AI VIET NAM – AI COURSE 2023

VAE-based Image Colorization

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

Ngày 2 tháng 4 năm 2024

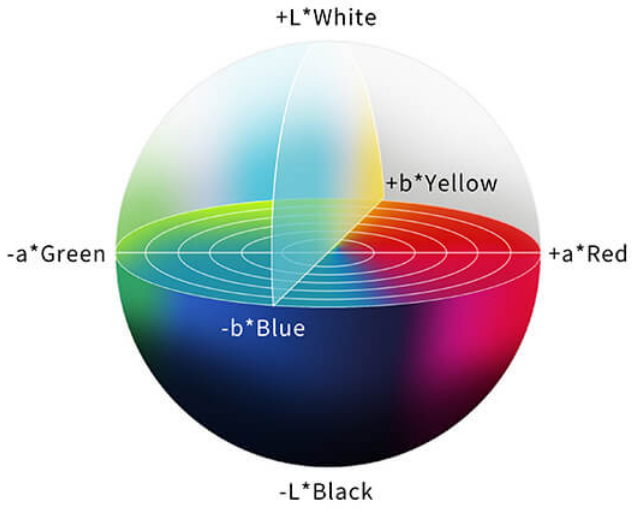
Phần I: Giới thiệu

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

Vấn đề tô màu cho ảnh mang lại một sự thuận lợi đáng kể về mặt dữ liệu, khi mà việc gán nhãn bị loại bỏ hoàn toàn. Bởi, mỗi bức ảnh đều có thể được phân chia thành hai thành phần thông tin chính: gray channel và color channel. Trong đó, gray channel được sử dụng làm đầu vào cho mô hình, đóng vai trò là cơ sở cho quá trình dự đoán. Mô hình sẽ tiến hành dự đoán color channel, tức là thông tin về màu sắc, dựa trên gray channel. Khi mà mô hình đã hoàn thành việc dự đoán, color channel sẽ được kết hợp với gray channel để tạo ra một ảnh hoàn chỉnh.

Trong các bài toán tô màu cho ảnh, không gian màu phổ biến thường được sử dụng là *Lab* thay vì *RGB* như trong các bài toán xử lý ảnh thông thường, trong đó:



Hình 2: Minh họa cho không gian màu *Lab*

- **L (Lightness)**: Đây là thành phần đại diện cho độ sáng của một pixel trong ảnh. Thành phần L có thể được xem như phiên bản grayscale của ảnh.
- **a (Green-Red)**: Thành phần a biểu diễn sự khác biệt màu giữa các màu xanh lam và đỏ. Khi giá trị của a tăng, màu sắc trở nên đỏ hơn. Khi giá trị giảm, màu sắc trở nên xanh hơn.
- **b (Blue-Yellow)**: Thành phần b biểu diễn sự khác biệt màu giữa các màu xanh lá cây và màu vàng. Khi giá trị của b tăng, màu sắc trở nên vàng hơn. Khi giá trị giảm, màu sắc trở nên xanh hơn.

Bằng cách sử dụng không gian màu *Lab*, mô hình của chúng ta sẽ nhận đầu vào là kênh L, đại diện cho độ sáng, và sử dụng kênh ab như ground truth của mô hình. Vì vậy, số lượng kênh màu cần dự đoán là 2, bao gồm thành phần màu a và b, giúp giảm bớt độ phức tạp so với việc dự đoán 3 kênh màu khi sử dụng không gian màu *RGB*.

Có nhiều phương pháp được sử dụng để thực hiện việc tô màu cho ảnh, bao gồm sử dụng mạng CNN truyền thống, hay các mạng tạo sinh đã và đang phát triển trong những năm gần đây như GAN, VAE (Variational Autoencoder) và cả Diffusion Models. Mỗi phương pháp mang lại những ưu điểm và hạn chế riêng, đều hướng tới mục tiêu cuối cùng là tạo ra các ảnh màu tự nhiên và chân thực.

Trong phần này, chúng ta sẽ tập trung vào việc xây dựng một mô hình dựa trên VAE để giải quyết vấn đề Image Colorization. Input và output của chương trình như sau:

- **Input:** Ảnh xám G (L channel).
- **Output:** Trường ảnh màu C (ab channels).

Phần II: Nội dung

Trong phần này, chúng ta sẽ triển khai mô hình VAE-base Image Colorization dựa trên bài báo Learning Diverse Image Colorization để học cách tạo ra tập ảnh màu đa dạng về mặt kết quả. Cụ thể, ta sẽ xây dựng chương trình dựa trên bộ dữ liệu LFW (Labeled Faces in the Wild Home), một trong những bộ dữ liệu quan trọng và phổ biến trong lĩnh vực nhận dạng khuôn mặt. Bộ dữ liệu này chứa các hình ảnh của khuôn mặt được thu thập từ các bức ảnh chụp thực tế, bao gồm nhiều điều kiện ánh sáng, góc chụp và nền khác nhau.

Hình 3: Ảnh minh họa cho LFW dataset

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

a) Data Preparation: Chuẩn bị dữ liệu cho tập huấn luyện.

b) Models: Xây dựng mô hình VAE và mô hình MDN (Mixture Density Network).

c) Loss Functions: Xây dựng hàm mất mát cho mô hình VAE và mô hình MDN.

d) Trainer: Xây dựng các hàm để huấn luyện cho từng mô hình.

e) 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 LFW thông qua dòng lệnh dưới đây. Bộ dữ liệu bao gồm hơn 12,000 ảnh train và 1,000 ảnh test. Cùng với đó là một bộ đặc trưng tương ứng với từng ảnh, trích xuất từ một mạng VGG được huấn luyện mạnh mẽ trên bộ dữ liệu lớn ImageNet.

