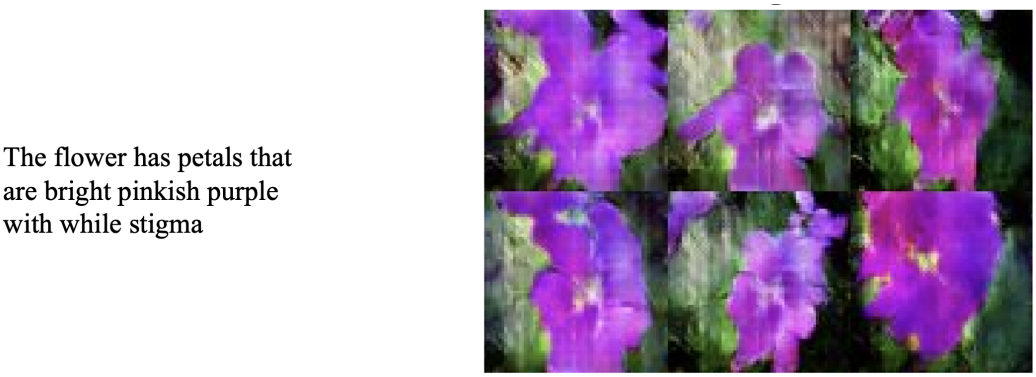
AI VIET NAM – AIO COURSE 2023

Exercise: Text to Image Synthesis using DCGAN

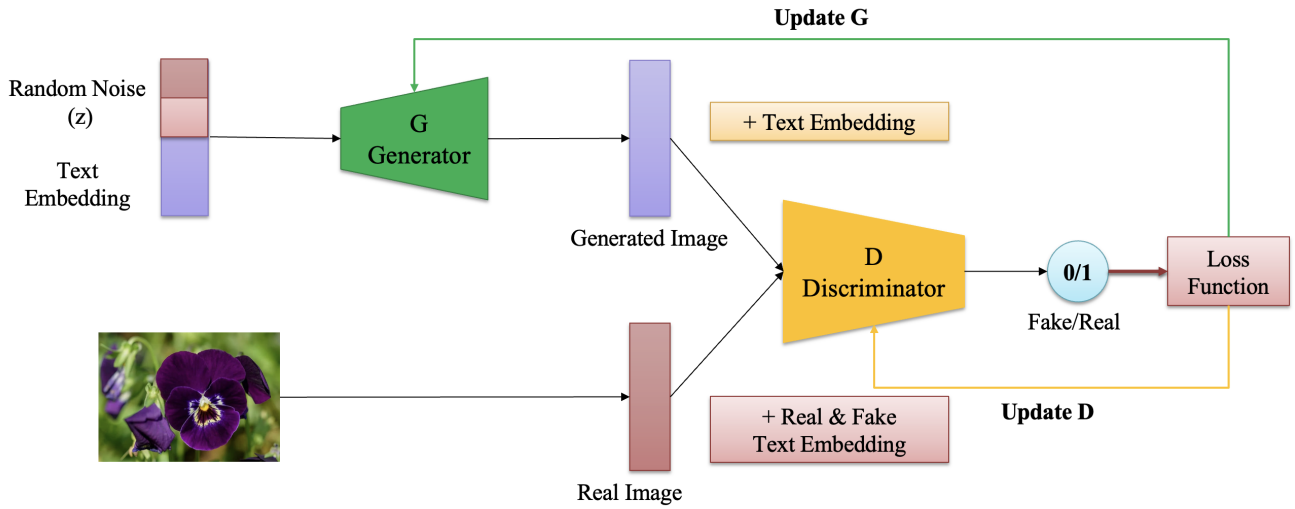
Quoc-Thai Nguyen và Quang-Vinh Dinh
PR-Team: Đăng-Nhã Nguyễn, Minh-Châu Phạm và Hoàng-Nguyên Vũ

Ngày 17 tháng 3 năm 2024

Phần I. Giới thiệu



Hình 1: Ví dụ sinh văn bản thành hình ảnh sử dụng mô hình DCGAN.



Hình 2: Mô hình DCGAN sinh văn bản thành hình ảnh.

Sinh hình ảnh từ văn bản hay tổng hợp văn bản thành hình ảnh (Text to Image) là bài toán ngày càng được ứng dụng manh mẽ trong lĩnh vực trí tuệ nhân tạo. Các mô hình được huấn luyện với đầu vào là đoạn văn bản và đầu ra là hình ảnh mô tả hoặc chứa các đối tượng được mô tả trong đoạn văn bản. Ví dụ về tổng hợp hình ảnh từ văn bản được mô tả trong Hình 1.

Hiện nay, có nhiều mô hình có thể xây dựng và giải quyết bài toán này có thể kể đến như GANs, Diffusion Models,... Ở trong phần này chúng ta sẽ sử dụng mô hình DCGAN (Deep Convolution Generative Adversarial Networks) để huấn luyện mô hình giải quyết bài toán này.

Mô hình DCGAN gồm 2 mạng là Generator và Discriminator. Trong đó:

1. Generator: Bao gồm các lớp mạng CNN nhận đầu vào là Noise (z) và Text Embedding (e - Biểu diễn của đoạn văn bản đầu vào). Đầu ra của khối Generator là ma trận ảnh được sinh ra G(z, e).

2. Discriminator: Bao gồm các lớp mạng CNN nhận đầu vào là biểu diễn của ảnh (ảnh thật hoặc ảnh fake được sinh ra từ khối Generator) sau đó kết hợp với Text Embedding để dự đoán ảnh thật hay ảnh fake.

