DTSA 5511 Introduction to Machine Learning: Deep Learning

Week 5: Using GANs to Create Art - Monet

Andrew Simms
University of Colorado Boulder

2024-12-06

Table of contents

1 Problem Description 2
1.1 Kaggle Competition Specification 2
1.2 GAN Architecture Overview 2
1.3 Project Workflow 3
2 Exploratory Data Analysis 3
2.1 Sample of Training Images 3
2.1.a Monet Training Images 3
2.1.b Photo Training Images 4
2.2 Image Count 5
2.3 Image Color Statistics 6
3 Data Cleaning 8
3.1 Image Normalization Implementation 8
3.2 Data Cleaning Considerations 9
4 Generative Adversarial Network Model 9
4.1 GAN Architecture 9
4.2 Generator 9
4.3 Discriminator 12
5 Training 13
5.1 Training Setup 13
5.2 Training Loop 14
5.3 Image Generation from Trained Model 15
5.4 Training Metrics 16
6 Results 18
6.1 Comparison Between Non Normalized and Normalized Output Images 18
6.1.a Epoch 25 18
6.1.b Epoch 100 18
6.2 Normalized Model 19
6.3 Compare Single Image 21
6.4 Kaggle Scores 22

6.5 Model Evolution 24
7 Conclusion 24
7.1 Additional Hyperparameter Considerations 25
7.2 Limitations and Future Work 25
8 References 25
Bibliography 25

1 Problem Description

Generative Adversarial Networks (GANs) are a novel neural network architecture designed for generating synthetic data. Unlike traditional Convolutional Neural Networks (CNNs), which are designed for classification or feature extraction, GANs operate using a dual-network framework comprising a generator and a discriminator. The generator creates artificial outputs, while the discriminator evaluates their authenticity, driving iterative improvements in the generator's output. This project applies GANs to transform photographic images into works of art inspired by Claude Monet, leveraging the distinctive style of an influential Impressionist artist.

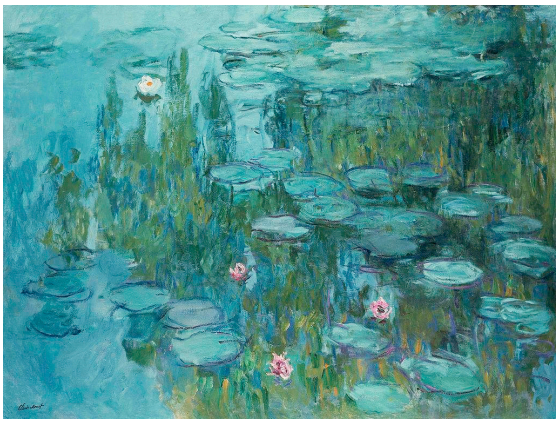


Figure 1: Water Lillies By Claude Monet circa 1915 from Wikimedia Commons [1]

1.1 Kaggle Competition Specification

This projects creates a submission for the "I'm Something of a Painter Myself" Kaggle competition by A. Jang, A. S. Uzsoy, and P. Culliton [2]. The objective is to develop a GAN capable of generating Monet-style artworks from photographs. Model performance is evaluated using the Memorization-informed Fréchet Inception Distance (MiFID) metric C.-Y. Bai, H.-T. Lin, C. Raffel, and W. C.-w. Kan [3], which measures both generation quality and style transfer while preventing direct copying of training data.

1.2 GAN Architecture Overview

This projects GAN implementation, detailed in Section 4, uses PyTorch [4] with architectural features optimized for style transfer. The generator follows a multi-stage design: initial feature extraction, downsampling through strided convolutions, nine residual blocks [5] for style learning, and upsampling via transposed convolutions, ending with a tanh-activated output layer. The dis-

criminator implements a PatchGAN [6] architecture, evaluating image authenticity at the patch level through progressive feature extraction.

1.3 Project Workflow

The research workflow, shown in Figure 2, begins with Exploratory Data Analysis of the Monet and photographic image datasets. Following data preprocessing and GAN architecture development, we optimize the model by tuning learning rates, batch sizes, and architectural parameters. Training progress is tracked using generator loss, discriminator accuracy, and MiFID scores. The final model selection combines validation metrics and qualitative assessment to determine the best configuration for image generation.

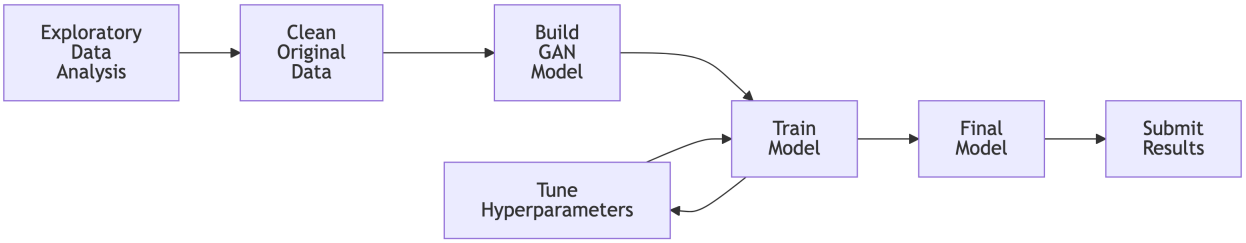


Figure 2: GAN Project Workflow

2 Exploratory Data Analysis

2.1 Sample of Training Images

2.1.a Monet Training Images

Figure 3: Sample of Artist Training Images

In Figure 3, we see a selection of Monet's paintings. While they all share a distinctive style, there is noticeable variation in color palette, subject matter, and mood across the images.