   Tải dữ liệu

```
!gdown 187x5YSXYibG4QwC5m_Hx8cNzPGVTXv6G
!unzip -q data.zip
!rm data.zip
```

   Khai báo các thư viện:

```
import os
import numpy as np
from tqdm import tqdm
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
```

```
from torch.utils.data import DataLoader
import cv2
import numpy as np
from torch.utils.data import Dataset
```

Khởi tạo class ColorDataset:

```
class ColorDataset(Dataset):
    def __init__(self, out_directory, listdir=None,
                 featslistdir=None, shape=(64, 64),
                 outshape=(256, 256), split="train"):

        # Save paths to a list
        self.img_fns = []
        self.feats_fns = []

        with open("%s/list.%s.vae.txt" % (listdir, split), "r") as ftr:
            for img_fn in ftr:
                self.img_fns.append(img_fn.strip("\n"))

        with open("%s/list.%s.txt" % (featslistdir, split), "r") as ftr:
            for feats_fn in ftr:
                self.feats_fns.append(feats_fn.strip("\n"))

        self.img_num = min(len(self.img_fns), len(self.feats_fns))
        self.shape = shape
        self.outshape = outshape
        self.out_directory = out_directory

        # Create a dictionary to save weight of 313 ab bins
        self.lossweights = None
        countbins = 1.0 / np.load("data/zhang_weights/prior_probs.npy")
        binedges = np.load("data/zhang_weights/ab_quantize.npy").reshape(2, 313)
        lossweights = {}
        for i in range(313):
            if binedges[0, i] not in lossweights:
                lossweights[binedges[0, i]] = {}
            lossweights[binedges[0, i]][binedges[1, i]] = countbins[i]
        self.binedges = binedges
        self.lossweights = lossweights

    def __len__(self):
        return self.img_num

    def __getitem__(self, idx):
        # Declare empty arrays to get values
        color_ab = np.zeros((2, self.shape[0], self.shape[1]),
                            dtype="f")
        weights = np.ones((2, self.shape[0], self.shape[1]),
                          dtype="f")
        recon_const = np.zeros((1, self.shape[0], self.shape[1]),
                               dtype="f")
        recon_const_outres = np.zeros((1, self.outshape[0], self.outshape[1]),
                                      dtype="f")
        greyfeats = np.zeros((512, 28, 28), dtype="f")

        # Read and reshape
        img_large = cv2.imread(self.img_fns[idx])
        if self.shape is not None:
            img = cv2.resize(img_large, (self.shape[0], self.shape[1]))
            img_outres = cv2.resize(img_large,
```

```
                                    (self.outshape[0], self.outshape[1]))

        # Convert BGR to LAB
        img_lab = cv2.cvtColor(img, cv2.COLOR_BGR2LAB)
        img_lab_outres = cv2.cvtColor(img_outres, cv2.COLOR_BGR2LAB)

        # Normalize to [-1..1]
        img_lab = ((img_lab * 2.0) / 255.0) - 1.0
        img_lab_outres = ((img_lab_outres * 2.0) / 255.0) - 1.0

        recon_const[0, :, :] = img_lab[..., 0]
        recon_const_outres[0, :, :] = img_lab_outres[..., 0]

        color_ab[0, :, :] = img_lab[..., 1].reshape(1, self.shape[0],
                                                    self.shape[1])
        color_ab[1, :, :] = img_lab[..., 2].reshape(1, self.shape[0],
                                                    self.shape[1])

        if self.lossweights is not None:
            weights = self.__getweights__(color_ab)

        # Load feature maps
        featobj = np.load(self.feats_fns[idx])
        greyfeats[:, :, :] = featobj["arr_0"]

        return color_ab, recon_const, weights, recon_const_outres, greyfeats

    def __getweights__(self, img):
        """
        Calculate weight values for each pixel of an image.
        """
        img_vec = img.reshape(-1)
        img_vec = img_vec * 128.0
        img_lossweights = np.zeros(img.shape, dtype="f")
        img_vec_a = img_vec[: np.prod(self.shape)]
        binedges_a = self.binedges[0, ...].reshape(-1)
        binid_a = [binedges_a.flat[np.abs(binedges_a - v).argmin()]
                   for v in img_vec_a]
        img_vec_b = img_vec[np.prod(self.shape) :]
        binedges_b = self.binedges[1, ...].reshape(-1)
        binid_b = [binedges_b.flat[np.abs(binedges_b - v).argmin()]
                   for v in img_vec_b]
        binweights = np.array([self.lossweights[v1][v2] for v1, v2 in zip(binid_a
    , binid_b)])
        img_lossweights[0, :, :] = binweights.reshape(self.shape[0],
        self.shape[1])
        img_lossweights[1, :, :] = binweights.reshape(self.shape[0], self.shape
    [1])
        return img_lossweights

    def saveoutput_gt(self, net_op, gt, prefix, batch_size,
                      num_cols=8, net_recon_const=None):
        """
        Save images
        """
        net_out_img = self.__tiledoutput__(net_op, batch_size, num_cols=num_cols,
                                           net_recon_const=net_recon_const)
        gt_out_img = self.__tiledoutput__(gt, batch_size, num_cols=num_cols,
                                          net_recon_const=net_recon_const)

```

```
        num_rows = np.int_(np.ceil((batch_size * 1.0) / num_cols))
        border_img = 255 * np.ones((num_rows * self.outshape[0], 128, 3),
                                   dtype="uint8")
        out_fn_pred = "%s/%s.png" % (self.out_directory, prefix)
        cv2.imwrite(out_fn_pred,
                    np.concatenate((net_out_img, border_img, gt_out_img), axis=1)
    )

    def __tiledoutput__(self, net_op, batch_size,
                        num_cols=8, net_recon_const=None):
        """
        Generate a combined image from these inputs by stitching the images into
    a large image.
        """
        num_rows = np.int_(np.ceil((batch_size * 1.0) / num_cols))
        out_img = np.zeros((num_rows*self.outshape[0], num_cols*self.outshape[1],
     3),
                           dtype="uint8")
        img_lab = np.zeros((self.outshape[0], self.outshape[1], 3),
                           dtype="uint8")
        c = 0
        r = 0

        for i in range(batch_size):
            if i % num_cols == 0 and i > 0:
                r = r + 1
                c = 0
            img_lab[..., 0] = self.__decodeimg__(net_recon_const[i, 0, :, :].
    reshape(self.outshape[0], self.outshape[1]))
            img_lab[..., 1] = self.__decodeimg__(net_op[i, 0, :, :].reshape(self.
    shape[0], self.shape[1]))
            img_lab[..., 2] = self.__decodeimg__(net_op[i, 1, :, :].reshape(self.
    shape[0], self.shape[1]))
            img_rgb = cv2.cvtColor(img_lab, cv2.COLOR_LAB2BGR)
            out_img[
                r * self.outshape[0] : (r + 1) * self.outshape[0],
                c * self.outshape[1] : (c + 1) * self.outshape[1],
                ...,
            ] = img_rgb
            c = c + 1

        return out_img

    def __decodeimg__(self, img_enc):
        """
        Denormalize from [-1..1] to [0..255]
        """
        img_dec = (((img_enc + 1.0) * 1.0) / 2.0) * 255.0
        img_dec[img_dec < 0.0] = 0.0
        img_dec[img_dec > 255.0] = 255.0
        return cv2.resize(np.uint8(img_dec), (self.outshape[0], self.outshape[1])
    )
```

Khởi tạo các siêu tham số toàn cụ cho chương trình.