Phần II. Text To Image Synthesis using DCGAN

Trong phần này chúng ta sẽ xây dựng và huấn luyện mô hình DCGAN trên bộ dữ liệu flowers. Bộ dữ liệu flowers bao gồm:

(a) Thư mục chứa hình ảnh: link

(b) Thư mục chứa các đoạn văn bản mô tả hình ảnh: link

(c) Thư mục chứa các vector biểu diễn cho các đoạn văn bản mô tả hình ảnh: link

1. Data Preparing

```
# install libs
!pip install -q torchfile

# import libs
import os
import h5py
import torchfile
from glob import glob

# read dataset and save h5py file
images_path = './data/'
embedding_path = './data/flowers_icml/'
text_path = './data/cvpr2016_flowers/text_c10/'
save_dataset_path = './data/flowers.hdf5'

train_path = './data/flowers_icml/trainclasses.txt'
val_path = './data/flowers_icml/valclasses.txt'
test_path = './data/flowers_icml/testclasses.txt'

train_classes = open(train_path).read().splitlines()
val_classes = open(val_path).read().splitlines()
test_classes = open(test_path).read().splitlines()

f = h5py.File(save_dataset_path, 'w')
train = f.create_group('train')
val = f.create_group('val')
test = f.create_group('test')

for _class in sorted(os.listdir(embedding_path)):
    split = ''
    if _class in train_classes:
        split = train
    elif _class in val_classes:
        split = val
    elif _class in test_classes:
        split = test

    data_path = os.path.join(embedding_path, _class)
    txt_path = os.path.join(text_path, _class)
    for example, txt_file in zip(sorted(glob(data_path + "/*.t7")), sorted(glob(
    txt_path + "/*.txt"))):
        example_data = torchfile.load(example)
        img_path = example_data[b'img'].decode("utf-8")
```

```
        embeddings = example_data[b'txt']
        example_name = img_path.split('/')[-1][:-4]

        f = open(txt_file, "r")
        txt = f.readlines()
        f.close()

        img_path = os.path.join(images_path, img_path)
        img = open(img_path, 'rb').read()

        print(example_name, txt)
        txt_choice = np.random.choice(range(10), 5)

        embeddings = embeddings[txt_choice]
        txt = np.array(txt)
        txt = txt[txt_choice]
        dt = h5py.special_dtype(vlen=str)

        for c, e in enumerate(embeddings):
            ex = split.create_group(example_name + '_' + str(c))
            ex.create_dataset('name', data=example_name)
            ex.create_dataset('img', data=np.void(img))
            ex.create_dataset('embeddings', data=e)
            ex.create_dataset('class', data=_class)
            ex.create_dataset('txt', data=txt[c].astype(object), dtype=dt)

        print(example_name, txt[1], _class)
```

2. Dataset

```
import io
import torch
import numpy as np
from PIL import Image
from torch.utils.data import Dataset

class Text2ImageDataset(Dataset):

    def __init__(self, dataset_path, transform=None, split=0):
        self.dataset_path = dataset_path
        self.transform = transform
        self.dataset = h5py.File(self.dataset_path, mode='r')
        self.split = 'train' if split == 0 else 'val' if split == 1 else 'test'
        self.dataset_keys = [str(k) for k in self.dataset[self.split].keys()]
        self.h5py2int = lambda x: int(np.array(x))

    def __len__(self):
        length = len(self.dataset[self.split])
        return length

    def __getitem__(self, idx):
        example_name = self.dataset_keys[idx]
        example = self.dataset[self.split][example_name]

        right_image = bytes(np.array(example['img']))
        right_embed = np.array(example['embeddings'], dtype=float)
        wrong_image = bytes(np.array(self.find_wrong_image(example['class'])))

        right_image = Image.open(io.BytesIO(right_image)).resize((64, 64))
        wrong_image = Image.open(io.BytesIO(wrong_image)).resize((64, 64))
```

```
31
32          right_image = self.validate_image(right_image)
33          wrong_image = self.validate_image(wrong_image)
34
35          try:
36              txt = np.array(example['txt']).astype(str)
37          except:
38
39              txt = np.array([example['txt'][()].decode('utf-8', errors='replace')])
40              txt = np.char.replace(txt, ' í ', ' ').astype(str)
41
42          sample = {
43                  'right_images': torch.FloatTensor(right_image),
44                  'right_embed': torch.FloatTensor(right_embed),
45                  'wrong_images': torch.FloatTensor(wrong_image),
46                  'txt': str(txt)
47                   }
48
49          sample['right_images'] = sample['right_images'].sub_(127.5).div_(127.5)
50          sample['wrong_images'] =sample['wrong_images'].sub_(127.5).div_(127.5)
51
52          return sample
53
54      def find_wrong_image(self, category):
55          idx = np.random.randint(len(self.dataset_keys))
56          example_name = self.dataset_keys[idx]
57          example = self.dataset[self.split][example_name]
58          _category = example['class']
59
60          if _category != category:
61              return example['img']
62
63          return self.find_wrong_image(category)
64
65
66      def validate_image(self, img):
67          img = np.array(img, dtype=float)
68          return img.transpose(2, 0, 1)
69
70  train_dataset = Text2ImageDataset(save_dataset_path, split=0)
71  val_dataset = Text2ImageDataset(save_dataset_path, split=1)
72  test_dataset = Text2ImageDataset(save_dataset_path, split=2)
73
74  # dataloader
75  from torch.utils.data import DataLoader
76
77  batch_size = 2048
78  train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
79  val_loader = DataLoader(val_dataset, batch_size=batch_size)
80  test_loader = DataLoader(test_dataset, batch_size=batch_size)
```