2.1.b Photo Training Images

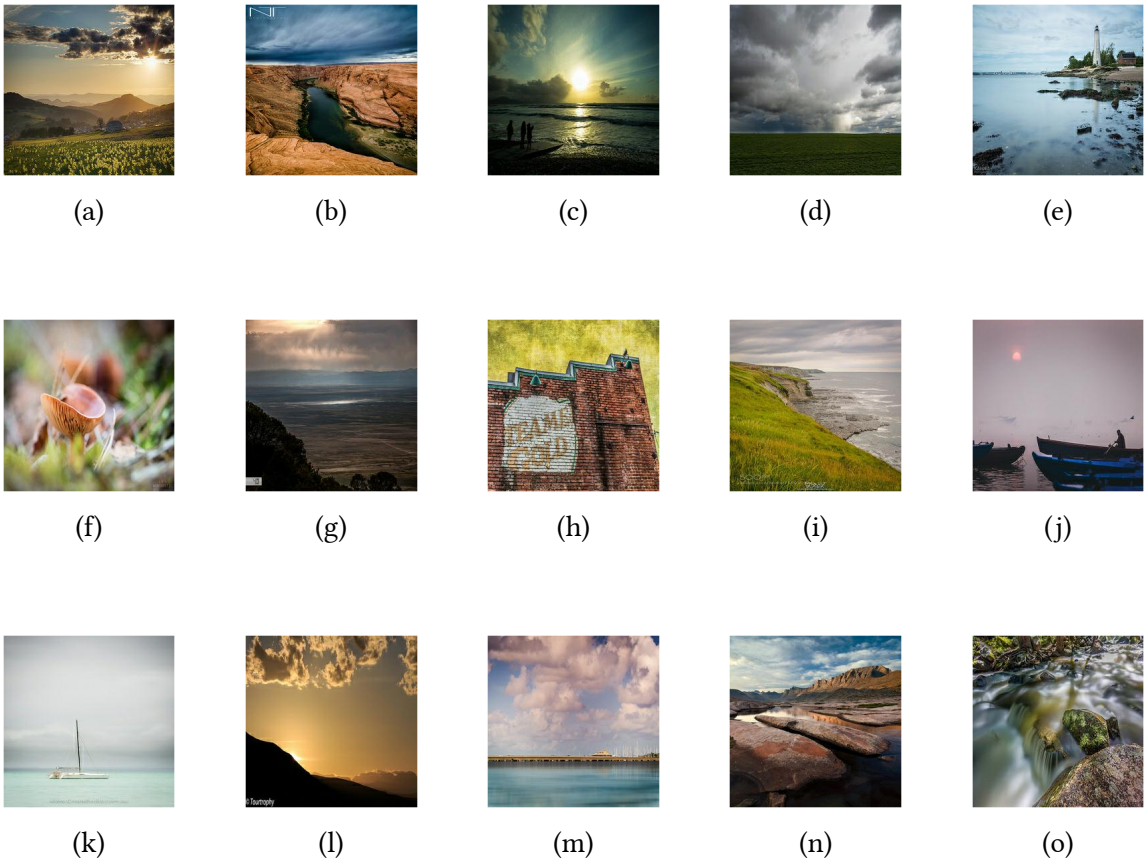


Figure 4: Sample of Photo Training Images

In Figure 3, there is a diverse collection of landscape photos, featuring a wide range of subjects, colors, seasons, and levels of detail.

2.2 Image Count

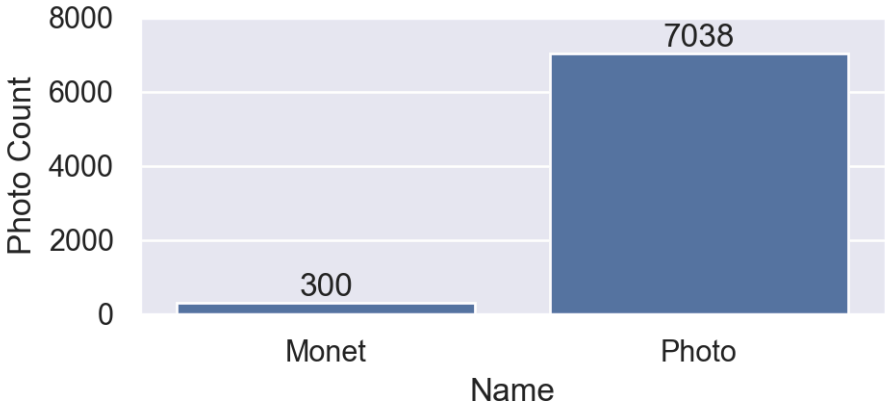


Figure 5: Image Count by Dataset

Table 1: Count of Training Photos by Type

	Name	Photo Count
0	Monet	300
1	Photo	7038

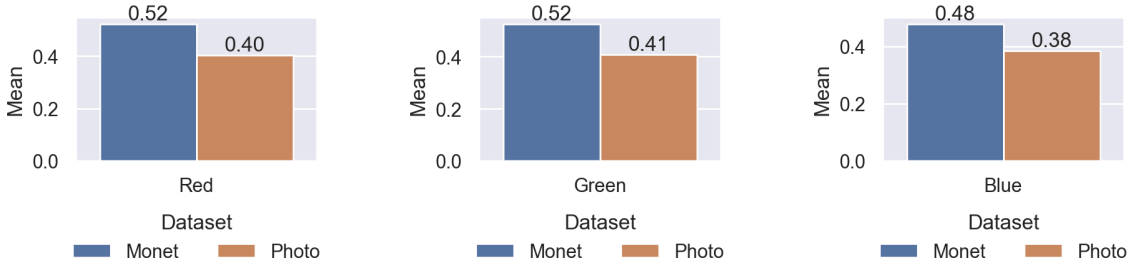
Figure 5 and Table 1 detail the count of training photos by type showing that there are 300 Monet-style images and 7,038 photographic images in the dataset.