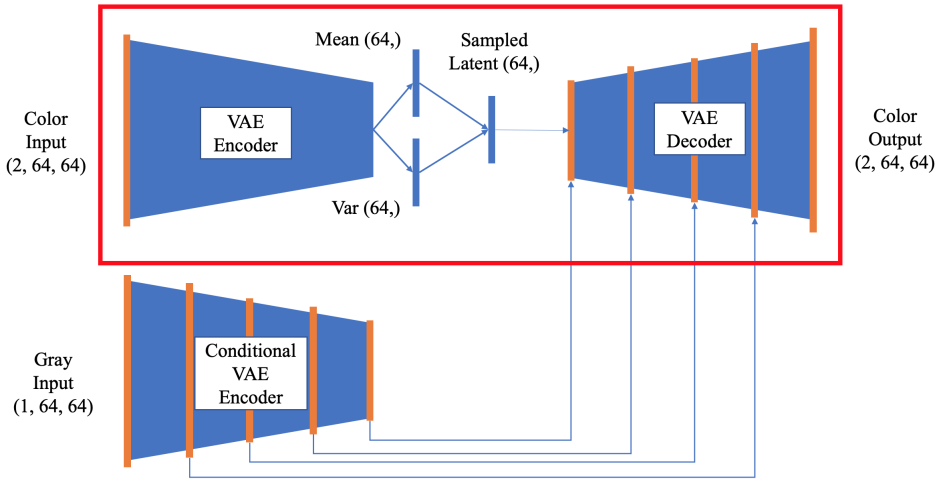
```
# Declare hyperparameters
args = {
    "gpu": 1,
    "epochs": 2,
    "batchsize": 32,
    "hiddensize": 64,
```

```
    "nthreads": 2,
    "epochs_mdn": 2,
    "nmix": 8,
    "logstep": 100,
    "dataset_key": "lfw"
}

def get_dirpaths(args):
    if args["dataset_key"] == "lfw":
        out_dir = "data/output/lfw"
        listdir = "data/imglist/lfw"
        featslistdir = "data/featslist/lfw"
    else:
        raise NameError("[ERROR] Incorrect key: %s" % (args.dataset_key))
    return out_dir, listdir, featslistdir
```

2. Models

Chúng ta sẽ tiến hành xây dựng mô hình VAE và mô hình MDN.

(a) **VAE model:** Trong bài toán này, chúng ta sẽ sử dụng một biến thể của mô hình VAE, được gọi là Conditional VAE (CVAE). Mô hình này bao gồm ba phần: khối Encoder chính và khối Decoder chính (hai khối này tạo thành một mạng VAE cơ bản, được bao quanh bởi hình chữ nhật màu đỏ), cùng với một khối Conditional Encoder (khối này giúp mô hình tận dụng tối đa những trường thông tin có sẵn). Đầu vào của mạng VAE cơ bản là trường màu C có kích thước (2 x h x w), và đầu ra là một feature map có kích thước tương tự (2 x h x w). Đồng thời, ảnh xám G (1 x h x w) cũng được sử dụng làm điểm khởi đầu cho khối Conditional Encoder để trích xuất các feature maps chứa thông tin cục bộ, và sau đó được sử dụng làm điều kiện để làm tăng khả năng cho khối Decoder.



Hình 4: Ảnh minh họa cho mô hình Conditional VAE.