3. Model

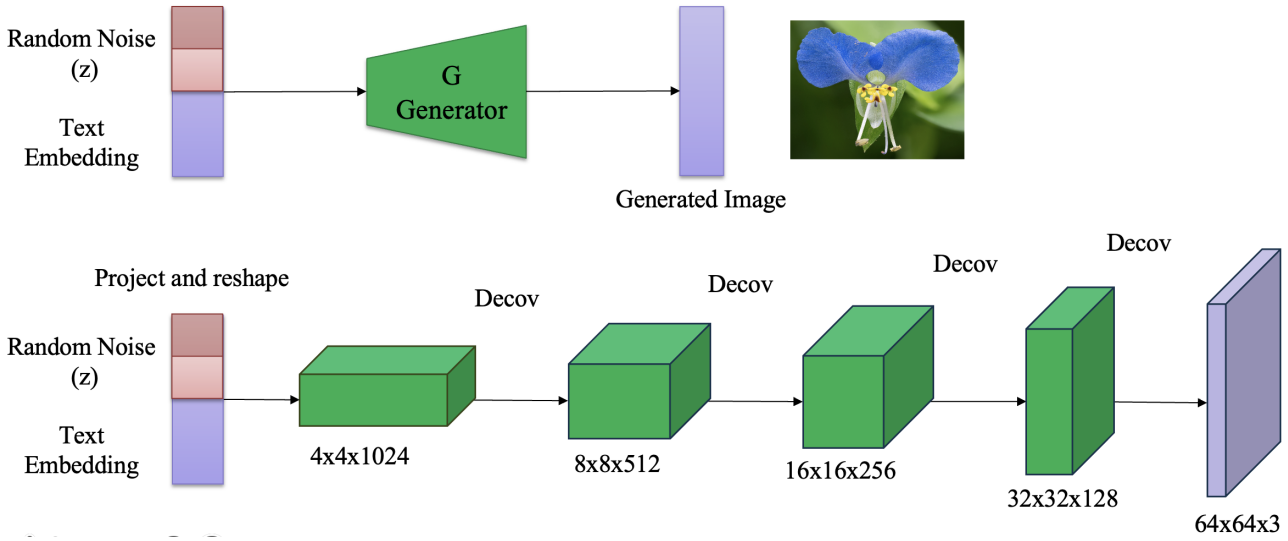
Trong phần này chúng ta xây dựng mô hình DCGAN bao gồm 2 mạng Generator và Discriminator:

3.1. Generator

Khối Generator sinh ra ảnh từ văn bản đầu vào:

(a) Input: Nhận đầu vào là vector: Random Noise (z) có kích thước R, được nối với Vector Embedding (Vector e - biểu diễn cho cả đoạn văn bản đầu vào) có kích thước là D. Vì vậy, vector đầu vào mạng Generator là R + D.

(b) Output: Sau khi học mối quan hệ để sinh ảnh, giá trị đầu ra của Generator sẽ là biểu diễn các điểm ảnh dự đoán trong không gian 3 chiều có kích thước là Channel x Width x Height (Ví dụ, CxWxH - 3x64x64)



Hình 3: Khối Generator trong DCGAN.

```python
import torch
import torch.nn as nn

# The Generator model
class Generator(nn.Module):
    def __init__(self, channels, noise_dim=100, embed_dim=1024, embed_out_dim=128):
        super(Generator, self).__init__()
        self.channels = channels
        self.noise_dim = noise_dim
        self.embed_dim = embed_dim
        self.embed_out_dim = embed_out_dim

        # Text embedding layers
        self.text_embedding = nn.Sequential(
            nn.Linear(self.embed_dim, self.embed_out_dim),
            nn.BatchNorm1d(self.embed_out_dim),
            nn.LeakyReLU(0.2, inplace=True)
        )

        # Generator architecture
        model = []
```