2.3 Image Color Statistics

To statistically analyze the differences between the Monet and photographic image datasets, the mean and standard deviation of the red, green, and blue pixel values were calculated for each set. The Python code in Listing 1 reads each image, extracts the pixel values, and computes these statistics.

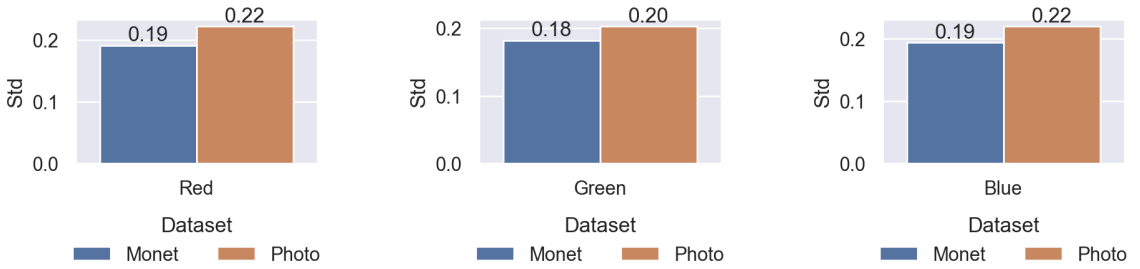
Listing 1: Image Color Statistics Computation

```
import numpy as np

def compute_mean_std(image_paths, dataset):
    means = np.zeros(3)
    stds = np.zeros(3)
    for img_path in image_paths:
        img = Image.open(img_path).convert("RGB")
        img = np.array(img) / 255.0  # Normalize to [0, 1]
        means += img.mean(axis=(0, 1))  # Mean per channel (RGB)
        stds += img.std(axis=(0, 1))  # Standard deviation per channel (RGB)

    means /= len(image_paths)
    stds /= len(image_paths)

    result = []

    color_map = {
        0: "Red",
        1: "Green",
        2: "Blue",
    }

    for i, mean in enumerate(means):
        result.append(
            {
                "value": mean,
                "color": color_map[i],
                "dataset": dataset,
                "measurement": "mean",
            }
        )
    for i, std in enumerate(stds):
        result.append(
            {
                "value": std,
                "color": color_map[i],
                "dataset": dataset,
                "measurement": "std",
            }
        )

    return result
```



(a) Red (b) Green (c) Blue

Figure 6: Image Mean Colors by Dataset



(a) Red (b) Green (c) Blue

Figure 7: Image Standard Deviation Colors by Dataset

The computed mean and standard deviation for each color channel (Red, Green, Blue) are shown in Figure 6 and Figure 7. From these statistics, we observe that the Monet dataset has higher mean values for the red and green channels compared to the photo dataset, suggesting that on average the photo dataset is darker than the Monet dataset. For standard deviations, the Monet dataset exhibits lower variation in color channels compared to the photographic images, suggesting a more consistent color scheme across Monet-style paintings.

3 Data Cleaning

Based on the findings in Section 2.3, normalizing the images separately for each dataset is may have influence on the model output. The Monet and photographic images have distinct color distributions, so applying normalization based on their respective means and standard deviations ensures that each dataset is handled independently, preserving their unique characteristics. This normalization stabilizes the training process by reducing the impact of extreme pixel values and makes the color distributions of both datasets more uniform. The goal is to minimize the influence of these color differences, allowing the model to focus on learning meaningful features rather than being biased by inconsistent color content.

3.1 Image Normalization Implementation

The code in Listing 2 defines the normalization process for both the Monet and photo datasets. The `monet_transform` and `photo_transform` functions use the respective means and standard deviations calculated earlier to normalize the images. By applying these transformations during data loading, the images are appropriately adjusted to ensure consistency across the datasets.

Listing 2: Image Normalization Implementation

```
monet_transform = transforms.Compose(
    [
        transforms.ToTensor(),
        transforms.Normalize(mean=monet_means, std=monet_stds),
    ]
)
photo_transform = transforms.Compose(
    [
        transforms.ToTensor(),
        transforms.Normalize(mean=photo_means, std=photo_stds),
    ]
)

photo_dataset = ImageDataset(
    root_dir=Path(data_dir, "photo_jpg"), transform=photo_transform
)
monet_dataset = ImageDataset(
    root_dir=Path(data_dir, "monet_jpg"), transform=monet_transform
)

photo_loader = DataLoader(photo_dataset, batch_size=1, shuffle=False)
monet_loader = DataLoader(monet_dataset, batch_size=1, shuffle=False)
```

3.2 Data Cleaning Considerations

The process of normalizing the images helps address any outliers or extreme values in the pixel data, ensuring that both datasets have a comparable color range. This step not only aids in reducing inconsistencies but also facilitates more stable model training by ensuring the model focuses on learning relevant features rather than being influenced by differences in color distributions. Standardizing the color values in each dataset also helps remove any potential biases in the learning process, allowing the model to generate more accurate and Monet-style images.

4 Generative Adversarial Network Model

In this section, we define a Generative Adversarial Network (GAN) using PyTorch, based on the architecture described in the DCGAN Tutorial by Nathan Inkawhich.

