Multi-Style Image Generation using StyleGANs

Aarush Singh Kushwaha - 230019
Anirudh Singh - 230142
Fariz Ali - 230402
Kshitij Bhardwaj - 230580

Abstract

This project introduces a simple and effective framework for multi-style image generation using a CycleGAN-based architecture with multiple types of attention mechanisms, namely Squeeze and Excitation, Convolutional Block Attention Modules(CBAM), Coordiante Attention and Multi Head Attention. The system is trained on balanced, preprocessed datasets of real faces (UTKFace, ages 18–120) , each resized and normalized for stable GAN training. The model uses adversarial, cycle-consistency, and identity losses to enable unpaired translation between realistic and Ghibli-style faces, while also supporting plausible age manipulation. The codebase demonstrates robust training, clear visual improvements, and identity-preserving results, providing a practical solution for creative style and attribute transfer tasks in digital arts and entertainment.

1. Introduction

Image-to-image translation stands as a cornerstone of contemporary computer vision, enabling the transformation of images across domains while preserving core semantic content. Recent advances in generative adversarial networks (GANs) and diffusion models have dramatically expanded the creative and practical horizons of this technology, making possible applications that were once the domain of science fiction. In this work, we present two independent, state-of-the-art systems that exemplify the research and engineering progress in this field: a robust age manipulation framework and a Studio Ghibli-inspired style conversion pipeline.

The first application, **age manipulation**, leverages a StyleGAN-based architecture [4]to perform realistic, identity-preserving age progression and regression on facial images. Users can upload a centered facial photograph and select either ”old to young” or ”young to old” transformation, receiving high-fidelity results in real time. This system is grounded in the latest research on disentangled latent space manipulation and age-conditioned generative modeling, reflecting the growing demand for controllable facial attribute editing in entertainment, forensics, and personalized digital experiences. Our approach is informed by recent literature that emphasizes fine-grained, continuous age transformation while rigorously maintaining subject identity, addressing key challenges highlighted in works such as SAM and AgeTransGAN.

The second application focuses on **Ghibli-style image generation**, utilizing an enhanced CycleGAN architecture with Coord(Coordinate Attention) blocks [2] to translate real facial images into the distinct, expressive style of Studio Ghibli animation. The model is trained on a ***custom, Stable Diffusion-generated dataset*** of Ghibli character faces, paired with carefully curated real face images. This system supports bidirectional translation, including the tentative capability to revert Ghibli-style images back to photorealistic face,a frontier rarely explored in prior work. The integration of Coord blocks is a deliberate architectural choice, enabling the model to recalibrate feature responses and better capture the nuanced textures and palettes characteristic of Ghibli art, as evidenced by recent advances in unpaired image translation.

Both systems are built on meticulously prepared datasets and rigorous data augmentation strategies, ensuring diversity and robustness in training. Preprocessing pipelines incorporate **Deepface** face detection and alignment, with all images standardized to **128×128 resolution** and normalized for stable GAN optimization. The training regimen, employs adversarial, cycle-consistency, and identity losses, but evaluation is purposefully centered on visual fidelity and user perception—a reflection of the ongoing discourse in generative modeling regarding the limitations of traditional quantitative metrics for subjective, creative outputs.

Our contributions are **twofold**: first, we provide deployable, user-facing applications that demonstrate the practical impact of generative modeling; second, we offer a research-centric analysis of architectural innovations and dataset curation, that can inform future work in multi-domain, multi-attribute image synthesis. These systems not only illustrate the current state of the art but also chart a path forward for

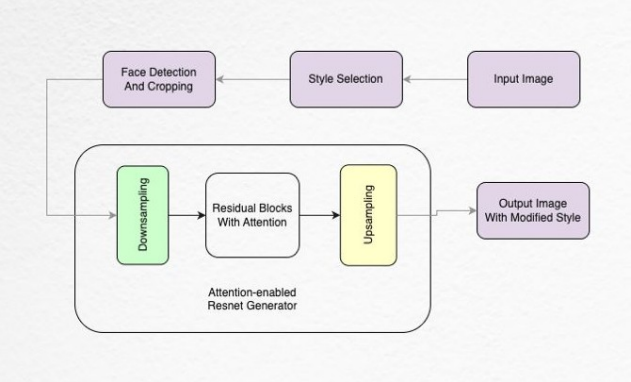

Figure 1. Proposed Pipeline

research at the intersection of AI, digital art, and human-centered design.

2. Background and Related Work

2.1. Evolution of Generative Adversarial Networks

Generative Adversarial Networks (GANs) have emerged as one of the most powerful tools in the field of image synthesis. GANs consist of two neural networks ,**a generator and a discriminator** , that compete in a zero-sum game. The generator attempts to create images that look as realistic as possible, while the discriminator tries to distinguish between real and generated images. Through this adversarial process, the generator learns to produce increasingly convincing outputs.

Early GAN architectures, such as DCGAN (Deep Convolutional GAN) [1], introduced convolutional layers and batch normalization to improve training stability and image quality. However, GANs still struggled with issues like mode collapse, training instability, and low-resolution outputs.

Significant progress was made with the introduction of StyleGAN [4], which transformed the field of high-resolution image generation. StyleGAN introduced a style-based architecture that uses adaptive instance normalization (AdaIN) to inject latent style vectors into various layers of the generator. This allows for fine-grained control over visual attributes such as pose, texture, and facial features, and enables smooth interpolation in the disentangled latent space. As a result, users can manipulate specific aspects of an image (e.g., changing hair color or expression) without affecting other unrelated features.