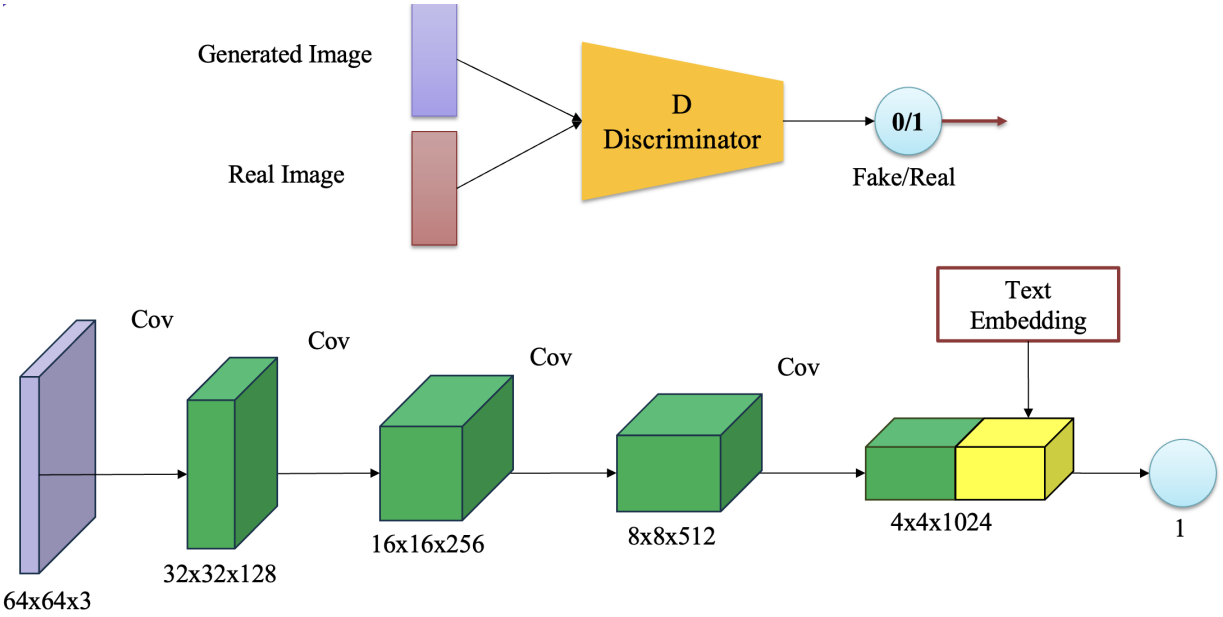
```
        model += self._create_layer(self.noise_dim + self.embed_out_dim, 512, 4, stride=1, padding=0)
        model += self._create_layer(512, 256, 4, stride=2, padding=1)
        model += self._create_layer(256, 128, 4, stride=2, padding=1)
        model += self._create_layer(128, 64, 4, stride=2, padding=1)
        model += self._create_layer(64, self.channels, 4, stride=2, padding=1, output=True)

        self.model = nn.Sequential(*model)

    def _create_layer(self, size_in, size_out, kernel_size=4, stride=2, padding=1, output=False):
        layers = [nn.ConvTranspose2d(size_in, size_out, kernel_size, stride=stride, padding=padding, bias=False)]
        if output:
            layers.append(nn.Tanh())  # Tanh activation for the output layer
        else:
            layers += [nn.BatchNorm2d(size_out), nn.ReLU(True)]  # Batch normalization and ReLU for other layers
        return layers

    def forward(self, noise, text):
        # Apply text embedding to the input text
        text = self.text_embedding(text)
        text = text.view(text.shape[0], text.shape[1], 1, 1)  # Reshape to match the generator input size
        z = torch.cat([text, noise], 1)  # Concatenate text embedding with noise
        return self.model(z)

def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        m.weight.data.normal_(0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        m.weight.data.normal_(1.0, 0.02)
        m.bias.data.fill_(0)

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
embed_dim = 1024
noise_dim = 100
embed_out_dim = 64
generator = Generator(
    channels=3, embed_dim=embed_dim, noise_dim=noise_dim, embed_out_dim=embed_out_dim
).to(device)
generator.apply(weights_init)
```

3.2. Discriminator

Khối Discriminator dự đoán hình ảnh Fake/Real, bao gồm các lớp CNN với:

(a) Input: Nhận đầu vào là ma trận điểm ảnh và Text Embedding. Ma trận điểm ảnh (CxHxW) sau khi qua các lớp CNN để học các đặc trưng của ảnh đầu vào sẽ được nối với vector embedding.

(b) Output: Vector sau khi được nối sẽ được sử dụng để dự đoán 0(Fake) hoặc 1(Real).



Hình 4: Khối Discriminator trong DCGAN.

- Đầu tiên chúng ta sẽ định nghĩa lớp Embedding để nối các đặc trưng của ảnh và Text Embedding.

```
# The Embedding model
class Embedding(nn.Module):
    def __init__(self, size_in, size_out):
        super(Embedding, self).__init__()
        self.text_embedding = nn.Sequential(
            nn.Linear(size_in, size_out),
            nn.BatchNorm1d(size_out),
            nn.LeakyReLU(0.2, inplace=True)
        )

    def forward(self, x, text):
        embed_out = self.text_embedding(text)
        embed_out_resize = embed_out.repeat(4, 4, 1, 1).permute(2, 3, 0, 1)  # Resize to match the discriminator input size
        out = torch.cat([x, embed_out_resize], 1)  # Concatenate text embedding with the input feature map
        return out
```

- Sau đó, chúng ta định nghĩa kiến trúc mạng Discriminator:

```
# The Discriminator model
class Discriminator(nn.Module):
    def __init__(self, channels, embed_dim=1024, embed_out_dim=128):
        super(Discriminator, self).__init__()
        self.channels = channels
        self.embed_dim = embed_dim
        self.embed_out_dim = embed_out_dim

        # Discriminator architecture
        self.model = nn.Sequential(
            *self._create_layer(self.channels, 64, 4, 2, 1, normalize=False),
            *self._create_layer(64, 128, 4, 2, 1),
            *self._create_layer(128, 256, 4, 2, 1),
            *self._create_layer(256, 512, 4, 2, 1)
        )
```

```
        self.text_embedding = Embedding(self.embed_dim, self.embed_out_dim)  # Text embedding module
        self.output = nn.Sequential(
            nn.Conv2d(512 + self.embed_out_dim, 1, 4, 1, 0, bias=False), nn.Sigmoid()
        )

    def _create_layer(self, size_in, size_out, kernel_size=4, stride=2, padding=1, normalize=True):
        layers = [nn.Conv2d(size_in, size_out, kernel_size=kernel_size, stride=stride, padding=padding)]
        if normalize:
            layers.append(nn.BatchNorm2d(size_out))
        layers.append(nn.LeakyReLU(0.2, inplace=True))
        return layers

    def forward(self, x, text):
        x_out = self.model(x)  # Extract features from the input using the discriminator architecture
        out = self.text_embedding(x_out, text)  # Apply text embedding and concatenate with the input features
        out = self.output(out)  # Final discriminator output
        return out.squeeze(), x_out

discriminator = Discriminator(
    channels=3, embed_dim=embed_dim, embed_out_dim=embed_out_dim
).to(device)
discriminator.apply(weights_init)
```