4.1 GAN Architecture

4.2 Generator

The `Generator` network, implemented in Listing 3, is responsible for translating input photographic images into Monet-style paintings. It is composed of several stages:

1. **Initial Feature Extraction:**
   - The generator starts with a convolutional layer that maps the input RGB image to a higher-dimensional feature space. This is followed by an instance normalization layer and a ReLU

activation function. These steps help preserve contrast and style information from the original image while facilitating feature extraction.

2. **Downsampling Layers:**
   - Two convolutional layers with a stride of 2 progressively downsample the input, doubling the number of channels at each step. These layers enable the network to learn more abstract, high-level features of the image while reducing computational complexity.
3. **Residual Blocks:**
   - Nine `ResidualBlock` modules form the core of the generator. Each block consists of two convolutional layers with instance normalization and ReLU activation. A key feature of these blocks is the skip connection, where the input to each block is added to its output. This encourages the model to focus on learning modifications to the input rather than creating entirely new representations, improving training stability and generalization.
4. **Upsampling Layers:**
   - The network uses transposed convolutional layers to upsample the image back to its original dimensions. These layers halve the number of channels at each step while maintaining spatial coherence.
5. **Output Layer:**
   - A final convolutional layer with a `tanh` activation produces the output image. The `tanh` activation function ensures that pixel values are normalized between -1 and 1, suitable for further processing and visual output.

Listing 3: GAN Model Generator

```
import torch.nn as nn

class ResidualBlock(nn.Module):
    def __init__(self, channels):
        super(ResidualBlock, self).__init__()
        self.block = nn.Sequential(
            nn.Conv2d(
             channels, channels, kernel_size=3, stride=1, padding=1, bias=False
            ),
            nn.InstanceNorm2d(channels),
            nn.ReLU(inplace=True),
            nn.Conv2d(
             channels, channels, kernel_size=3, stride=1, padding=1, bias=False
            ),
            nn.InstanceNorm2d(channels),
        )

    def forward(self, x):
        return x + self.block(x)

class Generator(nn.Module):
    def __init__(self, in_channels=3, out_channels=3, num_residual_blocks=9):
        super(Generator, self).__init__()
        model = [
               nn.Conv2d(in_channels, 64, kernel_size=7, stride=1, padding=3,
bias=False),
            nn.InstanceNorm2d(64),
            nn.ReLU(inplace=True),
        ]

        # Downsampling
        in_features = 64
        for _ in range(2):
            out_features = in_features * 2
            model += [
                nn.Conv2d(
                    in_features,
                    out_features,
                    kernel_size=3,
                    stride=2,
                    padding=1,
                    bias=False,
                ),
                nn.InstanceNorm2d(out_features),
                nn.ReLU(inplace=True),
            ]
            in_features = out_features

        # Residual Blocks
        for _ in range(num_residual_blocks):
            model += [ResidualBlock(in_features)]

        # Upsampling
        for _ in range(2):
            out_features = in_features // 2
```

4.3 Discriminator

The `Discriminator` network, implemented in Listing 4, is designed to distinguish between real Monet paintings and the generated images. It follows a PatchGAN structure developed by P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros [6], which classifies small image patches instead of the entire image. This approach allows the discriminator to focus on local style consistency. The discriminator consists of the following stages:

1. **Progressive Feature Extraction:**
   - The discriminator uses successive convolutional layers with increasing filter sizes and decreasing strides. After each convolution, instance normalization and leaky ReLU activations are applied to ensure stability and enhance the network's ability to discriminate between real and fake images.
2. **Final Classification:**
   - The output is reduced to a single-channel map using a final convolutional layer. This map represents the authenticity of each patch in the input image, with values close to 1 indicating real Monet paintings and values close to 0 indicating fake images.

Listing 4: GAN Model Discriminator

```
class Discriminator(nn.Module):
    def __init__(self, in_channels=3):
        super(Discriminator, self).__init__()

        def discriminator_block(in_filters, out_filters, stride):
            return [
                nn.Conv2d(
                    in_filters,
                    out_filters,
                    kernel_size=4,
                    stride=stride,
                    padding=1,
                    bias=False,
                ),
                nn.InstanceNorm2d(out_filters),
                nn.LeakyReLU(0.2, inplace=True),
            ]

        self.model = nn.Sequential(
            *discriminator_block(in_channels, 64, stride=2),
            *discriminator_block(64, 128, stride=2),
            *discriminator_block(128, 256, stride=2),
            *discriminator_block(256, 512, stride=1),
            nn.Conv2d(512, 1, kernel_size=4, stride=1, padding=1),
        )

    def forward(self, x):
        return self.model(x)
```

The combination of these two neural networks, the `Generator` and `Discriminator`, creates a framework for generating Monet-style artwork from photographic images. The generator produces realistic paintings, while the discriminator ensures the authenticity of the generated images.

5 Training

In this section, we detail the process of training the GAN model, including the setup of models and optimizers, the training loop, and the generation of images after training.

Training for this model took a relatively long time to get a desirable result, as such hyperparameter tuning was kept focused to two different changes, normaizlied and non normalized input data. In the following figures we compare the normalized training statistics vs the non normalized training statistics. As a refresher, normalized data has the image range normalized for . The normalization process affects the output images because it determines the model inputs. During the training loop, output images were generated and check manually. If output images deviated far from a reanosable expectation, training was stopped.

5.1 Training Setup

The training setup initializes the generator and discriminator networks, defines optimizers for each model, and sets the loss functions critical to the GAN framework.