In parallel, other GAN variants tackled different challenges. CycleGAN [7], for instance, addressed the problem of unpaired image-to-image translation enabling transformations between two domains (e.g., horses - zebras, summer - winter) without needing corresponding image pairs. This is achieved by enforcing a cycle-consistency loss, ensuring that translating from domain A to B and back to A yields the original input. Another notable approach, Pix2Pix, focuses on paired datasets and uses a conditional GAN framework for tasks like image colorization and sketch-to-photo synthesis.

Recent advancements also include the integration of attention mechanisms and transformer-based architectures into GAN pipelines, further enhancing their ability to model global relationships and fine details. These innovations are paving the way for more robust, flexible, and interpretable generative models.

Together, these developments have made GANs central to many computer vision tasks, ranging from artistic style transfer and photo editing, to medical imaging and data augmentation.

2.2. Attention Mechanism

Attention mechanisms have become a fundamental component in modern neural network architectures. The core idea behind attention is to enable a model to focus on relevant parts of the input data while suppressing irrelevant information. In its basic form, an attention function maps a query and a set of key-value pairs to an output, where the output is computed as a weighted sum of the values, with the weight assigned to each value determined by a compatibility function between the query and the corresponding key.

The Transformer architecture, introduced in the paper ”Attention Is All You Need ”[5] , popularized attention mechanisms by showing that complex sequence processing tasks could be handled using attention alone, without relying on recurrent or convolutional operations. The effectiveness of this approach stems from attention’s ability to capture long-range dependencies regardless of their distance in the input sequence.

2.2.1 Squeeze-and-Excitation (SE) Blocks

Building on the concept of attention, Squeeze-and-Excitation (SE) blocks [3] were introduced as a channel attention mechanism specifically for convolutional neural networks. SE blocks train upon inter dependencies between channels by using global information to selectively emphasize informative features.

The SE module operates in two steps:

1. **Squeeze:** Global average pooling is applied to compress spatial information into channel descriptors

2. **Excitation:** A simple gating mechanism with two fully-connected layers and a sigmoid activation function adaptively recalibrates channel-wise feature responses

By explicitly modeling channel interdependencies, SE blocks allow the network to boost useful features and suppress less useful ones, improving representation power with minimal computational overhead. This approach demonstrated significant improvements across various computer vision tasks including image classification.

2.2.2 Convolutional Block Attention Module (CBAM)

CBAM [6] extends the concept of attention further by incorporating both channel and spatial attention. While SE blocks focus solely on channel relationships, CBAM argues that both "what" (channel) and "where" (spatial) are important for enhancing feature representation.

CBAM sequentially applies two submodules:

1. **Channel attention module:** Unlike SE which uses only average-pooled features, CBAM leverages both average-pooled and max-pooled features to capture different aspects of channel information. These aggregated features are fed into a shared network to produce the channel attention map.

2. **Spatial attention module:** This module generates a spatial attention map by utilizing channel-pooled features (both average and max pooling along the channel axis) followed by a convolution operation.

The two attention maps are applied sequentially to refine the intermediate feature map. Extensive experiments on ImageNet classification, MS COCO object detection, and VOC 2007 detection demonstrate that CBAM consistently improves performance across various network architectures with minimal parameter increase.

2.2.3 Coordinate Attention

Coordinate Attention [2] represents a further evolution in attention mechanisms, addressing limitations of both channel and spatial attention. While channel attention (like SE) loses spatial information and standard spatial attention can be computationally expensive, Coordinate Attention efficiently encodes both channel and positional information.

Coordinate Attention decomposes the 2D spatial attention into two 1D feature encoding processes that capture the interdependencies between channels and spatial coordinates. By leveraging positional information along horizontal and vertical directions separately, it creates attention maps that maintain precise positional information while being computationally efficient.

This approach bridges the gap between channel and spatial attention mechanisms, providing a balanced solution that enhances representation power while preserving position-sensitive information. The coordinate-wise attention design enables networks to selectively emphasize informative features with specific spatial locations, which is particularly beneficial for tasks requiring precise spatial understanding such as object detection and semantic segmentation.

Through this progression from basic attention to more sophisticated mechanisms like Coordinate Attention, we can observe a continuous refinement of how neural networks attend to and emphasize important information, leading to increasingly powerful visual representations with relatively small computational overhead.

3. Experimentation and Discussion

3.1. Ablation and Architectural Experiments

We systematically experimented with key architectural and training parameters:

- **SE Block Ablation:** Removing Squeeze-and-Excitation blocks led to further degradation in identity preservation and increased noise, confirming their positive contribution to feature recalibration.

- **Normalization Strategies:** Switching from instance normalization to batch or layer normalization resulted in either mode collapse or ghosting effects, reaffirming the choice of instance normalization for style transfer tasks.

- **Loss Function Variations:** Increasing the cycle consistency loss weight improved structure but blurred stylistic features; perceptual loss based on VGG16 did not transfer well due to domain gap between natural and anime images.

- **Progressive Growing:** Training with progressive image resolutions (from 64×64 to 128×128) improved texture synthesis at lower resolutions but introduced phase artifacts at higher scales.