4. Training

```
# setting up Adam optimizer for Generator and Discriminator
learning_rate = 0.0002
optimizer_G = torch.optim.Adam(
    generator.parameters(), lr=learning_rate, betas=(0.5, 0.999)
)
optimizer_D = torch.optim.Adam(
    discriminator.parameters(), lr=learning_rate, betas=(0.5, 0.999)
)

# loss functions
criterion = nn.BCELoss()
l2_loss = nn.MSELoss()
l1_loss = nn.L1Loss()

num_epochs = 20
real_label = 1.
fake_label = 0.
l1_coef = 50
l2_coef = 100

D_losses = []
G_losses = []

for epoch in range(num_epochs):
    epoch_D_loss = []
    epoch_G_loss = []
    batch_time = time.time()

    for idx, batch in enumerate(train_loader):

```

```
        images = batch['right_images'].to(device)
        wrong_images = batch['wrong_images'].to(device)
        embeddings = batch['right_embed'].to(device)
        batch_size = images.size(0)

        # ======================================================================== #
        #                        Train the discriminator                           #
        # ======================================================================== #

        # Clear gradients for the discriminator
        optimizer_D.zero_grad()

        # Generate random noise
        noise = torch.randn(batch_size, noise_dim, 1, 1, device=device)

        # Generate fake image batch with the generator
        fake_images = generator(noise, embeddings)

        # Forward pass real batch and calculate loss
        real_out, real_act = discriminator(images, embeddings)
        d_loss_real = criterion(real_out, torch.full_like(real_out, real_label, device
    =device))

        # Forward pass wrong batch and calculate loss
        wrong_out, wrong_act = discriminator(wrong_images, embeddings)
        d_loss_wrong = criterion(wrong_out, torch.full_like(wrong_out, fake_label,
    device=device))

        # Forward pass fake batch and calculate loss
        fake_out, fake_act = discriminator(fake_images.detach(), embeddings)
        d_loss_fake = criterion(fake_out, torch.full_like(fake_out, fake_label, device
    =device))

        # Compute total discriminator loss
        d_loss = d_loss_real + d_loss_wrong + d_loss_fake

        # Backpropagate the gradients
        d_loss.backward()

        # Update the discriminator
        optimizer_D.step()

        # ======================================================================== #
        #                          Train the generator                             #
        # ======================================================================== #

        # Clear gradients for the generator
        optimizer_G.zero_grad()

        # Generate new random noise
        noise = torch.randn(batch_size, noise_dim, 1, 1, device=device)

        # Generate new fake images using Generator
        fake_images = generator(noise, embeddings)

        # Get discriminator output for the new fake images
        out_fake, act_fake = discriminator(fake_images, embeddings)

        # Get discriminator output for the real images
        out_real, act_real = discriminator(images, embeddings)
```

```
88
89          # Calculate losses
90          g_bce = criterion(out_fake, torch.full_like(out_fake, real_label, device=
    device))
91          g_l1 = l1_coef * l1_loss(fake_images, images)
92          g_l2 = l2_coef * l2_loss(torch.mean(act_fake, 0), torch.mean(act_real, 0).
    detach())
93
94          # Compute total generator loss
95          g_loss = g_bce + g_l1 + g_l2
96
97          # Backpropagate the gradients
98          g_loss.backward()
99
100         # Update the generator
101         optimizer_G.step()
102
103         epoch_D_loss.append(d_loss.item())
104         epoch_G_loss.append(g_loss.item())
105
106     print('Epoch {} [{}/{}] loss_D: {:.4f} loss_G: {:.4f} time: {:.2f}'.format(
107         epoch+1, idx+1, len(train_loader),
108         d_loss.mean().item(),
109         g_loss.mean().item(),
110         time.time() - batch_time)
111     )
112     D_losses.append(sum(epoch_D_loss)/len(epoch_D_loss))
113     G_losses.append(sum(epoch_G_loss)/len(epoch_G_loss))
114
115 # save model
116 model_save_path = './save_model'
117 torch.save(generator.state_dict(), os.path.join(model_save_path, 'generator.pth'))
118 torch.save(discriminator.state_dict(), os.path.join(model_save_path,'discriminator.pth
    '))
```

5. Prediction

```
1  import matplotlib.pyplot as plt
2
3  example = next(iter(test_dataset))
4
5  # show real image
6  plt.imshow(example['right_images'].permute(1, 2, 0))
7  plt.show()
8
9  # prediction
10 embeddings = example['right_embed'].to(device)
11 noise = torch.randn(1, noise_dim, 1, 1, device=device)
12 pred = generator(noise, embeddings.unsqueeze(0))
13
14 # show the prediction image
15 plt.imshow(pred[0].cpu().detach().permute(1, 2, 0))
16 plt.show()
```

Phần III. Text To Image Synthesis using DCGAN and BERTs

Trong phần này chúng ta sẽ cải tiến mô hình tạo hình ảnh từ văn bản bằng cách sử dụng các mô hình ngôn ngữ giàu ngữ cảnh hơn để biểu diễn văn bản thành các vector như BERTs,...