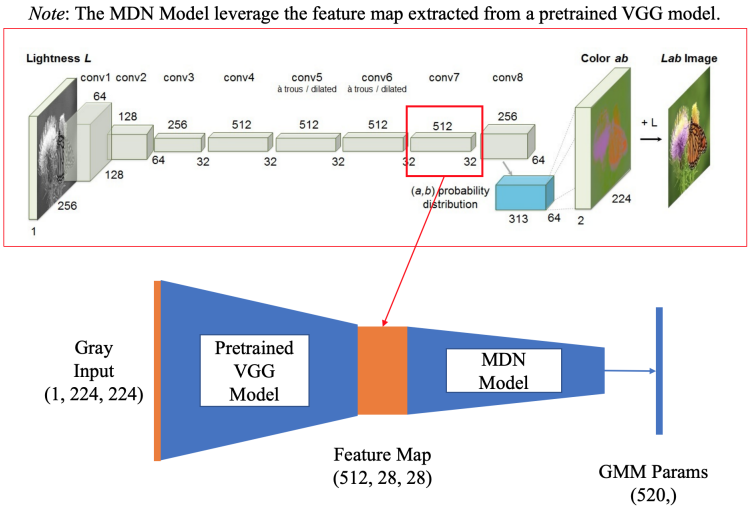
```
import torch
import torch.nn as nn
import torch.nn.functional as F

class VAE(nn.Module):
    def __init__(self):
        super(VAE, self).__init__()
```

```
        self.hidden_size = 64

        # Encoder block
        self.enc_conv1 = nn.Conv2d(2, 128, 5, stride=2, padding=2)
        self.enc_bn1 = nn.BatchNorm2d(128)
        self.enc_conv2 = nn.Conv2d(128, 256, 5, stride=2, padding=2)
        self.enc_bn2 = nn.BatchNorm2d(256)
        self.enc_conv3 = nn.Conv2d(256, 512, 5, stride=2, padding=2)
        self.enc_bn3 = nn.BatchNorm2d(512)
        self.enc_conv4 = nn.Conv2d(512, 1024, 3, stride=2, padding=1)
        self.enc_bn4 = nn.BatchNorm2d(1024)
        self.enc_fc1 = nn.Linear(4*4*1024, self.hidden_size*2)
        self.enc_dropout1 = nn.Dropout(p=0.7)

        # Conditional encoder block
        self.cond_enc_conv1 = nn.Conv2d(1, 128, 5, stride=2, padding=2)
        self.cond_enc_bn1 = nn.BatchNorm2d(128)
        self.cond_enc_conv2 = nn.Conv2d(128, 256, 5, stride=2, padding=2)
        self.cond_enc_bn2 = nn.BatchNorm2d(256)
        self.cond_enc_conv3 = nn.Conv2d(256, 512, 5, stride=2, padding=2)
        self.cond_enc_bn3 = nn.BatchNorm2d(512)
        self.cond_enc_conv4 = nn.Conv2d(512, 1024, 3, stride=2, padding=1)
        self.cond_enc_bn4 = nn.BatchNorm2d(1024)

        # Decoder block
        self.dec_upsamp1 = nn.Upsample(scale_factor=4, mode='bilinear')
        self.dec_conv1 = nn.Conv2d(1024+self.hidden_size, 512, 3, stride=1,
padding=1)
        self.dec_bn1 = nn.BatchNorm2d(512)
        self.dec_upsamp2 = nn.Upsample(scale_factor=2, mode='bilinear')
        self.dec_conv2 = nn.Conv2d(512*2, 256, 5, stride=1, padding=2)
        self.dec_bn2 = nn.BatchNorm2d(256)
        self.dec_upsamp3 = nn.Upsample(scale_factor=2, mode='bilinear')
        self.dec_conv3 = nn.Conv2d(256*2, 128, 5, stride=1, padding=2)
        self.dec_bn3 = nn.BatchNorm2d(128)
        self.dec_upsamp4 = nn.Upsample(scale_factor=2, mode='bilinear')
        self.dec_conv4 = nn.Conv2d(128*2, 64, 5, stride=1, padding=2)
        self.dec_bn4 = nn.BatchNorm2d(64)
        self.dec_upsamp5 = nn.Upsample(scale_factor=2, mode='bilinear')
        self.dec_conv5 = nn.Conv2d(64, 2, 5, stride=1, padding=2)

    def encoder(self, x):
        x = F.relu(self.enc_conv1(x))
        x = self.enc_bn1(x)
        x = F.relu(self.enc_conv2(x))
        x = self.enc_bn2(x)
        x = F.relu(self.enc_conv3(x))
        x = self.enc_bn3(x)
        x = F.relu(self.enc_conv4(x))
        x = self.enc_bn4(x)
        x = x.view(-1, 4*4*1024)
        x = self.enc_dropout1(x)
        x = self.enc_fc1(x)
        mu = x[..., :self.hidden_size]
        logvar = x[..., self.hidden_size:]
        return mu, logvar

    def cond_encoder(self, x):
        x = F.relu(self.cond_enc_conv1(x))
        sc_feat32 = self.cond_enc_bn1(x)
```

```
        x = F.relu(self.cond_enc_conv2(sc_feat32))
        sc_feat16 = self.cond_enc_bn2(x)
        x = F.relu(self.cond_enc_conv3(sc_feat16))
        sc_feat8 = self.cond_enc_bn3(x)
        x = F.relu(self.cond_enc_conv4(sc_feat8))
        sc_feat4 = self.cond_enc_bn4(x)
        return sc_feat32, sc_feat16, sc_feat8, sc_feat4

    def decoder(self, z, sc_feat32, sc_feat16, sc_feat8, sc_feat4):
        x = z.view(-1, self.hidden_size, 1, 1)
        x = self.dec_upsamp1(x)
        x = torch.cat([x, sc_feat4], 1)
        x = F.relu(self.dec_conv1(x))
        x = self.dec_bn1(x)
        x = self.dec_upsamp2(x)
        x = torch.cat([x, sc_feat8], 1)
        x = F.relu(self.dec_conv2(x))
        x = self.dec_bn2(x)
        x = self.dec_upsamp3(x)
        x = torch.cat([x, sc_feat16], 1)
        x = F.relu(self.dec_conv3(x))
        x = self.dec_bn3(x)
        x = self.dec_upsamp4(x)
        x = torch.cat([x, sc_feat32], 1)
        x = F.relu(self.dec_conv4(x))
        x = self.dec_bn4(x)
        x = self.dec_upsamp5(x)
        x = torch.tanh(self.dec_conv5(x))
        return x

    def forward(self, color, greylevel, z_in=None):
        sc_feat32, sc_feat16, sc_feat8, sc_feat4 = self.cond_encoder(
    greylevel)
        mu, logvar = self.encoder(color)
        if self.training:
            stddev = torch.sqrt(torch.exp(logvar))
            eps = torch.randn_like(stddev)
            z = mu + eps * stddev
            z = z.to(greylevel.device)
        else:
            z = z_in
            z = z.to(greylevel.device)
        color_out = self.decoder(z, sc_feat32, sc_feat16, sc_feat8, sc_feat4)
        return mu, logvar, color_out
```

(b) **MDN model**

Đầu vào của một mạng Conditional Variational Autoencoder (CVAE) yêu cầu thông tin về cả trường màu C và ảnh xám G. Trong quá trình huấn luyện, khối Encoder chính ánh xạ thông tin của trường màu C thành phân phối hậu nghiệm P, sau đó lấy mẫu từ phân phối P để làm điểm khởi đầu cho khối Decoder. Tuy nhiên, trong quá trình dự đoán, không có thông tin về trường màu C được cung cấp. Chính vì thế, một mạng MDN (Mixture Density Network) được đã được thiết kế. MDN nhận đầu vào là vector đặc trưng, được tạo ra bằng cách cho ảnh xám G đi qua mạng VGG đã được huấn luyện trước trong bài báo Colorful Image Colorization. Kết quả đầu ra của mô hình MDN sau đó được sử dụng để tạo ra các tham số phân phối cho mô hình Gaussian Mixture Model, một mô hình thực hiện việc xấp xỉ phân phối P được tạo ra từ khối Encoder vừa được huấn luyện trước đó.



Hình 5: Ảnh minh họa cho mô hình MDN.