3.2. Custom Dataset and Data Augmentation

To address the shortcomings of the large-scale anime dataset, we curated a custom dataset of 4,000 Ghibli-style faces using Stable Diffusion, paired with a filtered subset of real faces from UTKFace. Data augmentation (random flips, rotations, color jitter) was employed to increase diversity and mitigate overfitting.

Results: Training on the custom dataset yielded more visually consistent and stylistically faithful outputs, with improved color harmony and better preservation of facial structure. However, some artifacts persisted, particularly when translating challenging poses or low-light images.

3.3. Qualitative Evaluation and Failure Analysis

Throughout all experiments, evaluation relied primarily on visual inspection of input-output pairs, as GAN loss metrics did not correlate well with perceptual quality. Successes were most prominent on frontal, well-lit faces with neutral expressions. Failure cases included:

- Profile views or occluded faces, which led to disfigurements and loss of identity.
- Reverse translation (anime to real) occasionally resulted in uncanny, overly smooth faces.
- Low-light or noisy images, which the model misinterpreted as stylistic shading.

3.4. Training on Large-Scale AnimeFace Dataset

Our initial experiments utilized the publicly available AnimeFace dataset, comprising over 63,000 images, in conjunction with the UTKFace dataset for the real face domain. The expectation was that the model, with its CycleGAN backbone and SE-enhanced residual blocks, would learn a generalizable mapping between photorealistic and anime-style faces, producing visually coherent cartoonish outputs.

Observations: Despite extensive training and regular visual validation on custom and held-out images, the outputs were frequently unstable and unpredictable. Generated images often exhibited:

- Severe artifacts such as checkerboard patterns and color blotches.
- Anatomically implausible features, including distorted facial proportions and unnatural blending of human and anime characteristics.
- Over-saturation, loss of facial identity, and inconsistent stylistic transfer.

Analysis: These issues can be attributed to several factors:

- *Domain Mismatch:* The AnimeFace dataset features a wide variety of stylizations, facial poses, and artistic conventions, making it difficult for the model to learn a consistent style mapping.
- *Preprocessing Limitations:* Face detection and cropping algorithms, optimized for real faces, often failed on stylized anime faces, leading to poorly aligned training data.
- *Resolution Constraints:* Downsampling to 128×128 pixels, while necessary for computational feasibility, led to loss of fine details crucial for both photorealism and anime aesthetics.

Summary: Our experimentation underscores the importance of domain-specific data curation, robust preprocessing, and architectural choices such as SE blocks and instance normalization. While the custom Ghibli-style dataset improved results, further work is needed to address failure cases and enhance generalization across diverse facial inputs.

4. Methodology and Implementation

4.1. System Architecture

Our framework is built on a dual-generator, dual-discriminator CycleGAN architecture tailored for robust, multi-style facial synthesis and age manipulation. This configuration enables unpaired image-to-image translation between photorealistic and Ghibli-style facial domains, as well as effective aging and de-aging transformations.

Generator G_A2B: Transforms real (photographic) faces into anime/Ghibli-style faces.

Generator G_B2A: Converts anime/Ghibli-style faces back to realistic faces.

Discriminator D_A: Judges the realism of generated or real faces in the photographic domain.

Discriminator D_B: Assesses the stylistic authenticity of faces in the anime/Ghibli domain.

Each generator is based on a ResNet-inspired architecture featuring:

1. An initial convolutional block (reflection padding, convolution, instance normalization, ReLU).
2. Two downsampling layers to increase feature depth.
3. Nine residual blocks with integrated Squeeze-and-Excitation (SE) modules for enhanced channel-wise attention.
4. Two upsampling layers for spatial reconstruction.
5. A final convolutional block with Tanh activation for output normalization.

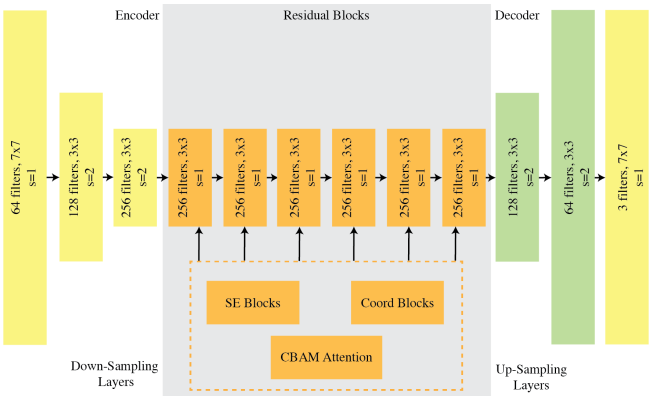

Figure 2. Attention Enabled ResNet generator

The discriminators adopt a PatchGAN-style CNN design, evaluating local image patches for domain authenticity, which is well established for stable and effective adversarial training in unpaired image translation tasks.

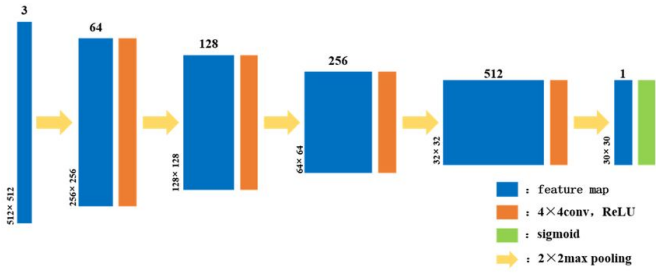

Figure 3. PatchGAN Discriminator Architecture

4.2. Loss Functions

We learn mappings between domains X and Y via generators $G : X \to Y$ and $F : Y \to X$, and discriminators D_X, D_Y. These discriminators enforce that generated samples match the distribution of real images in the target domain.