1. Data Preparing

```
images_path = './data/'
text_path = './data/cvpr2016_flowers/text_c10/'
save_dataset_path = './data/flowers.hdf5' # from section 2

# load BERTs
import torch
from transformers import AutoTokenizer, DistilBertModel

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased")
model = DistilBertModel.from_pretrained("distilbert-base-uncased")
model.to(device)
model.eval()

# Max Pooling - Take the max value over time for every dimension.
def max_pooling(model_output, attention_mask):
    token_embeddings = model_output[0] #First element of model_output contains all token embeddings
    input_mask_expanded = attention_mask.unsqueeze(-1).expand(token_embeddings.size()).float()
    token_embeddings[input_mask_expanded == 0] = -1e9  # Set padding tokens to large negative value
    return torch.max(token_embeddings, 1)[0]

def convert_text_to_feature(sentences, max_length=50):
    inputs = tokenizer.batch_encode_plus(
        sentences, padding='max_length', max_length=max_length, truncation=True, return_tensors='pt'
    )
    input_ids = inputs['input_ids'].to(device)
    attention_mask = inputs['attention_mask'].to(device)
    outputs = model(input_ids=input_ids, attention_mask=attention_mask)
    sentence_embeddings = max_pooling(outputs, attention_mask)
    return sentence_embeddings
```

2. Dataset

```
import io
import torch
import numpy as np
from PIL import Image
from torch.utils.data import Dataset

class Text2ImageDataset(Dataset):

    def __init__(self, dataset_path, transform=None, split=0):
        self.dataset_path = dataset_path
        self.transform = transform
        self.dataset = h5py.File(self.dataset_path, mode='r')
```

```
        self.split = 'train' if split == 0 else 'val' if split == 1 else 'test'
        self.dataset_keys = [str(k) for k in self.dataset[self.split].keys()]
        self.h5py2int = lambda x: int(np.array(x))

    def __len__(self):
        length = len(self.dataset[self.split])
        return length

    def __getitem__(self, idx):
        example_name = self.dataset_keys[idx]
        example = self.dataset[self.split][example_name]

        right_image = bytes(np.array(example['img']))
        right_embed = np.array(example['embeddings'], dtype=float)
        wrong_image = bytes(np.array(self.find_wrong_image(example['class'])))

        right_image = Image.open(io.BytesIO(right_image)).resize((64, 64))
        wrong_image = Image.open(io.BytesIO(wrong_image)).resize((64, 64))

        right_image = self.validate_image(right_image)
        wrong_image = self.validate_image(wrong_image)

        try:
            txt = np.array(example['txt']).astype(str)
        except:

            txt = np.array([example['txt'][()].decode('utf-8', errors='replace')])
            txt = np.char.replace(txt, ' í ', ' ').astype(str)

        embeddings = convert_text_to_feature([str(txt)]).detach().cpu()

        sample = {
                'right_images': torch.FloatTensor(right_image),
                'right_embed': embeddings[0],
                'wrong_images': torch.FloatTensor(wrong_image),
                'txt': str(txt)
                 }

        sample['right_images'] = sample['right_images'].sub_(127.5).div_(127.5)
        sample['wrong_images'] =sample['wrong_images'].sub_(127.5).div_(127.5)

        return sample

    def find_wrong_image(self, category):
        idx = np.random.randint(len(self.dataset_keys))
        example_name = self.dataset_keys[idx]
        example = self.dataset[self.split][example_name]
        _category = example['class']

        if _category != category:
            return example['img']

        return self.find_wrong_image(category)

    def validate_image(self, img):
        img = np.array(img, dtype=float)
        return img.transpose(2, 0, 1)

# dataset
train_dataset = Text2ImageDataset(save_dataset_path, split=0)
```

```
val_dataset = Text2ImageDataset(save_dataset_path, split=1)
test_dataset = Text2ImageDataset(save_dataset_path, split=2)

# dataloader
from torch.utils.data import DataLoader

batch_size = 2048
train_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=batch_size)
test_loader = DataLoader(test_dataset, batch_size=batch_size)
```

