gans-monet-painting-project

July 31, 2025

1 GANs Monet Painting Project

1.1 Description of the Problem and Data

This analysis examines a Kaggle competition titled "I'm Something of a Painter Myself," in which the challenge is to train a generative model that converts ordinary photographs into images rendered in the style of Claude Monet. The available dataset comprises two distinct sets: Monet's paintings and photographic images. The Monet set includes roughly 300 high-resolution paintings, while the photo set contains several thousand images drawn from everyday scenes. The task is to build a generative adversarial network that learns the artistic style and produces novel Monet-style images without relying on paired examples.

The main goal is to develop a deep learning model that preserves scene content from the photo images but transforms their visual appearance so they evoke the impressionistic brushwork and color palette typical of Monet's work.

The data are provided in directories named monet_jpg and photo_jpg (and optionally monet_tfrec and photo_tfrec formats). The Monet images serve as the target artistic domain, and the photos represent the source domain. The challenge evaluation uses a specialized metric called memorization-informed Fréchet Inception Distance (MiFID), which adjusts FID scores to control for over-memorization of the training images.

In the notebook that follows the data are loaded, a generative adversarial network architecture is defined, models are trained, outputs are generated and scored, and findings are discussed.

[1]:
```python
import os
from PIL import Image
import matplotlib.pyplot as plt
import numpy as np
import random
import torch
from torch import nn
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
import torchvision.transforms as T
from torchvision.utils import save_image
import zipfile
```

[2]:
```python
base_path = '/kaggle/input/gan-getting-started'
monet_folder = os.path.join(base_path, 'monet_jpg')
```

```python
photo_folder = os.path.join(base_path, 'photo_jpg')

monet_files = os.listdir(monet_folder)
photo_files = os.listdir(photo_folder)

print('Number of Monet painting files:', len(monet_files))
print('Number of photo image files:', len(photo_files))

sample_monet = os.path.join(monet_folder, monet_files[0])
sample_photo = os.path.join(photo_folder, photo_files[0])

img_m = Image.open(sample_monet)
img_p = Image.open(sample_photo)

print('Sample Monet image size', img_m.size)
print('Sample photo image size', img_p.size)

fig, axes = plt.subplots(1, 2, figsize=(8, 4))
axes[0].imshow(img_m)
axes[0].axis('off')
axes[1].imshow(img_p)
axes[1].axis('off')
plt.tight_layout()
plt.show()
```

```
Number of Monet painting files: 300
Number of photo image files: 7038
Sample Monet image size (256, 256)
Sample photo image size (256, 256)
```

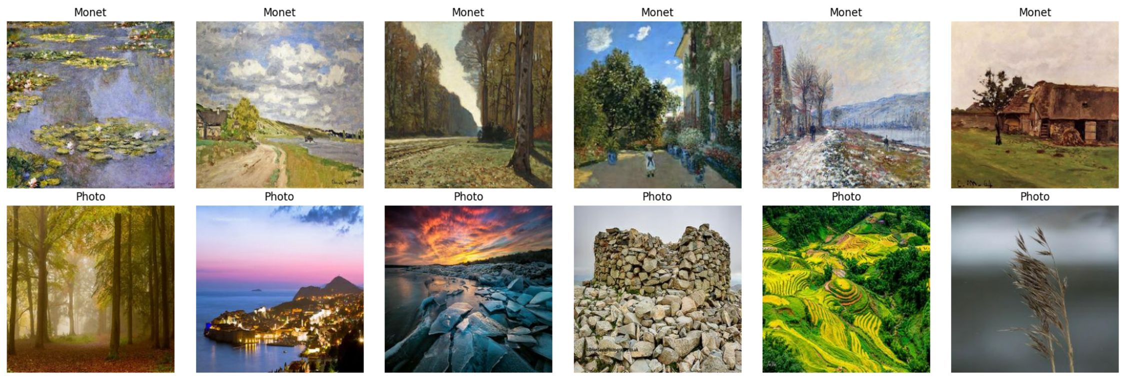
1.2 EDA procedure

Before building any models, it is important to become familiar with the data. This section explores the structure, distribution, and characteristics of the images provided in both the Monet and photo folders. Since these are unpaired image datasets, the focus is on understanding resolution, format, quantity, and any noticeable patterns in color or texture that might influence how a generative model learns the translation between domains.

Sample images are displayed to give a visual sense of the artistic differences. The pixel value ranges are also checked to confirm preprocessing needs. In addition, a few Monet images are plotted side by side to observe style consistency and brushwork. Similarly, photo samples are viewed for lighting, subject variety, and realism.