4.2.1 Adversarial Loss

The adversarial loss for generator G and discriminator D_Y is defined as:

$$\mathcal{L}_{\text{GAN}}(G, D_Y, X, Y) = \mathbb{E}_{y \sim p_{\text{data}}(y)}[\log D_Y(y)] + \mathbb{E}_{x \sim p_{\text{data}}(x)}[\log(1 - D_Y(G(x)))] \quad (1)$$

A similar loss is defined for F and D_X.

4.2.2 Cycle Consistency Loss

To ensure meaningful mappings, we enforce that translations are cycle-consistent:

$$\mathcal{L}_{\text{cyc}}(G, F) = \mathbb{E}_{x \sim p_{\text{data}}(x)}[\|F(G(x)) - x\|_1] + \mathbb{E}_{y \sim p_{\text{data}}(y)}[\|G(F(y)) - y\|_1] \quad (2)$$

4.2.3 Full Objective

The overall objective combines both adversarial and cycle-consistency losses:

$$\mathcal{L}(G, F, D_X, D_Y) = \mathcal{L}_{\text{GAN}}(G, D_Y, X, Y) + \mathcal{L}_{\text{GAN}}(F, D_X, Y, X) + \lambda \mathcal{L}_{\text{cyc}}(G, F) \quad (3)$$

Here, λ controls the relative importance of the cycle consistency loss.

4.3. Novel Approaches

4.3.1 Squeeze-and-Excitation (SE) Enhanced Residual Blocks

A key innovation in our architecture is the integration of Squeeze-and-Excitation (SE) blocks within each residual block of the generators. SE blocks perform dynamic, channel-wise feature recalibration, allowing the network to adaptively emphasize the most informative features for both style and age transformations. This is especially important in facial synthesis, where subtle cues (e.g., wrinkles, color gradients) are critical for photorealism and stylistic fidelity.

Mathematical Formulation:

$$\text{SE}(x) = x \cdot \sigma(W_2 \delta(W_1 \text{GAP}(x)))$$

where W_1, W_2 are learned transformations, GAP denotes Global Average Pooling, (reduction ratio $= 16$), δ = ReLU, and σ = Sigmoid.

```python
class SEBlock(nn.Module):
    def __init__(self, channel, reduction=16):
        super(SEBlock, self).__init__()
        self.fc = nn.Sequential(
            nn.AdaptiveAvgPool2d(1),  # Squeeze: output size (N, channel, 1, 1)
            nn.Conv2d(channel, channel // reduction, 1),
            nn.ReLU(inplace=True),
            nn.Conv2d(channel // reduction, channel, 1),
            nn.Sigmoid()  # Excitation: channel weights between 0 and 1
        )

    def forward(self, x):
        weights = self.fc(x)
        return x * weights  # channel-wise multiplication
```

Justification:

SE blocks have proven effective in both classification and generative tasks, improving the model's ability to focus on domain-specific details.

In our context, they allow the generator to better distinguish and preserve identity-relevant features during age transformation, and to capture stylistic nuances in Ghibli-style synthesis.

This approach is supported by recent literature demonstrating the benefits of attention and adaptive residual mechanisms in facial image generation and translation tasks.

4.3.2 Instance Normalization

We employ instance normalization throughout the architecture instead of batch normalization. Instance normalization is empirically superior for style transfer and unpaired image translation, as it normalizes each sample independently, preserving instance-specific style cues and supporting more stable GAN training.

4.3.3 PatchGAN Discriminators

Our discriminators are designed as PatchGANs, which classify overlapping **70×70** patches as real or fake. This local focus encourages the generator to produce realistic fine-grained details, a crucial property for both facial realism and stylization.

4.4. Dataset Engineering

4.4.1 Age Manipulation Domain

Parameter	Value
Source	UTKFace
Total Images	4000
Age range	Young: 18 - 30 Old: 50-120
Resolution	128×128

Preprocessing:

Images are filtered by age, face-cropped using RetinaFace/Haar cascades, resized to **128×128**, and normalized to **[1,1]**.

Augmentation: Horizontal flip, rotation, and color jitter are applied to increase diversity, improve generalization, and mitigate overfitting.

4.4.2 Ghibli-Style Domain

Preprocessing:

Synthetic Ghibli-style faces are generated via Stable Diffusion, face-cropped, resized, and normalized identically to the real-face pipeline.

Parameter	Value
Source	Custom Ghibli Dataset (StableDiffusion)
Total Images	4000
Resolution	128×128

Augmentation: Identical to the real domain to ensure balanced training and robust translation.

4.4.3 Data Loading

Custom PyTorch Dataset classes and DataLoaders are used for efficient batch processing and shuffling, ensuring each mini-batch contains diverse samples from both domains

4.5. Training Strategy and Stability

4.5.1 Loss Functions

- Adversarial Loss: Least-squares GAN loss for stable convergence.
- Cycle Consistency Loss (λ_{cycle}=10.0): Ensures that translating an image to the other domain and back reconstructs the original.
- Identity Loss ($\lambda_{identity}$=5.0): Encourages generators to preserve color composition and facial structure when the input is already in the target domain.