3. Model

```
import torch
import torch.nn as nn

# The Generator model
class Generator(nn.Module):
    def __init__(self, channels, noise_dim=100, embed_dim=1024, embed_out_dim=128):
        super(Generator, self).__init__()
        self.channels = channels
        self.noise_dim = noise_dim
        self.embed_dim = embed_dim
        self.embed_out_dim = embed_out_dim

        # Text embedding layers
        self.text_embedding = nn.Sequential(
            nn.Linear(self.embed_dim, self.embed_out_dim),
            nn.BatchNorm1d(self.embed_out_dim),
            nn.LeakyReLU(0.2, inplace=True)
        )

        # Generator architecture
        model = []
        model += self._create_layer(self.noise_dim + self.embed_out_dim, 512, 4,
    stride=1, padding=0)
        model += self._create_layer(512, 256, 4, stride=2, padding=1)
        model += self._create_layer(256, 128, 4, stride=2, padding=1)
        model += self._create_layer(128, 64, 4, stride=2, padding=1)
        model += self._create_layer(64, self.channels, 4, stride=2, padding=1, output=
    True)

        self.model = nn.Sequential(*model)

    def _create_layer(self, size_in, size_out, kernel_size=4, stride=2, padding=1,
    output=False):
        layers = [nn.ConvTranspose2d(size_in, size_out, kernel_size, stride=stride,
    padding=padding, bias=False)]
        if output:
            layers.append(nn.Tanh())  # Tanh activation for the output layer
        else:
            layers += [nn.BatchNorm2d(size_out), nn.ReLU(True)]  # Batch normalization
     and ReLU for other layers
        return layers

    def forward(self, noise, text):
        # Apply text embedding to the input text
        text = self.text_embedding(text)
        text = text.view(text.shape[0], text.shape[1], 1, 1)  # Reshape to match the
    generator input size
```

```
        z = torch.cat([text, noise], 1)  # Concatenate text embedding with noise
        return self.model(z)


# The Embedding model
class Embedding(nn.Module):
    def __init__(self, size_in, size_out):
        super(Embedding, self).__init__()
        self.text_embedding = nn.Sequential(
            nn.Linear(size_in, size_out),
            nn.BatchNorm1d(size_out),
            nn.LeakyReLU(0.2, inplace=True)
        )

    def forward(self, x, text):
        embed_out = self.text_embedding(text)
        embed_out_resize = embed_out.repeat(4, 4, 1, 1).permute(2, 3, 0, 1)  # Resize to match the discriminator input size
        out = torch.cat([x, embed_out_resize], 1)  # Concatenate text embedding with the input feature map
        return out


# The Discriminator model
class Discriminator(nn.Module):
    def __init__(self, channels, embed_dim=1024, embed_out_dim=128):
        super(Discriminator, self).__init__()
        self.channels = channels
        self.embed_dim = embed_dim
        self.embed_out_dim = embed_out_dim

        # Discriminator architecture
        self.model = nn.Sequential(
            *self._create_layer(self.channels, 64, 4, 2, 1, normalize=False),
            *self._create_layer(64, 128, 4, 2, 1),
            *self._create_layer(128, 256, 4, 2, 1),
            *self._create_layer(256, 512, 4, 2, 1)
        )
        self.text_embedding = Embedding(self.embed_dim, self.embed_out_dim)  # Text embedding module
        self.output = nn.Sequential(
            nn.Conv2d(512 + self.embed_out_dim, 1, 4, 1, 0, bias=False), nn.Sigmoid()
        )

    def _create_layer(self, size_in, size_out, kernel_size=4, stride=2, padding=1, normalize=True):
        layers = [nn.Conv2d(size_in, size_out, kernel_size=kernel_size, stride=stride, padding=padding)]
        if normalize:
            layers.append(nn.BatchNorm2d(size_out))
        layers.append(nn.LeakyReLU(0.2, inplace=True))
        return layers

    def forward(self, x, text):
        x_out = self.model(x)  # Extract features from the input using the discriminator architecture
        out = self.text_embedding(x_out, text)  # Apply text embedding and concatenate with the input features
        out = self.output(out)  # Final discriminator output
        return out.squeeze(), x_out
```

```
95
96  def weights_init(m):
97      classname = m.__class__.__name__
98      if classname.find('Conv') != -1:
99          m.weight.data.normal_(0.0, 0.02)
100     elif classname.find('BatchNorm') != -1:
101         m.weight.data.normal_(1.0, 0.02)
102         m.bias.data.fill_(0)
103
104 device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
105 embed_dim = 1024
106 noise_dim = 100
107 embed_out_dim = 64
108 generator = Generator(
109     channels=3, embed_dim=embed_dim, noise_dim=noise_dim, embed_out_dim=embed_out_dim
110 ).to(device)
111 generator.apply(weights_init)
112
113 discriminator = Discriminator(
114     channels=3, embed_dim=embed_dim, embed_out_dim=embed_out_dim
115 ).to(device)
116 discriminator.apply(weights_init)
```