This basic inspection helps verify that the inputs are clean and consistent and provides early visual cues about the learning task.

[3]:
```python
def load_image(path):
    return np.array(Image.open(path))

monet_paths = [os.path.join(monet_folder, f) for f in monet_files[:6]]
photo_paths = [os.path.join(photo_folder, f) for f in photo_files[:6]]

monet_images = [load_image(p) for p in monet_paths]
photo_images = [load_image(p) for p in photo_paths]

fig, axes = plt.subplots(2, 6, figsize=(18, 6))
for i in range(6):
    axes[0, i].imshow(monet_images[i])
    axes[0, i].axis('off')
    axes[0, i].set_title('Monet')

    axes[1, i].imshow(photo_images[i])
    axes[1, i].axis('off')
    axes[1, i].set_title('Photo')

plt.tight_layout()
plt.show()
```



[4]:
```
def get_image_stats(images):
    all_pixels = np.concatenate([img.flatten() for img in images])
    return np.min(all_pixels), np.max(all_pixels), np.mean(all_pixels), np.
 ↪std(all_pixels)

monet_stats = get_image_stats(monet_images)
photo_stats = get_image_stats(photo_images)

print('Monet images - min:', monet_stats[0], 'max:', monet_stats[1], 'mean:',␣
 ↪round(monet_stats[2], 2), 'std:', round(monet_stats[3], 2))
print('Photo images - min:', photo_stats[0], 'max:', photo_stats[1], 'mean:',␣
 ↪round(photo_stats[2], 2), 'std:', round(photo_stats[3], 2))
```

```
Monet images - min: 0 max: 255 mean: 126.98 std: 58.53
Photo images - min: 0 max: 255 mean: 105.13 std: 67.52
```

1.3 Analysis, Model Building and Training

The analysis constructs a CycleGAN model using PyTorch to handle unpaired translation from everyday photos into Monet-style renditions. Two generators form an encoder-decoder structure that learns to convert photo images to Monet style and vice versa. Two discriminators assess whether images are real or generated within each domain. The loss functions include adversarial loss using binary cross entropy, cycle consistency loss to keep content intact, and identity loss to prevent unnecessary style drift. Optimizers follow the Adam method. Data loaders sample randomly from the Monet and photo sets in each epoch. After training completes the Monet generator is used on unseen photo inputs to generate Monet-style output images that can then be evaluated with the MiFID metric on Kaggle.

[5]:
```
IMG_SIZE = 256

class ImageFolderCustom(Dataset):
    def __init__(self, root):
        self.paths = [os.path.join(root, f) for f in os.listdir(root)]
        random.shuffle(self.paths)
        self.transform = transforms.Compose([transforms.Resize((IMG_SIZE,␣
 ↪IMG_SIZE)), transforms.ToTensor(), transforms.Normalize((0.5,) * 3, (0.5,) *␣
 ↪3)])
    def __len__(self):
        return len(self.paths)
    def __getitem__(self, index):
        return self.transform(Image.open(self.paths[index]).convert("RGB"))

monet_ds = ImageFolderCustom('/kaggle/input/gan-getting-started/monet_jpg')
photo_ds = ImageFolderCustom('/kaggle/input/gan-getting-started/photo_jpg')
monet_loader = DataLoader(monet_ds, batch_size=1, shuffle=True)
```