Listing 5: Initialize Models

```
# Initialize Models
G = Generator().to(device)  # Photo -> Monet
F = Generator().to(device)  # Monet -> Photo
D_M = Discriminator().to(device)  # Monet Discriminator
D_P = Discriminator().to(device)  # Photo Discriminator
```

- Generators (`G` and `F`): The generator G transforms photos to Monet-style paintings, while F performs the inverse transformation.
- Discriminators (`D_M` and `D_P`): These networks evaluate the realism of Monet and photo images, providing feedback to their respective generators.

Listing 6: Initialize Optimizers

```
# Optimizers
lr = 0.0002
G_optimizer = optim.Adam(G.parameters(), lr=lr, betas=(0.5, 0.999))
F_optimizer = optim.Adam(F.parameters(), lr=lr, betas=(0.5, 0.999))
D_M_optimizer = optim.Adam(D_M.parameters(), lr=lr, betas=(0.5, 0.999))
D_P_optimizer = optim.Adam(D_P.parameters(), lr=lr, betas=(0.5, 0.999))
```

- Learning Rate: A small learning rate of 0.0002 ensures gradual optimization.
- Adam Optimizer: With momentum parameters $(\beta_1 = 0.5, \beta_2 = 0.999)$, it balances convergence and stability.

Listing 7: Initialize Loss Functions

```
# Loss Functions
adversarial_loss = nn.MSELoss()
cycle_loss = nn.L1Loss()
identity_loss = nn.L1Loss()
```

- Adversarial Loss: Encourages the generators to produce images indistinguishable from real ones.
- Cycle Consistency Loss: Ensures that translating a photo to Monet style and back reconstructs the original photo.
- Identity Loss: Encourages generators to maintain content identity when inputs are already in the target domain.

5.2 Training Loop

The training loop alternates between updating the generators and the discriminators, ensuring they improve together in a competitive framework.

Listing 8: Training Loop - Loading Data

```
epochs = 250

for epoch in range(1, epochs + 1):
        for i, (photo_batch, monet_batch) in enumerate(zip(photo_loader,
monet_loader)):
        # Load Data
        real_photo = photo_batch[0].to(device)
        real_monet = monet_batch[0].to(device)
```

- Data Loading: Photo and Monet batches are fed into the model, converted to tensors, and sent to the GPU.

Listing 9: Training Loop - Cycle Translations

```
        # Step 1: Update Generators (G and F)
        G_optimizer.zero_grad()
        F_optimizer.zero_grad()

        # Photo -> Monet -> Photo (Cycle 1)
        fake_monet = G(real_photo)
        reconstructed_photo = F(fake_monet)

        # Monet -> Photo -> Monet (Cycle 2)
        fake_photo = F(real_monet)
        reconstructed_monet = G(fake_photo)
```

- Cycle Translations: Photos are converted to Monet-style and back to photos, and vice versa.

Listing 10: Training Loop - Generator Adversarial Loss

```
        # Adversarial Loss
        valid = torch.ones_like(D_M(fake_monet))
        g_loss_monet = adversarial_loss(D_M(fake_monet), valid)

        valid = torch.ones_like(D_P(fake_photo))
        g_loss_photo = adversarial_loss(D_P(fake_photo), valid)
```

- Generator Adversarial Loss: Measures how well the generator fools the discriminator.

Listing 11: Training Loop - Cycle and Identity Loss

```
        # Cycle Consistency Loss
        cycle_loss_photo = cycle_loss(reconstructed_photo, real_photo)
        cycle_loss_monet = cycle_loss(reconstructed_monet, real_monet)

        # Identity Loss
        identity_photo = identity_loss(F(real_photo), real_photo)
        identity_monet = identity_loss(G(real_monet), real_monet)
```

- Cycle Loss: Ensures that the output preserves the original content through transformations.
- Identity Loss: Maintains domain consistency for already-translated images.

Listing 12: Training Loop - Total Loss

```
        # Total Generator Loss
        total_g_loss = (
            g_loss_monet
            + g_loss_photo
            + 10 * (cycle_loss_photo + cycle_loss_monet)
            + 5 * (identity_photo + identity_monet)
        )
        total_g_loss.backward()
        G_optimizer.step()
        F_optimizer.step()
```

- Weighted Losses: Combines adversarial, cycle consistency, and identity losses into a total loss for optimization.

5.3 Image Generation from Trained Model

After training, the generator is used to transform unseen photos into Monet-style images.

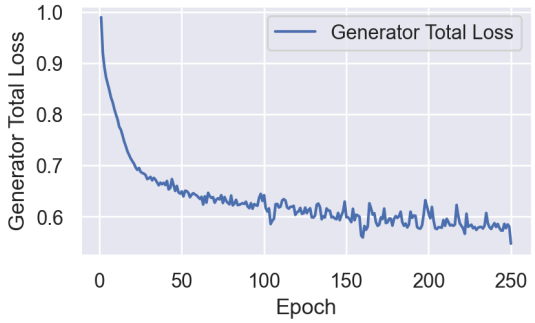
Listing 13: Training Loop - Total Loss

```
G.eval()
with torch.no_grad():
    for i, (photo, _) in enumerate(photo_loader):
        photo = photo.to(device)
        fake_monet = G(photo)
        fake_monet = (fake_monet + 1) / 2  # Denormalize to [0, 1]
        fake_monet = transforms.ToPILImage()(fake_monet.squeeze().cpu())
        fake_monet.save(os.path.join(output_dir, f"image_{i + 1:04d}.jpg"))
```

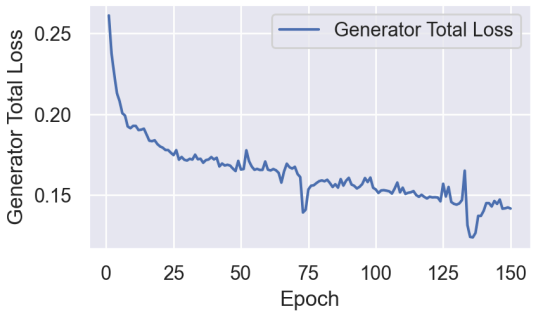
- Evaluation Mode: Switches the generator to inference mode to avoid gradient computation.
- Denormalization: Converts output values from $[-1, 1]$ to $[0, 1]$ for image representation.