4.5.2 Optimization

- Adam optimizer, learning rate 0.0002, β_1 =0.5, β_2 = 0.999
- Batch size: 6, Epochs: 500 (selected based on GPU memory and empirical stability).
- Weight Initialization: Normal initialization for convolutional layers to promote stable early training.

4.5.3 Stability Enhancements

- Instance normalization for all convolutional layers.
- Regular checkpointing and visual inspection every 10 epochs.
- Training on diverse, augmented data to prevent mode collapse.
- Visual validation on held-out, user-supplied images.

4.5.4 Hardware

Training distributed across Kaggle (P100 GPUs), Colab (T4 GPUs), and local machines, with a 30-hour compute cap per session.

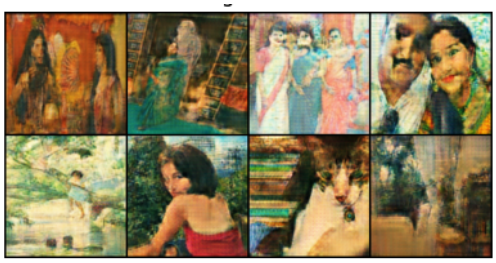

Figure 4. Ghibli Style transfer Results using SE-Block

4.6. Evaluation and Validation

4.6.1 Visual Evaluation

- Outputs are assessed by direct visual inspection, as GAN loss metrics do not reliably reflect perceptual quality in creative tasks.
- Identity preservation is verified using DeepFace for select transformations.
- Side-by-side input/output galleries and qualitative user feedback are used for model selection.

5. Results

5.1. Results for Ghibli using SE and coord blocks

From the results obtained in Figure 4 and Figure 5, we can visually observe that the outputs generated using Coordinate Attention are superior to those produced with Squeeze-and-Excitation (SE) blocks. . Coordinate Attention effectively encodes both channel relationships and precise positional information, allowing the model to more accurately focus on important spatial regions and enhance feature localization. This leads to improved visual quality and more faithful style transfer in generated images.

In summary, adopting a more advanced attention mechanism such as Coordinate Attention yields clear improvements in image generation quality over both SE-based and attention-free architectures, demonstrating the value of integrating direction-aware and position-sensitive attention modules in generative models

Attention Mechanism	Avg. Generator Loss	Epochs
SE	1.99	110
Coordinate Attention	1.817	134

Table 1. Comparison of average generator loss and epochs to convergence for models with SE and with coordinate attention.

5.2. Results of Young-Old model

Figure 6 and Figure 7 are the outputs on our own images generated from the demo application that we have deployed

Figure 5. Ghibli style transfer Results using Coordinate Attention LEFT: Generated Ghibli images — Right: Reconstructed Original Images

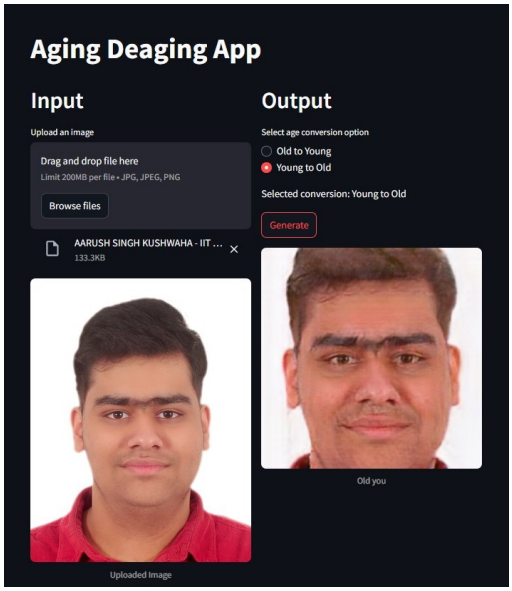

Figure 6. Sample image 1 generated using our pipeline

Attention Mechanism	Avg. Generator Loss	Epochs
None (Without Attention)	1.75	130
Coordinate Attention	1.05	66

Table 2. Comparison of average generator loss and epochs to convergence for models without attention and with coordinate attention.

6. Conclusion

In this project, we developed a comprehensive framework for multi-style image generation using advanced GAN

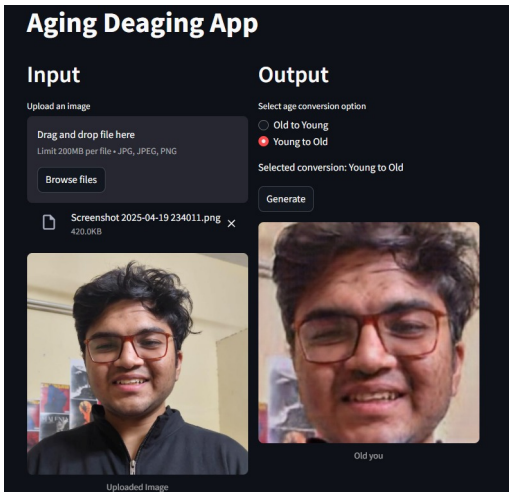

Figure 7. Sample Image 2 generated using our pipeline

architectures enhanced with multiple attention mechanisms, including Squeeze-and-Excitation, CBAM, and Coordinate Attention blocks.. Our system preserves important identity features while demonstrating strong unpaired image-to-image translation between real and Ghibli-style faces and efficient age manipulation. We addressed issues with domain mismatch, preprocessing constraints, and style fidelity by means of meticulous dataset curation, focused data augmentation, and innovative architectural designs. Although some artefacts persist in difficult situations like occluded or low-light images, qualitative evaluation demonstrated that our method yields visually convincing and identity-preserving results.Overall, our work highlights the potential of attention-augmented GANs for creative style and attribute transfer, and provides practical insights for future research at the intersection of generative modeling, digital art, and human-centered design