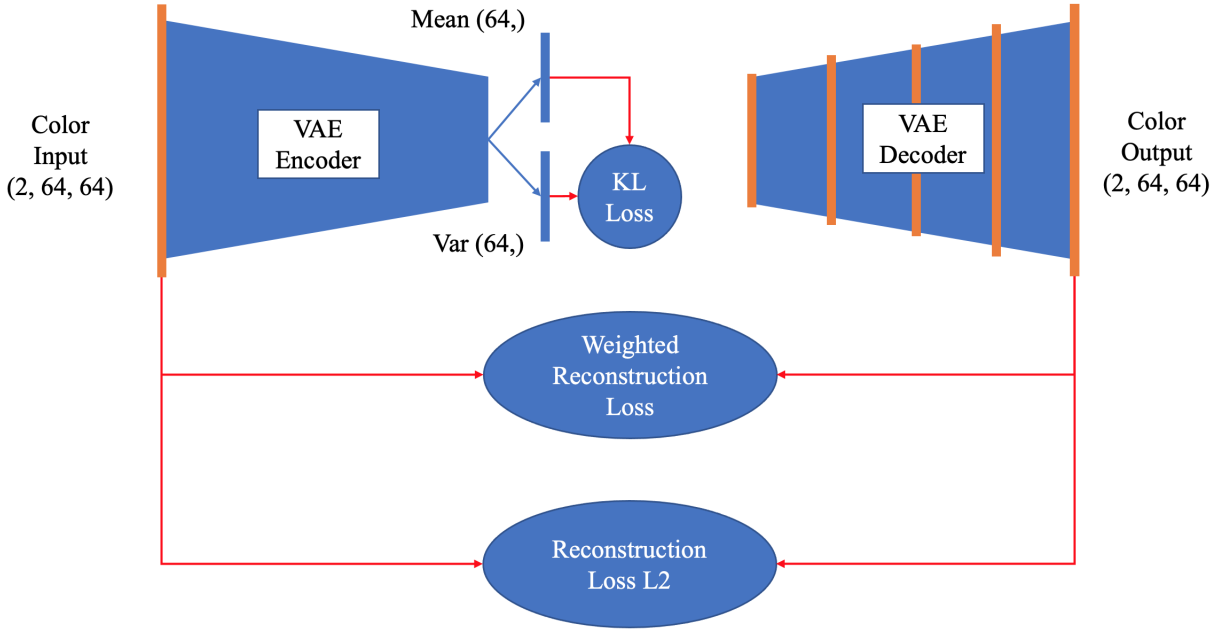
```python
import torch
import torch.nn as nn
import torch.nn.functional as F

class MDN(nn.Module):
    def __init__(self):
        super(MDN, self).__init__()

        self.feats_nch = 512
        self.hidden_size = 64
        self.nmix = 8
        self.nout = (self.hidden_size + 1) * self.nmix

        self.model = nn.Sequential(
            nn.Conv2d(self.feats_nch, 384, 5, stride=1, padding=2),
            nn.BatchNorm2d(384),
            nn.ReLU(),
            nn.Conv2d(384, 320, 5, stride=1, padding=2),
            nn.BatchNorm2d(320),
            nn.ReLU(),
            nn.Conv2d(320, 288, 5, stride=1, padding=2),
            nn.BatchNorm2d(288),
            nn.ReLU(),
            nn.Conv2d(288, 256, 5, stride=2, padding=2),
            nn.BatchNorm2d(256),
            nn.ReLU(),
            nn.Conv2d(256, 128, 5, stride=1, padding=2),
            nn.BatchNorm2d(128),
            nn.ReLU(),
            nn.Conv2d(128, 96, 5, stride=2, padding=2),
            nn.BatchNorm2d(96),
            nn.ReLU(),
            nn.Conv2d(96, 64, 5, stride=2, padding=2),
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.Dropout(p=0.7)
        )

        self.fc = nn.Linear(4 * 4 * 64, self.nout)
```

```
    def forward(self, feats):
        x = self.model(feats)
        x = x.view(-1, 4 * 4 * 64)
        x = F.relu(x)
        x = F.dropout(x, p=0.7, training=self.training)
        x = self.fc(x)
        return x
```

3. Loss Functions

Trong phần này chúng ta xây dựng hàm mất mát cho các mô hình VAE và MDN.

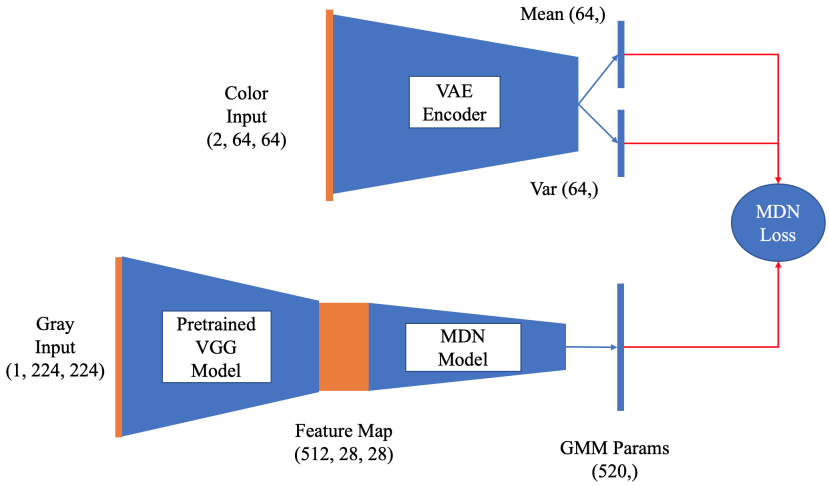
VAE Loss



Hình 6: Ảnh minh họa cho VAE Loss.

```
def vae_loss(mu, logvar, pred, gt, lossweights, batchsize):
    """
    Return the loss values of the VAE model.
    """
    kl_element = torch.add(torch.add(torch.add(mu.pow(2), logvar.exp()), -1), logvar.mul(-1))
    kl_loss = torch.sum(kl_element).mul(0.5)
    gt = gt.reshape(-1, 64 * 64 * 2)
    pred = pred.reshape(-1, 64 * 64 * 2)
    recon_element = torch.sqrt(torch.sum(torch.mul(torch.add(gt, pred.mul(-1)).pow(2), lossweights), 1))
    recon_loss = torch.sum(recon_element).mul(1.0 / (batchsize))

    recon_element_l2 = torch.sqrt(torch.sum(torch.add(gt, pred.mul(-1)).pow(2), 1))
    recon_loss_l2 = torch.sum(recon_element_l2).mul(1.0 / (batchsize))

    return kl_loss, recon_loss, recon_loss_l2
```