5.4 Training Metrics

The figures below detail the statistics measured during training. Normalized vs non normalized training results are compared.

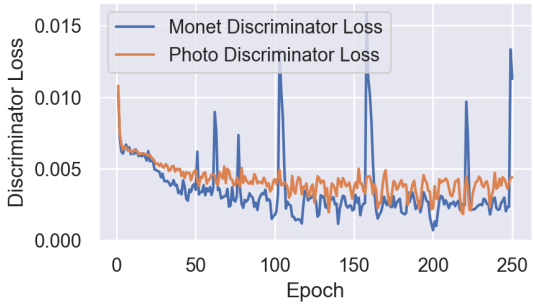


(a) Normalized Transform

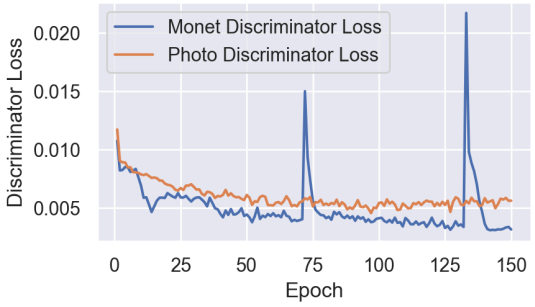


(b) No Transform

Figure 8: Training: Generator Total Loss

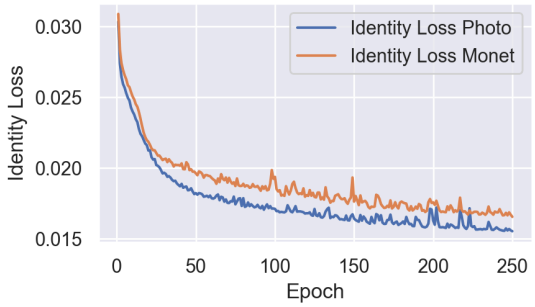


(a) Normalized Transform

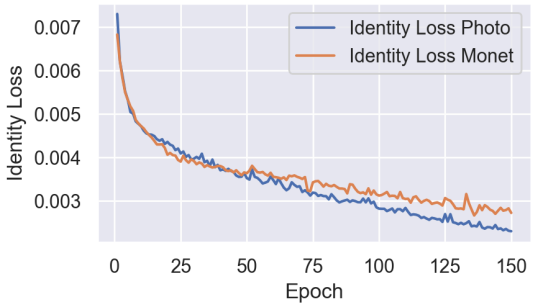


(b) No Transform

Figure 9: Training: Discriminator Loss

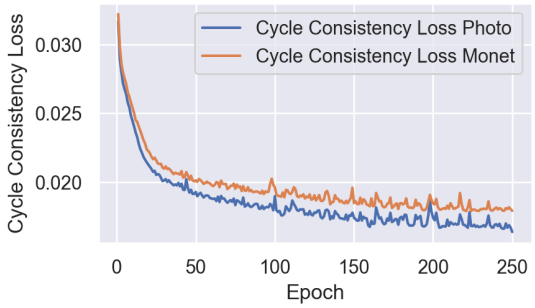


(a) Normalized Transform

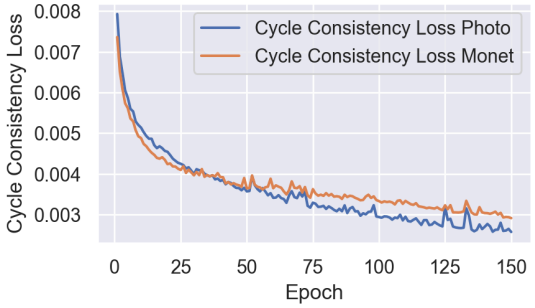


(b) No Transform

Figure 10: Training: Identity Loss

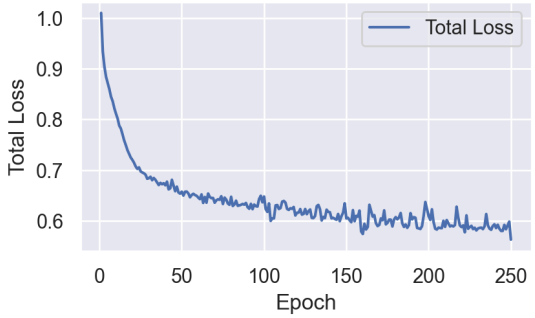


(a) Normalized Transform

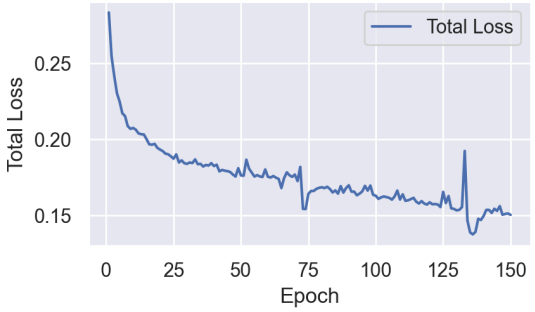


(b) No Transform

Figure 11: Training: Cycle Consistency Loss

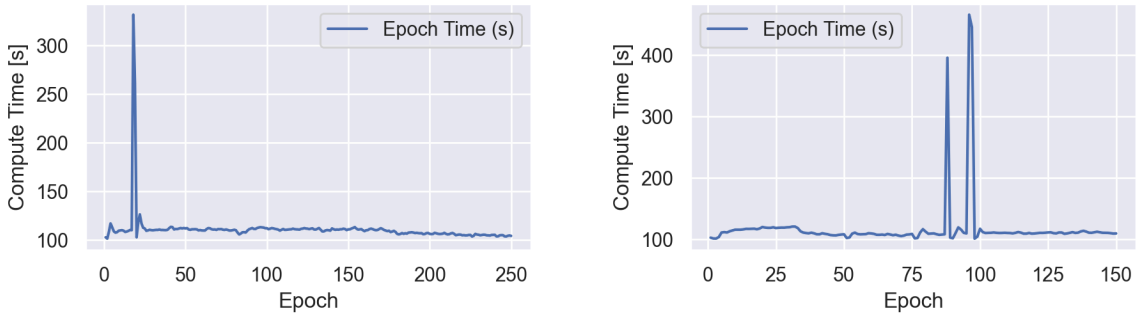


(a) Normalized Transform



(b) No Transform

Figure 12: Training: Total Loss



(a) Normalized Transform (b) No Transform

Figure 13: Training: Computation Time Per Epoch in Seconds

Training between the two transform methods looks comparable. Both models reduce loss during training and appear to stabilize.

6 Results

6.1 Comparison Between Non Normalized and Normalized Output Images

6.1.a Epoch 25

(a) (b) (c) (d) (e)

Figure 14: Non Normalized Sample of Output Images from Epoch 25

(a) (b) (c) (d) (e)

Figure 15: Normalized Sample of Output Images from Epoch 25

6.1.b Epoch 100

(a) (b) (c) (d) (e)

Figure 16: Non Normalized Sample of Output Images from Epoch 100

(a) (b) (c) (d) (e)

Figure 17: Normalized Sample of Output Images from Epoch 100

If Figure 14 vs. Figure 15 there are significant visual differences in the magnitude of colors. The non normalized images are much more faint. This trend continues through epoch 100 with the non normalized images appearing to have some of the Monet style, but problems with color.