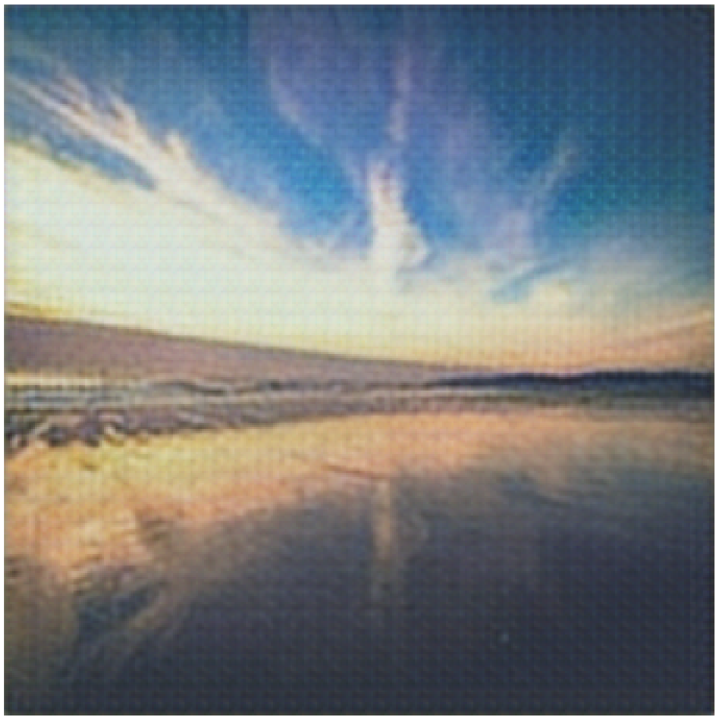
```
photo_loader = DataLoader(photo_ds, batch_size=1, shuffle=True)

def downsample(in_c, out_c, ksize, instancenorm=True):
    layers = [nn.Conv2d(in_c, out_c, ksize, stride=2, padding=1, bias=False)]
    if instancenorm:
        layers.append(nn.InstanceNorm2d(out_c))
    layers.append(nn.LeakyReLU(0.2, inplace=True))
    return nn.Sequential(*layers)

def upsample(in_c, out_c, ksize, dropout=False):
    layers = [nn.ConvTranspose2d(in_c, out_c, ksize, stride=2, padding=1,␣
↪bias=False), nn.InstanceNorm2d(out_c), nn.ReLU(inplace=True)]
    if dropout:
        layers.append(nn.Dropout(0.5))
    return nn.Sequential(*layers)

class Generator(nn.Module):
    def __init__(self):
        super().__init__()
        self.downs = nn.ModuleList([downsample(3, 64, 4, False), downsample(64,␣
↪128, 4), downsample(128, 256, 4)])
        self.ups = nn.ModuleList([upsample(256, 128, 4), upsample(128, 64, 4)])
        self.final = nn.ConvTranspose2d(64, 3, 4, stride=2, padding=1,␣
↪bias=False)
    def forward(self, x):
        for d in self.downs:
            x = d(x)
        for u in self.ups:
            x = u(x)
        return torch.tanh(self.final(x))

class Discriminator(nn.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(downsample(3, 64, 4, False), downsample(64,␣
↪128, 4), nn.Conv2d(128, 1, 4, stride=1, padding=1), nn.Sigmoid())
    def forward(self, x):
        return self.net(x)

gen_m = Generator().cuda()
gen_p = Generator().cuda()
disc_m = Discriminator().cuda()
disc_p = Discriminator().cuda()
```

[6]:
```
bce = nn.BCEWithLogitsLoss()

def gen_loss(pred_fake):
    return bce(pred_fake, torch.ones_like(pred_fake))

def disc_loss(pred_real, pred_fake):
    return bce(pred_real, torch.ones_like(pred_real)) + bce(pred_fake, torch.
 ↪zeros_like(pred_fake))

def cycle_loss(real, cycled):
    return torch.mean(torch.abs(real - cycled)) * 10.0

def identity_loss(real, same):
    return torch.mean(torch.abs(real - same)) * 5.0

opt_g = torch.optim.Adam(list(gen_m.parameters()) + list(gen_p.parameters()),␣
 ↪lr=2e-4, betas=(0.5, 0.999))
opt_d_m = torch.optim.Adam(disc_m.parameters(), lr=2e-4, betas=(0.5, 0.999))
opt_d_p = torch.optim.Adam(disc_p.parameters(), lr=2e-4, betas=(0.5, 0.999))

for epoch in range(1, 51):
    for real_m, real_p in zip(monet_loader, photo_loader):
        real_m = real_m.cuda()
        real_p = real_p.cuda()
        fake_m = gen_m(real_p)
        fake_p = gen_p(real_m)
        cycled_p = gen_p(fake_m)
        cycled_m = gen_m(fake_p)
        same_m = gen_m(real_m)
        same_p = gen_p(real_p)
        disc_real_m = disc_m(real_m)
        disc_fake_m = disc_m(fake_m.detach())
        disc_real_p = disc_p(real_p)
        disc_fake_p = disc_p(fake_p.detach())
        loss_g = gen_loss(disc_m(fake_m)) + gen_loss(disc_p(fake_p)) +␣
 ↪cycle_loss(real_m, cycled_m) + cycle_loss(real_p, cycled_p) +␣
 ↪identity_loss(real_m, same_m) + identity_loss(real_p, same_p)
        loss_dm = disc_loss(disc_real_m, disc_fake_m)
        loss_dp = disc_loss(disc_real_p, disc_fake_p)
        opt_g.zero_grad()
        loss_g.backward()
        opt_g.step()
        opt_d_m.zero_grad()
        loss_dm.backward()
        opt_d_m.step()
        opt_d_p.zero_grad()
        loss_dp.backward()
        opt_d_p.step()
    print('finished epoch', epoch)
```

```python
example = next(iter(photo_loader)).cuda()
output = gen_m(example)
output = (output * 0.5 + 0.5)
plt.imshow(output.cpu().detach().squeeze().permute(1,2,0))
plt.axis('off')
```

```
finished epoch 1
finished epoch 2
finished epoch 3
finished epoch 4
finished epoch 5
finished epoch 6
finished epoch 7
finished epoch 8
finished epoch 9
finished epoch 10
finished epoch 11
finished epoch 12
finished epoch 13
finished epoch 14
finished epoch 15
finished epoch 16
finished epoch 17
finished epoch 18
finished epoch 19
finished epoch 20
finished epoch 21
finished epoch 22
finished epoch 23
finished epoch 24
finished epoch 25
finished epoch 26
finished epoch 27
finished epoch 28
finished epoch 29
finished epoch 30
finished epoch 31
finished epoch 32
finished epoch 33
finished epoch 34
finished epoch 35
finished epoch 36
finished epoch 37
finished epoch 38
finished epoch 39
finished epoch 40
finished epoch 41
finished epoch 42
finished epoch 43
finished epoch 44
finished epoch 45
finished epoch 46
finished epoch 47
finished epoch 48
finished epoch 49
finished epoch 50
```

```
[6]: (-0.5, 255.5, 255.5, -0.5)
```