4. Training

```
1  # setting up Adam optimizer for Generator and Discriminator
2  learning_rate = 0.0002
3  optimizer_G = torch.optim.Adam(
4      generator.parameters(), lr=learning_rate, betas=(0.5, 0.999)
5  )
6  optimizer_D = torch.optim.Adam(
7      discriminator.parameters(), lr=learning_rate, betas=(0.5, 0.999)
8  )
9
10 # loss functions
11 criterion = nn.BCELoss()
12 l2_loss = nn.MSELoss()
13 l1_loss = nn.L1Loss()
14
15 num_epochs = 20
16 real_label = 1.
17 fake_label = 0.
18 l1_coef = 50
19 l2_coef = 100
20
21 D_losses = []
22 G_losses = []
23
24 for epoch in range(num_epochs):
25     epoch_D_loss = []
26     epoch_G_loss = []
27     batch_time = time.time()
28
29     for idx, batch in enumerate(train_loader):
30
31         images = batch['right_images'].to(device)
32         wrong_images = batch['wrong_images'].to(device)
33         embeddings = batch['right_embed'].to(device)
34         batch_size = images.size(0)
35
```

```
        # ================================================================== #
        #                        Train the discriminator                     #
        # ================================================================== #

        # Clear gradients for the discriminator
        optimizer_D.zero_grad()

        # Generate random noise
        noise = torch.randn(batch_size, noise_dim, 1, 1, device=device)

        # Generate fake image batch with the generator
        fake_images = generator(noise, embeddings)

        # Forward pass real batch and calculate loss
        real_out, real_act = discriminator(images, embeddings)
        d_loss_real = criterion(real_out, torch.full_like(real_out, real_label, device
    =device))

        # Forward pass wrong batch and calculate loss
        wrong_out, wrong_act = discriminator(wrong_images, embeddings)
        d_loss_wrong = criterion(wrong_out, torch.full_like(wrong_out, fake_label,
    device=device))

        # Forward pass fake batch and calculate loss
        fake_out, fake_act = discriminator(fake_images.detach(), embeddings)
        d_loss_fake = criterion(fake_out, torch.full_like(fake_out, fake_label, device
    =device))

        # Compute total discriminator loss
        d_loss = d_loss_real + d_loss_wrong + d_loss_fake

        # Backpropagate the gradients
        d_loss.backward()

        # Update the discriminator
        optimizer_D.step()

        # ================================================================== #
        #                          Train the generator                       #
        # ================================================================== #

        # Clear gradients for the generator
        optimizer_G.zero_grad()

        # Generate new random noise
        noise = torch.randn(batch_size, noise_dim, 1, 1, device=device)

        # Generate new fake images using Generator
        fake_images = generator(noise, embeddings)

        # Get discriminator output for the new fake images
        out_fake, act_fake = discriminator(fake_images, embeddings)

        # Get discriminator output for the real images
        out_real, act_real = discriminator(images, embeddings)

        # Calculate losses
        g_bce = criterion(out_fake, torch.full_like(out_fake, real_label, device=
    device))
        g_l1 = l1_coef * l1_loss(fake_images, images)
```

```
        g_l2 = l2_coef * l2_loss(torch.mean(act_fake, 0), torch.mean(act_real, 0).detach())

        # Compute total generator loss
        g_loss = g_bce + g_l1 + g_l2

        # Backpropagate the gradients
        g_loss.backward()

        # Update the generator
        optimizer_G.step()

        epoch_D_loss.append(d_loss.item())
        epoch_G_loss.append(g_loss.item())

    print('Epoch {} [{}/{}] loss_D: {:.4f} loss_G: {:.4f} time: {:.2f}'.format(
        epoch+1, idx+1, len(train_loader),
        d_loss.mean().item(),
        g_loss.mean().item(),
        time.time() - batch_time)
    )
    D_losses.append(sum(epoch_D_loss)/len(epoch_D_loss))
    G_losses.append(sum(epoch_G_loss)/len(epoch_G_loss))

# save model
model_save_path = './save_model'
torch.save(generator.state_dict(), os.path.join(model_save_path, 'generator.pth'))
torch.save(discriminator.state_dict(), os.path.join(model_save_path,'discriminator.pth'))
```

5. Prediction

```
generator.eval()
example = next(iter(test_dataset))
sentence = example['txt'] # this flower consists of large yellow-green petals that are vertically oriented.
embeddings = convert_text_to_feature([str(sentence)])
noise = torch.randn(1, noise_dim, 1, 1, device=device)
pred = generator(noise, embeddings.detach())

plt.imshow(pred[0].cpu().detach().permute(1, 2, 0))
plt.show()
```

Phần 4. Câu hỏi trắc nghiệm

Câu hỏi 1 Mục đích của bài toán Text to Image là gì?
a) Tạo hình ảnh tương ứng với thông tin mô tả từ văn bản
b) Phân loại hình ảnh
c) Phân loại văn bản
d) Phát hiện cạnh của các đối tượng trong hình ảnh

Câu hỏi 2 Mô hình nào được xây dựng để giải quyết bài toán Text-to-Image?
a) GAN-CLS
b) BERT
c) ResNet
d) VGG

Câu hỏi 3 Khối Generator được sử dụng để làm gì?
a) Sinh hình ảnh mới từ Noise và Text Embedding
b) Dự đoán hình ảnh Real/Fake
c) Sinh hình ảnh mới từ hình ảnh cũ
d) Sinh văn bản mới từ Noise và Text Embedding

Câu hỏi 4 Khối Discriminator được sử dụng để làm gì?
a) Sử dụng kết hợp hình ảnh và Text Embedding dự đoán hình ảnh Fake/Real
b) Sử dụng Text Embedding dự đoán hình ảnh Fake/Real
c) Cả 2 đáp án trên đều đúng
d) Cả 2 đáp án trên đều sai

Câu hỏi 5 Kiến trúc của Generator và Discriminator không bao gồm layer nào sau đây?
a) CNN
b) Batch Normalization
c) LeakyReLU Activation
d) RNN

Câu hỏi 6 Số chiều biểu diễn văn bản được sử dụng trong phần thực nghiệm 2 là bao nhiêu?
a) 512
b) 1024
c) 768
d) 2048

Câu hỏi 7 Dựa vào code thực nghiệm trong phần 2, lớp Convolution cuối cùng trong khối Discriminator có tham số ‘in channels‘ là bao nhiêu?
a) 512

b) 64
c) 576
d) 1024

Câu hỏi 8 Hàm loss nào sau đây không được sử dụng trong phần thực nghiệm?
a) BCELoss
b) MSELoss
c) L1Loss
d) CTCLoss

Câu hỏi 9 Mô hình pre-trained language model nào được sử dụng trong thực nghiệm?
a) BERT
b) DistilBERT
c) RoBERTa
d) T5

Câu hỏi 10 Dựa vào phần code thực nghiệm trong phần 3, kích thước chiều embedding của text của mô hình pre-trained language model được sử dụng là?
a) 512
b) 768
c) 576
d) 1024

- Hết -