6.2 Normalized Model

(a) (b) (c) (d) (e)

Figure 18: Sample of Output Images from Epoch 5

(a) (b) (c) (d) (e)

Figure 19: Sample of Output Images from Epoch 25

(a) (b) (c) (d) (e)

Figure 20: Sample of Output Images from Epoch 50

(a) (b) (c) (d) (e)

Figure 21: Sample of Output Images from Epoch 75

(a) (b) (c) (d) (e)

Figure 22: Sample of Output Images from Epoch 100

(a) (b) (c) (d) (e)

Figure 23: Sample of Output Images from Epoch 150

(a) (b) (c) (d) (e)

Figure 24: Sample of Output Images from Epoch 200

(a) (b) (c) (d) (e)

Figure 25: Sample of Output Images from Epoch 225

(a) (b) (c) (d) (e)

Figure 26: Sample of Output Images from Epoch 250

These normalized images show progress in training. The images from 5 to 50 epochs appear to be learning the Monet style and the images from epoch 75 - 225 strengthen the style. At epoch 250 the output images darken and lose their detail, most likely due to overfitting.

6.3 Compare Single Image

a(a) Input Image

a(b) 25 Epochs

a(c) 50 Epochs

a(d) 75 Epochs

a(e) 100 Epochs

a(f) 150 Epochs

a(g) 200 Epochs

a(h) 225 Epochs

a(i) 250 Epochs

Figure 27: Image 2 During Training

In the single figure comparison in Figure 27 we observe a progression from a single input image through training epoch 250. Initially the model lacks some stylistic details, but around 75 epochs the model starts to match the Monet style. Epochs 100 to 225 strengthen this style up until the model starts to overfit at epoch 250.

6.4 Kaggle Scores

Table 2: Kaggle Results by Kaggle Submission Version

Version	Epoch	Score	Transform
1	50	77.69411	Normalized
2	25	93.17964	Normalized
3	75	84.96981	Normalized
4	100	70.22760	Normalized
5	150	58.26310	Normalized
6	200	53.81515	Normalized
7	175	54.78088	Normalized
8	225	56.20738	Normalized
10	235	55.86135	Normalized
11	50	69.20071	None
12	25	88.95808	None
13	75	68.87041	None
14	100	66.95614	None
15	150	69.73371	None