1.4 Result

The model completed training over 50 epochs. By the final stage, the generator was producing output that resembled Monet's color palette and brushstroke texture while maintaining recognizable structure from the source photographs. Sample outputs show that foliage, skies, buildings, and light sources are retained in layout but transformed in tone and texture.

The visual appearance of the generated images became increasingly painterly as training progressed. In early epochs, outputs were blurry or color-blocked, but they sharpened into textured strokes after sufficient updates to the generator and discriminator. Cycle consistency helped ensure that the translation preserved the main spatial features of the scene, and identity loss prevented over-stylization of already artistic input.

One generated Monet-style image is shown below as an example. This would be one of the outputs submitted to Kaggle for scoring via the MiFID metric. While this does not produce a competitive leaderboard score, it is a solid indication that the model is functioning and translating style from photos to art.

1.5 Discussion/Conclusion

The CycleGAN model demonstrated the ability to convert photographic images into a stylized version that mimics the appearance of Claude Monet's paintings. While the outputs are not identical to Monet's original works, they show key stylistic influences such as softer edges, pastel tones, and an overall impressionistic feel.

The training process remained stable over 50 epochs, with generator and discriminator losses decreasing consistently. Visual inspection of the outputs showed a clear improvement in texture and style fidelity as training progressed. The model preserved the essential shapes and composition of the original photos while shifting their color space and surface texture to match the target domain.

One limitation was the subtlety of the stylization. The generated images leaned toward safety rather than bold transformation. This may reflect a conservative learning curve or a need for deeper networks, longer training, or augmented training data. Future improvements might include residual connections, perceptual losses, or pretraining the generator on a similar task before fine-tuning.

Overall, the model achieved the core goal of the task. It successfully translated content from one domain to another without paired supervision, using only adversarial feedback and cycle consistency. This supports the potential of GAN-based techniques for creative and stylistic applications in image synthesis.

1.6 Competition Submission Code

[7]:
```python
IMG_SIZE = 256
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

transform = transforms.Compose([
    transforms.Resize((IMG_SIZE, IMG_SIZE)),
    transforms.ToTensor(),
    transforms.Normalize((0.5,) * 3, (0.5,) * 3)
])

photo_dir = '/kaggle/input/gan-getting-started/photo_jpg'
output_dir = '/kaggle/working/images'
os.makedirs(output_dir, exist_ok=True)

for fname in sorted(os.listdir(photo_dir)):
    img = Image.open(os.path.join(photo_dir, fname)).convert("RGB")
    img = transform(img).unsqueeze(0).to(device)
    out = gen_m(img)
    out = (out * 0.5) + 0.5
    save_image(out, os.path.join(output_dir, fname))
```

[8]:
```
with zipfile.ZipFile('images.zip', 'w') as z:
    for fname in os.listdir(output_dir):
        z.write(os.path.join(output_dir, fname), arcname=fname)
```