MDN Loss



Hình 7: Ảnh minh họa cho MDN Loss.

```python
def get_gmm_coeffs(gmm_params):
    """
    Return the distribution coefficients of the GMM.
    """
    gmm_mu = gmm_params[..., : args["hiddensize"] * args["nmix"]]
    gmm_mu.contiguous()
    gmm_pi_activ = gmm_params[..., args["hiddensize"] * args["nmix"] :]
    gmm_pi_activ.contiguous()
    gmm_pi = F.softmax(gmm_pi_activ, dim=1)
    return gmm_mu, gmm_pi

def mdn_loss(gmm_params, mu, stddev, batchsize):
    """
    Calculates the loss by comparing two distribution
    - the predicted distribution of the MDN (given by gmm_mu and gmm_pi) with
    - the target distribution created by the Encoder block (given by mu and
    stddev).
    """
    gmm_mu, gmm_pi = get_gmm_coeffs(gmm_params)
    eps = torch.randn(stddev.size()).normal_().cuda()
    z = torch.add(mu, torch.mul(eps, stddev))
    z_flat = z.repeat(1, args["nmix"])
    z_flat = z_flat.reshape(batchsize * args["nmix"], args["hiddensize"])
    gmm_mu_flat = gmm_mu.reshape(batchsize * args["nmix"], args["hiddensize"])
    dist_all = torch.sqrt(torch.sum(torch.add(z_flat, gmm_mu_flat.mul(-1)).pow(2)
    .mul(50), 1))
    dist_all = dist_all.reshape(batchsize, args["nmix"])
    dist_min, selectids = torch.min(dist_all, 1)
    gmm_pi_min = torch.gather(gmm_pi, 1, selectids.reshape(-1, 1))
    gmm_loss = torch.mean(torch.add(-1 * torch.log(gmm_pi_min + 1e-30), dist_min)
    )
    gmm_loss_l2 = torch.mean(dist_min)
    return gmm_loss, gmm_loss_l2
```

4. Trainer

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

Train VAE model

```python
def test_vae(model):
    model.eval()

    # Load hyperparameters
    out_dir, listdir, featslistdir = get_dirpaths(args)
    batchsize = args["batchsize"]
    hiddensize = args["hiddensize"]
    nmix = args["nmix"]

    # Create DataLoader
    data = ColorDataset(
        os.path.join(out_dir, "images"),
        listdir=listdir,
        featslistdir=featslistdir,
        split="test",
    )
    nbatches = np.int_(np.floor(data.img_num / batchsize))
    data_loader = DataLoader(
        dataset=data,
        num_workers=args["nthreads"],
        batch_size=batchsize,
        shuffle=False,
        drop_last=True,
    )

    # Eval
    test_loss = 0.0
    for batch_idx, (
        batch,
        batch_recon_const,
        batch_weights,
        batch_recon_const_outres,
        _,
    ) in tqdm(enumerate(data_loader), total=nbatches):

        input_color = batch.cuda()
        lossweights = batch_weights.cuda()
        lossweights = lossweights.reshape(batchsize, -1)
        input_greylevel = batch_recon_const.cuda()
        z = torch.randn(batchsize, hiddensize)

        mu, logvar, color_out = model(input_color, input_greylevel, z)
        _, _, recon_loss_l2 = vae_loss(
            mu, logvar, color_out, input_color, lossweights, batchsize
        )
        test_loss = test_loss + recon_loss_l2.item()

    test_loss = (test_loss * 1.0) / nbatches
    model.train()

    return test_loss


def train_vae():
    # Load hyperparameters
    out_dir, listdir, featslistdir = get_dirpaths(args)
    batchsize = args["batchsize"]
    hiddensize = args["hiddensize"]
    nmix = args["nmix"]
```

```
    nepochs = args["epochs"]

    # Create DataLoader
    data = ColorDataset(
        os.path.join(out_dir, "images"),
        listdir=listdir,
        featslistdir=featslistdir,
        split="train",
    )
    nbatches = np.int_(np.floor(data.img_num / batchsize))
    data_loader = DataLoader(
        dataset=data,
        num_workers=args["nthreads"],
        batch_size=batchsize,
        shuffle=True,
        drop_last=True,
    )

    # Initialize VAE model
    model = VAE()
    model.cuda()
    model.train()

    optimizer = optim.Adam(model.parameters(), lr=5e-5)

    # Train
    itr_idx = 0
    for epochs in range(nepochs):
        train_loss = 0.0

        for batch_idx, (
            batch,
            batch_recon_const,
            batch_weights,
            batch_recon_const_outres,
            _,
        ) in tqdm(enumerate(data_loader), total=nbatches):

            input_color = batch.cuda()
            lossweights = batch_weights.cuda()
            lossweights = lossweights.reshape(batchsize, -1)
            input_greylevel = batch_recon_const.cuda()
            z = torch.randn(batchsize, hiddensize)

            optimizer.zero_grad()
            mu, logvar, color_out = model(input_color, input_greylevel, z)
            kl_loss, recon_loss, recon_loss_l2 = vae_loss(
                mu, logvar, color_out, input_color, lossweights, batchsize
            )
            loss = kl_loss.mul(1e-2) + recon_loss
            recon_loss_l2.detach()
            loss.backward()
            optimizer.step()

            train_loss = train_loss + recon_loss_l2.item()

            if batch_idx % args["logstep"] == 0:
                data.saveoutput_gt(
                    color_out.cpu().data.numpy(),
                    batch.numpy(),
```

```
                    "train_%05d_%05d" % (epochs, batch_idx),
                    batchsize,
                    net_recon_const=batch_recon_const_outres.numpy(),
                )

        train_loss = (train_loss * 1.0) / (nbatches)
        print("VAE Train Loss, epoch %d has loss %f" % (epochs, train_loss))
        test_loss = test_vae(model)
        print("VAE Test Loss, epoch %d has loss %f" % (epochs, test_loss))

        # Save VAE model
        torch.save(model.state_dict(), "%s/models/model_vae.pth" % (out_dir))

    print("Complete VAE training")

train_vae()
```