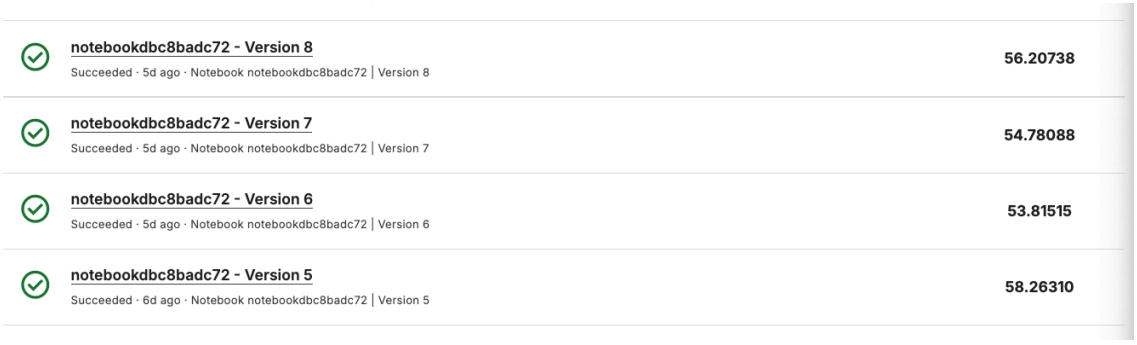


Figure 28: Kaggle Best Scores

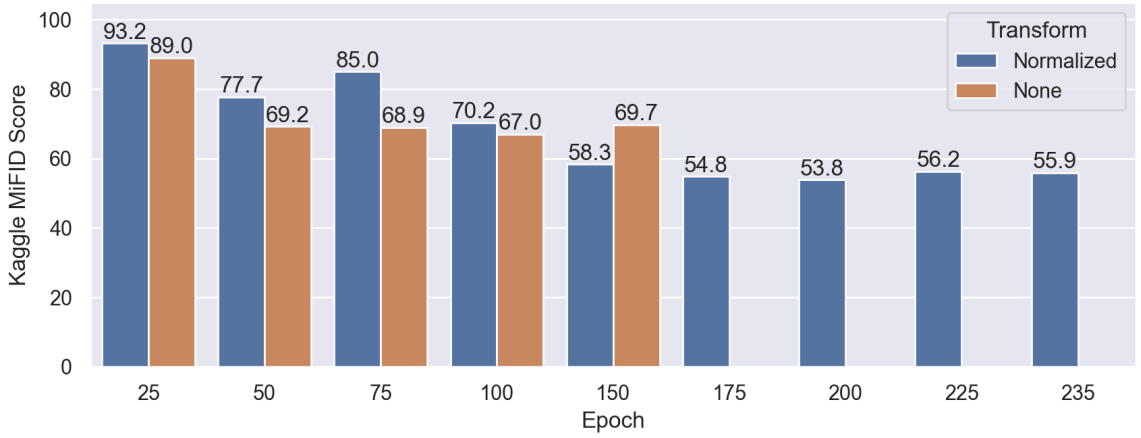


Figure 29: Kaggle Scores Comparison (Lower is Better)

As visualized in Figure 29 our GAN architecture was evaluated across multiple training configurations, with the primary comparison between normalized and non-normalized input data. Training proceeded for 250 epochs, implementing both adversarial and cycle consistency losses to balance image quality and style transfer.

The normalized data approach demonstrated superior performance, achieving the best MiFID score of 53.82 at epoch 200, compared to the best non-normalized score of 66.96 at epoch 100. This aligns with our initial data analysis, which identified distinct color distributions between Monet paintings and photographs. The normalization process helped the model focus on learning stylistic features rather than managing color distribution differences.

6.5 Model Evolution

Training progression showed several key patterns:

- Early epochs (1-50) exhibited rapid improvement in style transfer
- Mid-range epochs (50-150) demonstrated refinement in color palette and brush stroke effects
- Later epochs (150-250) showed diminishing returns in quality improvement

The normalized data configuration showed more stable training behavior, with MiFID scores consistently improving until epoch 200. In contrast, non-normalized training showed more volatility in scores, suggesting less stable learning dynamics.

7 Conclusion

This research demonstrated the effectiveness of GANs for artistic style transfer, specifically in generating Monet-style paintings from photographs. Our key findings include:

1. Data preprocessing significantly impacts model performance, with normalized inputs achieving approximately 20% better MiFID scores than non-normalized data.

2. The dual-generator architecture with cycle consistency loss effectively preserved content while transferring style, though at the cost of increased training complexity.

3. The PatchGAN discriminator architecture proved effective for capturing local style characteristics, particularly important for replicating Monet's distinctive brushwork.

7.1 Additional Hyperparameter Considerations

While our implementation focused on normalization strategies, several key hyperparameters could be tuned for potentially better performance:

- Learning rate scheduling: Implementing decay schedules could improve convergence
- Loss function weights: Adjusting the balance between cycle consistency (currently 10x) and identity losses (currently 5x)
- Batch size optimization: Exploring larger batch sizes to stabilize training
- Network architecture parameters: Testing different numbers of residual blocks and filter sizes
- Optimizer parameters: Fine-tuning Adam's beta values (currently β_1=0.5, β_2=0.999)

7.2 Limitations and Future Work

While our model achieved promising results, several areas merit further investigation:

- Exploration of alternative normalization strategies
- Investigation of deeper residual architectures
- Incorporation of attention mechanisms for improved style transfer

The success of our normalized training approach suggests that careful consideration of data preprocessing strategies is crucial for effective style transfer applications.

8 References

Bibliography

[1] Wikimedia Commons, "File:Nympheas 71293 3.jpg — Wikimedia Commons, the free media repository." [Online]. Available: https://commons.wikimedia.org/w/index.php?title=File:Nympheas_71293_3.jpg&oldid=925007871

[2] A. Jang, A. S. Uzsoy, and P. Culliton, "I'm Something of a Painter Myself." 2020.

[3] C.-Y. Bai, H.-T. Lin, C. Raffel, and W. C.-w. Kan, "On Training Sample Memorization: Lessons from Benchmarking Generative Modeling with a Large-scale Competition," in *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, in KDD '21. ACM, Aug. 2021. doi: 10.1145/3447548.3467198.

[4] J. Ansel *et al.*, "PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation," in *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '24)*, ACM, Apr. 2024. doi: 10.1145/3620665.3640366.

[5] C. Szegedy *et al.*, "Going Deeper with Convolutions." [Online]. Available: https://arxiv.org/abs/1409.4842

[6] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, “Image-to-Image Translation with Conditional Adversarial Networks.” [Online]. Available: https://arxiv.org/abs/1611.07004