LAB2BGR)
53     if is_save:
54         cv2.imwrite("results.png", arr_bgr)
55     arr_bgr = cv2.cvtColor(arr_bgr, cv2.COLOR_BGR2RGB)
56     plt.imshow(arr_bgr)
57     plt.axis(False)
58
59 output, visuals = inference(colordiff_model, test_sample)
60 show_tensor_image(test_sample[0], output[0].cpu(), is_save=True)
61 plt.show()

```

Kết quả thực nghiệm mô hình sau khi huấn luyện



Hình 8: Kết quả thực nghiệm mô hình sau khi huấn luyện.

## Phần III: Câu hỏi trắc nghiệm

1. Trong Diffusion Model, Loss Function là:
  - (a) ELBO (Evidence Lower Bound)
  - (b) VLB (Variational Lower Bound)
  - (c) Simplified MSE
  - (d) Tất cả đáp án đều đúng.
2. Gray channel đóng vai trò gì trong quá trình huấn luyện mô hình Diffusion-based Image Colorization?
  - (a) Nó được sử dụng làm điều kiện mang thông tin cấu trúc của ảnh trong quá trình tạo màu.
  - (b) Nó bị bỏ qua trong quá trình tạo màu để chỉ tập trung vào sắc độ.
  - (c) Nó được tăng cường thông tin qua mô hình khuếch tán để cải thiện độ trung thực của màu sắc ở đầu ra cuối cùng.
3. Đầu là nhược điểm chính của DDPM trong việc sinh ảnh dữ liệu có chất lượng, độ phân giải cao? (chọn phương án đúng nhất)
  - (a) Tốn nhiều thời gian, tài nguyên tính toán để thực hiện sampling process.
  - (b) Không ổn định trong việc huấn luyện
  - (c) Phụ thuộc vào surrogate loss.
  - (d) Khó khăn trong việc thiết kế kiến trúc mô hình.
4. Phát biểu nào sau đây đúng về Diffusion-based Image Colorization:
  - (a) Cần xác định tấm ảnh đầy màu sắc làm groundtruth.
  - (b) Ràng buộc trong việc sử dụng các kênh màu phổ biến RGB hoặc HSV.
  - (c) Quá trình Sampling là sự kết hợp giữa việc sử dụng color channel được mô hình dự đoán (sau khi denoise) và việc sử dụng kênh màu gray-scale để tạo nên tấm ảnh Lab Image.
5. Phương pháp định tính được sử dụng để đánh giá mô hình Image Colorization là?
  - (a) PSNR
  - (b) SSIM
  - (c) FID Score
  - (d) Fooling rate
6. Công dụng chính của Weights & Biases (Wandb) trong việc huấn luyện các mô hình Machine Learning là gì?
  - (a) Tiền xử lý và làm sạch dữ liệu cho quá trình huấn luyện.
  - (b) Huấn luyện và triển khai trực tiếp các mô hình Machine Learning.
  - (c) Theo dõi, quan sát và so sánh các thử nghiệm Machine Learning.
  - (d) Finetune các siêu tham số một cách tự động.

- **Hết** -