7. Appendix

7.1. Links to datasets

UtkFace Dataset Ghibli Dataset

7.2. Source Code

7.2.1 Utk-Face Dataloader (Young-Old)

```

import os
import shutil
from PIL import Image

# Define paths
input_dir = '/kaggle/input/utkface-new/UTKFace'  # Replace with the path to your UTKFace dataset
young_dir = '/kaggle/working/Input/young'  # Replace with the path where you want to save the young images
old_dir = '/kaggle/working/Input/old'  # Replace with the path where you want to save the old images

# Create directories if they don't exist
os.makedirs(young_dir, exist_ok=True)
os.makedirs(old_dir, exist_ok=True)

# Age brackets
young_min_age, young_max_age = 18, 30
old_min_age, old_max_age = 50, 120

# Function to get age from filename
def get_age_from_filename(filename):
    return int(filename.split('_')[0])

# Process images
for filename in os.listdir(input_dir):
    if filename.endswith('.jpg'):
        age = get_age_from_filename(filename)

        if young_min_age <= age <= young_max_age:
            if random.choice([0, 1]):
                shutil.copy(os.path.join(input_dir, filename), os.path.join(young_dir, filename))
        elif old_min_age <= age <= old_max_age:
            if random.choice([0, 1]):
                shutil.copy(os.path.join(input_dir, filename), os.path.join(old_dir, filename))

print("Images have been successfully divided and saved.")


import os
from PIL import Image
from torchvision import transforms,datasets
from torch.utils.data import Dataset, DataLoader

class CustomImageFolderDataset(Dataset):
    def __init__(self, root_dir, transform=None):
        self.root_dir = root_dir
        self.transform = transform
        self.images = os.listdir(root_dir)

    def __len__(self):
        return len(self.images)

    def __getitem__(self, idx):
        img_name = self.images[idx]
        img_path = os.path.join(self.root_dir, img_name)
        image = Image.open(img_path).convert('RGB')

        if self.transform:
            image = self.transform(image)

        label = 0 if 'young' in self.root_dir else 1  # Assign labels based on directory name

```

```
        return image, label


transform = transforms.Compose([
    transforms.Resize((128,128)),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])

young_dataset = CustomImageFolderDataset('/kaggle/working/Input/young', transform=transform)
old_dataset = CustomImageFolderDataset('/kaggle/working/Input/old', transform=transform)


# Create DataLoaders
batch_size = 6 # Adjust batch size as needed
young_loader = DataLoader(young_dataset, batch_size=batch_size, shuffle=True)
old_loader = DataLoader(old_dataset, batch_size=batch_size, shuffle=True)

# Example usage of the loaders
for inputs, labels in young_loader:
    # Process inputs and labels as needed
    print(f"Young batch shape: {inputs.shape}, Labels: {labels}")
    break

for inputs, labels in old_loader:
    # Process inputs and labels as needed
    print(f"Old batch shape: {inputs.shape}, Labels: {labels}")
    break
```

7.2.2 Ghibli Dataloader

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.autograd import Variable
import itertools
import random

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
import os
from PIL import Image
from torchvision import transforms,datasets
from torch.utils.data import Dataset, DataLoader

class GhibliImageTypeDataset(Dataset):
    def __init__(self, base_dir, image_type='g', transform=None):
        """
        image_type: either 'g' or 'o' to pick specific images.
        """
        self.transform = transform
        self.samples = []
        self.label = 1 if image_type == 'g' else 0

        for split in ['training', 'validation']:
            split_path = os.path.join(base_dir, split)
            if not os.path.exists(split_path):
                continue

            for folder in os.listdir(split_path):
                folder_path = os.path.join(split_path, folder)
                if os.path.isdir(folder_path):
                    img_path = os.path.join(folder_path, f'{image_type}.png')
                    if os.path.exists(img_path):
                        self.samples.append(img_path)

    def __len__(self):
        return len(self.samples)

    def __getitem__(self, idx):
        img_path = self.samples[idx]
        image = Image.open(img_path).convert('RGB')

        if self.transform:
            image = self.transform(image)

        return image, self.label

transform = transforms.Compose([
    transforms.Resize((140, 140)),                # Slightly upscale for cropping
    transforms.RandomCrop((128, 128)),            # Random crop to desired size
    transforms.RandomHorizontalFlip(),            # Flip images with 0.5 prob
    transforms.RandomRotation(15),                # Rotate images randomly within 15 degrees
    transforms.ColorJitter(
    # Randomly change brightness/contrast/etc.
        brightness=0.2, contrast=0.2, saturation=0.2, hue=0.1
    ),
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

transform_2 = transforms.Compose([
    transforms.Resize((128,128)),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])