Train MDN model

```
def test_mdn(model_vae, model_mdn):
    # Load hyperparameters
    out_dir, listdir, featslistdir = get_dirpaths(args)
    batchsize = args["batchsize"]
    hiddensize = args["hiddensize"]
    nmix = args["nmix"]

    # Create DataLoader
    data = ColorDataset(
        os.path.join(out_dir, "images"), listdir, featslistdir, split="test"
    )
    nbatches = np.int_(np.floor(data.img_num / batchsize))
    data_loader = DataLoader(
        dataset=data,
        num_workers=args["nthreads"],
        batch_size=batchsize,
        shuffle=True,
        drop_last=True,
    )

    optimizer = optim.Adam(model_mdn.parameters(), lr=1e-3)

    # Eval
    model_vae.eval()
    model_mdn.eval()
    itr_idx = 0
    test_loss = 0.0

    for batch_idx, (batch, batch_recon_const, batch_weights, _, batch_feats) in
    tqdm(
        enumerate(data_loader), total=nbatches
    ):
        input_color = batch.cuda()
        input_greylevel = batch_recon_const.cuda()
        input_feats = batch_feats.cuda()
        z = torch.randn(batchsize, hiddensize)
        optimizer.zero_grad()

        # Get the parameters of the posterior distribution
        mu, logvar, _ = model_vae(input_color, input_greylevel, z)

        # Get the GMM vector
```

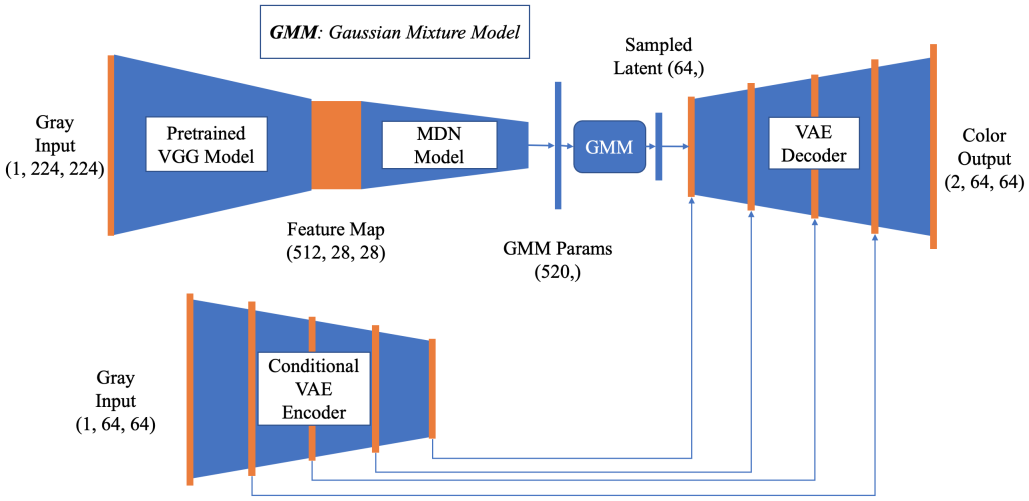
```
        mdn_gmm_params = model_mdn(input_feats)

        # Compare 2 distributions
        loss, _ = mdn_loss(mdn_gmm_params, mu, torch.sqrt(torch.exp(logvar)),
    batchsize)

        test_loss = test_loss + loss.item()

    test_loss = (test_loss * 1.0) / (nbatches)
    model_vae.train()
    return test_loss


def train_mdn():
    # Load hyperparameters
    out_dir, listdir, featslistdir = get_dirpaths(args)
    batchsize = args["batchsize"]
    hiddensize = args["hiddensize"]
    nmix = args["nmix"]
    nepochs = args["epochs_mdn"]

    # Create DataLoader
    data = ColorDataset(
        os.path.join(out_dir, "images"), listdir, featslistdir, split="train"
    )
    nbatches = np.int_(np.floor(data.img_num / batchsize))
    data_loader = DataLoader(
        dataset=data,
        num_workers=args["nthreads"],
        batch_size=batchsize,
        shuffle=True,
        drop_last=True,
    )

    # Initialize VAE model
    model_vae = VAE()
    model_vae.cuda()
    model_vae.load_state_dict(torch.load("%s/models/model_vae.pth" % (out_dir)))
    model_vae.eval()

    # Initialize MDN model
    model_mdn = MDN()
    model_mdn.cuda()
    model_mdn.train()

    optimizer = optim.Adam(model_mdn.parameters(), lr=1e-3)

    # Train
    itr_idx = 0
    for epochs_mdn in range(nepochs):
        train_loss = 0.0

        for batch_idx, (
            batch,
            batch_recon_const,
            batch_weights,
            _,
            batch_feats,
        ) in tqdm(enumerate(data_loader), total=nbatches):
            input_color = batch.cuda()
```

```
            input_greylevel = batch_recon_const.cuda()
            input_feats = batch_feats.cuda()
            z = torch.randn(batchsize, hiddensize)
            optimizer.zero_grad()

            # Get the parameters of the posterior distribution
            mu, logvar, _ = model_vae(input_color, input_greylevel, z)

            # Get the GMM vector
            mdn_gmm_params = model_mdn(input_feats)

            # Compare 2 distributions
            loss, loss_l2 = mdn_loss(
                mdn_gmm_params, mu, torch.sqrt(torch.exp(logvar)), batchsize
            )

            loss.backward()
            optimizer.step()
            train_loss = train_loss + loss.item()

        train_loss = (train_loss * 1.0) / (nbatches)
        test_loss = test_mdn(model_vae, model_mdn)
        print(
            f"End of epoch {epochs_mdn:3d} | Train Loss {train_loss:8.3f} |  Test Loss {test_loss:8.3f}"
        )

        # Save MDN model
        torch.save(model_mdn.state_dict(), "%s/models_mdn/model_mdn.pth" % (out_dir))

    print("Complete MDN training")

train_mdn()
```