# Datasets
g_dataset = GhibliImageTypeDataset('/kaggle/input/ghibli-dataset/Ghibli', image_type='g', transform=transform)
o_dataset = GhibliImageTypeDataset('/kaggle/input/ghibli-dataset/Ghibli', image_type='o', transform=transform)

# DataLoaders
batch_size = 6
g_loader = DataLoader(g_dataset, batch_size=batch_size, shuffle=True)
o_loader = DataLoader(o_dataset, batch_size=batch_size, shuffle=True)
```

```

# Sample usage
for inputs, labels in g_loader:
    print(f"G image batch shape: {inputs.shape}, Labels: {labels}")
    break

for inputs, labels in o_loader:
    print(f"O image batch shape: {inputs.shape}, Labels: {labels}")
    break
```

7.2.3 Helper Functions

```

import numpy as np
# Adversarial ground truths
def get_adversarial_targets(batch_size, device, output_shape):
    valid = torch.ones((batch_size, *output_shape), device=device, requires_grad=False)
    fake = torch.zeros((batch_size, *output_shape), device=device, requires_grad=False)
    return valid, fake

def weights_init_normal(m):
    if isinstance(m, nn.Conv2d):
        if m.weight is not None:
            nn.init.normal_(m.weight.data, 0.0, 0.02)
        if m.bias is not None:
            nn.init.constant_(m.bias.data, 0.0)

    elif isinstance(m, nn.BatchNorm2d):
        if m.weight is not None:
            nn.init.normal_(m.weight.data, 1.0, 0.02)
        if m.bias is not None:
            nn.init.constant_(m.bias.data, 0.0)
```

7.2.4 SE-Block

```
class SEBlock(nn.Module):
    def __init__(self, channel, reduction=16):
        super(SEBlock, self).__init__()
        self.fc = nn.Sequential(
            nn.AdaptiveAvgPool2d(1),  # Squeeze: output size (N, channel, 1, 1)
            nn.Conv2d(channel, channel // reduction, 1),
            nn.ReLU(inplace=True),
            nn.Conv2d(channel // reduction, channel, 1),
            nn.Sigmoid()  # Excitation: channel weights between 0 and 1
        )

    def forward(self, x):
        weights = self.fc(x)
        return x * weights
```

7.2.5 CBAM-Blocks(along with its helper functions)

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

class BasicConv(nn.Module):
    def __init__(self, in_planes, out_planes, kernel_size, stride=1, padding=0, dilation=1, groups=1, relu=True, bn=True, bias=False):
        super(BasicConv, self).__init__()
        self.out_channels = out_planes
        self.conv = nn.Conv2d(in_planes, out_planes, kernel_size=kernel_size, stride=stride, padding=padding, dilation=dilation, groups=groups, bias=bias)
        self.bn = nn.BatchNorm2d(out_planes,eps=1e-5, momentum=0.01, affine=True) if bn else None
        self.relu = nn.ReLU() if relu else None

    def forward(self, x):
        x = self.conv(x)
        if self.bn is not None:
            x = self.bn(x)
        if self.relu is not None:
            x = self.relu(x)
        return x

class Flatten(nn.Module):
    def forward(self, x):
        return x.view(x.size(0), -1)

class ChannelGate(nn.Module):
    def __init__(self, gate_channels, reduction_ratio=16, pool_types=['avg', 'max']):
        super(ChannelGate, self).__init__()
        self.gate_channels = gate_channels
        self.mlp = nn.Sequential(
            Flatten(),
            nn.Linear(gate_channels, gate_channels // reduction_ratio),
            nn.ReLU(),
            nn.Linear(gate_channels // reduction_ratio, gate_channels)
            )
        self.pool_types = pool_types
    def forward(self, x):
        channel_att_sum = None
        for pool_type in self.pool_types:
            if pool_type=='avg':
                avg_pool = F.avg_pool2d( x, (x.size(2), x.size(3)), stride=(x.size(2), x.size(3)))
                channel_att_raw = self.mlp( avg_pool )
            elif pool_type=='max':
                max_pool = F.max_pool2d( x, (x.size(2), x.size(3)), stride=(x.size(2), x.size(3)))
                channel_att_raw = self.mlp( max_pool )
            elif pool_type=='lp':
                lp_pool = F.lp_pool2d( x, 2, (x.size(2), x.size(3)), stride=(x.size(2), x.
```

```
    size(3)))
                channel_att_raw = self.mlp(
    lp_pool )
            elif pool_type=='lse':
                # LSE pool only
                lse_pool = logsumexp_2d(x)
                channel_att_raw = self.mlp(
    lse_pool )

            if channel_att_sum is None:
                channel_att_sum = channel_att_raw
            else:
                channel_att_sum = channel_att_sum
     + channel_att_raw

        scale = F.sigmoid( channel_att_sum ).
    unsqueeze(2).unsqueeze(3).expand_as(x)
        return x * scale

def logsumexp_2d(tensor):
    tensor_flatten = tensor.view(tensor.size(0),
    tensor.size(1), -1)
    s, _ = torch.max(tensor_flatten, dim=2,
    keepdim=True)
    outputs = s + (tensor_flatten - s).exp().sum(
    dim=2, keepdim=True).log()
    return outputs

class ChannelPool(nn.Module):
    def forward(self, x):
        return torch.cat( (torch.max(x,1)[0].
    unsqueeze(1), torch.mean(x,1).unsqueeze(1)),
    dim=1 )

class SpatialGate(nn.Module):
    def __init__(self):
        super(SpatialGate, self).__init__()
        kernel_size = 7
        self.compress = ChannelPool()
        self.spatial = BasicConv(2, 1,
    kernel_size, stride=1, padding=(kernel_size
    -1) // 2, relu=False)
    def forward(self, x):
        x_compress = self.compress(x)
        x_out = self.spatial(x_compress)
        scale = F.sigmoid(x_out) # broadcasting
        return x * scale

class CBAM(nn.Module):
    def __init__(self, gate_channels,
    reduction_ratio=16, pool_types=['avg', 'max'
    ], no_spatial=False):
        super(CBAM, self).__init__()
        self.ChannelGate = ChannelGate(
    gate_channels, reduction_ratio, pool_types)
        self.no_spatial=no_spatial
        if not no_spatial:
            self.SpatialGate = SpatialGate()
    def forward(self, x):
        x_out = self.ChannelGate(x)
        if not self.no_spatial:
            x_out = self.SpatialGate(x_out)
        return x_out
```