5. Inference

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

```
# Download VAE checkpoint
!gdown 1wdyK198lXwwZO4NIB7DzJmA5arwUVWDU
# Download MDN checkpoint
!gdown 1AhilMrR_C04v7_sysuf5ffEVsQllo2W6
```



Hình 8: Ảnh minh họa cho Big Model ở giai đoạn inference

```python
def inference():
    # Load hyperparameters
    out_dir, listdir, featslistdir = get_dirpaths(args)
    batchsize = args["batchsize"]
    hiddensize = args["hiddensize"]
    nmix = args["nmix"]

    # Create DataLoader
    data = ColorDataset(
        os.path.join(out_dir, "images"),
        listdir=listdir,
        featslistdir=featslistdir,
        split="test",
    )

    nbatches = np.int_(np.floor(data.img_num / batchsize))

    data_loader = DataLoader(
        dataset=data,
        num_workers=args["nthreads"],
        batch_size=batchsize,
        shuffle=False,
        drop_last=True,
    )

    # Load VAE model
    model_vae = VAE()
    model_vae.cuda()
    model_vae.load_state_dict(torch.load("%s/models/model_vae.pth" % (out_dir)))
    model_vae.eval()

    # Load MDN model
    model_mdn = MDN()
    model_mdn.cuda()
    model_mdn.load_state_dict(torch.load("%s/models/model_mdn.pth" % (out_dir)))
    model_mdn.eval()

    # Infer
    for batch_idx, (
```

```
        batch,
        batch_recon_const,
        batch_weights,
        batch_recon_const_outres,
        batch_feats,
    ) in tqdm(enumerate(data_loader), total=nbatches):

        input_feats = batch_feats.cuda()

        # Get GMM parameters
        mdn_gmm_params = model_mdn(input_feats)
        gmm_mu, gmm_pi = get_gmm_coeffs(mdn_gmm_params)
        gmm_pi = gmm_pi.reshape(-1, 1)
        gmm_mu = gmm_mu.reshape(-1, hiddensize)

        for j in range(batchsize):
            batch_j = np.tile(batch[j, ...].numpy(), (batchsize, 1, 1, 1))
            batch_recon_const_j = np.tile(
                batch_recon_const[j, ...].numpy(), (batchsize, 1, 1, 1)
            )
            batch_recon_const_outres_j = np.tile(
                batch_recon_const_outres[j, ...].numpy(), (batchsize, 1, 1, 1)
            )

            input_color = torch.from_numpy(batch_j).cuda()
            input_greylevel = torch.from_numpy(batch_recon_const_j).cuda()

            # Get mean from GMM
            curr_mu = gmm_mu[j * nmix : (j + 1) * nmix, :]
            orderid = np.argsort(
                gmm_pi[j * nmix : (j + 1) * nmix, 0].cpu().data.numpy().reshape
    (-1)
            )

            # Sample from GMM
            z = curr_mu.repeat(int((batchsize * 1.0) / nmix), 1)

            # Predict color
            _, _, color_out = model_vae(input_color, input_greylevel, z)

            # Save images
            data.saveoutput_gt(
                color_out.cpu().data.numpy()[orderid, ...],
                batch_j[orderid, ...],
                "divcolor_%05d_%05d" % (batch_idx, j),
                nmix,
                net_recon_const=batch_recon_const_outres_j[orderid, ...],
            )

    print("Complete inference")

vae_ckpt = "model_vae.pth"
mdn_ckpt = "model_mdn.pth"
inference(vae_ckpt, mdn_ckpt)
```

Kết quả thực nghiệm mô hình sau khi huấn luyện

Hình 9: Kết quả thực nghiệm mô hình sau khi huấn luyện.

Phần III: Câu hỏi trắc nghiệm

1. VAE có thể được sử dụng trong các ứng dụng nào?

(*a*) Tô màu cho ảnh xám

(*b*) Nén ảnh

(*c*) Sinh ảnh mới

(*d*) Tất cả các phương án trên

2. Mô hình VAE cơ bản có bao nhiêu block chính?

(*a*) 1

(*b*) 2

(*c*) 3

(*d*) 4

3. Trong VAE, đối tượng cần được mã hóa được biểu diễn như thế nào?

(*a*) Dưới dạng một giá trị số thực.

(*b*) Dưới dạng một phân phối xác suất

(*c*) Dưới dạng một véc-tơ nhị phân

(*d*) Dưới dạng một ma trận đặc trưng

4. Trong VAE, khi huấn luyện mô hình, ta muốn KL Divergence Loss đạt giá trị bằng bao nhiêu?

(*a*) 0

(*b*) 1

(*c*) Không có giá trị nhất định

(*d*) Càng lớn càng tốt

5. Trong Image Colorization, vì sao không gian màu Lab thường được sử dụng hơn không gian màu RGB (chọn phương án đúng nhất)?

(*a*) **Phân biệt rõ ràng giữa độ sáng và màu sắc**: Không gian màu Lab phân chia màu sắc và độ sáng thành hai kênh riêng biệt (L, a, và b), giúp mô hình tập trung vào việc tái tạo màu sắc một cách chính xác hơn. Trong khi đó, ảnh RGB có thể làm mất thông tin về độ sáng khi thêm màu vào, gây ra hiệu ứng không mong muốn.

(*b*) **Khả năng lưu trữ thông tin màu sắc chi tiết**: Các giá trị trong không gian màu Lab là liên tục, ngược lại với các giá trị rời rạc trong không gian màu RGB. Điều này dẫn đến việc không gian màu Lab có khả năng lưu trữ lớn hơn và chính xác hơn trong việc biểu diễn hình ảnh

(*c*) **Độ phức tạp thấp hơn khi dự đoán màu sắc**: Với không gian màu Lab, chúng ta chỉ cần dự đoán hai kênh màu a và b, thay vì cả ba kênh màu như trong không gian màu RGB. Điều này giúp giảm độ phức tạp của bài toán và tăng hiệu suất của mô hình.

(*d*) Tất cả đáp án trên.

6. Trong quá trình inference, mô hình CVAE có sự tham gia của những thành phần nào?

(*a*) Encoder, Conditional Encoder, Decoder

(*b*) Encoder, Decoder

(*c*) Conditional Encoder, Decoder

(*d*) Decoder

- Hết -