7.2.6 Coordinate Attention Blocks(along with helper functions)

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

class h_sigmoid(nn.Module):
    def __init__(self, inplace=True):
        super(h_sigmoid, self).__init__()
        self.relu = nn.ReLU6(inplace=inplace)

    def forward(self, x):
        return self.relu(x + 3) / 6

class h_swish(nn.Module):
    def __init__(self, inplace=True):
        super(h_swish, self).__init__()
        self.sigmoid = h_sigmoid(inplace=inplace)

    def forward(self, x):
        return x * self.sigmoid(x)

class CoordAtt(nn.Module):
    def __init__(self, inp, oup, reduction=32):
        super(CoordAtt, self).__init__()
        self.pool_h = nn.AdaptiveAvgPool2d((None,
     1))
        self.pool_w = nn.AdaptiveAvgPool2d((1,
    None))

        mip = max(8, inp // reduction)

        self.conv1 = nn.Conv2d(inp, mip,
    kernel_size=1, stride=1, padding=0)
        self.bn1 = nn.BatchNorm2d(mip)
        self.act = h_swish()

        self.conv_h = nn.Conv2d(mip, oup,
    kernel_size=1, stride=1, padding=0)
        self.conv_w = nn.Conv2d(mip, oup,
    kernel_size=1, stride=1, padding=0)


    def forward(self, x):
        identity = x

        n,c,h,w = x.size()
        x_h = self.pool_h(x)
        x_w = self.pool_w(x).permute(0, 1, 3, 2)

        y = torch.cat([x_h, x_w], dim=2)
        y = self.conv1(y)
        y = self.bn1(y)
        y = self.act(y)

        x_h, x_w = torch.split(y, [h, w], dim=2)
        x_w = x_w.permute(0, 1, 3, 2)

        a_h = self.conv_h(x_h).sigmoid()
        a_w = self.conv_w(x_w).sigmoid()

        out = identity * a_w * a_h

        return out
```

7.2.7 Common Resnet Blocks

```
class ResnetBlock(nn.Module):
    def __init__(self, dim, reduction=16):
        super(ResnetBlock, self).__init__()
        self.conv_block = self.build_conv_block(dim)

        self.attn = CoordAtt(dim,dim, reduction)
    # or SEBlock(dim, reduction) or CBAM(dim, reduction) accordingly

    def build_conv_block(self, dim):
        conv_block = [
            nn.ReflectionPad2d(1),
            nn.Conv2d(dim, dim, kernel_size=3, padding=0),
            nn.InstanceNorm2d(dim),
            nn.ReLU(True),
            nn.ReflectionPad2d(1),
            nn.Conv2d(dim, dim, kernel_size=3, padding=0),
            nn.InstanceNorm2d(dim)
        ]
        return nn.Sequential(*conv_block)

    def forward(self, x):
        out = self.conv_block(x)
        out = self.attn(out)
        return x + out
```

7.2.8 Generator and Discriminator Architecture

```
class GeneratorResNet(nn.Module):
    def __init__(self, input_nc, output_nc, n_residual_blocks=14):
        super(GeneratorResNet, self).__init__()

        # Initial convolution block
        model = [nn.ReflectionPad2d(3),
                 nn.Conv2d(input_nc, 64, 7),
                 nn.InstanceNorm2d(64),
                 nn.ReLU(inplace=True)]

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

        # Residual blocks
        for _ in range(n_residual_blocks):
            model += [ResnetBlock(in_features)]

        # Upsampling
        out_features = in_features // 2
        for _ in range(2):
            model += [nn.ConvTranspose2d(in_features, out_features, 3, stride=2, padding=1, output_padding=1),
                      nn.InstanceNorm2d(out_features),
                      nn.ReLU(inplace=True)]
            in_features = out_features
            out_features = in_features // 2

        # Output layer
        model += [nn.ReflectionPad2d(3),
                  nn.Conv2d(64, output_nc, 7),
                  nn.Tanh()]

        self.model = nn.Sequential(*model)

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


# Discriminator model
class Discriminator(nn.Module):
    def __init__(self, input_nc):
        super(Discriminator, self).__init__()

        model = [nn.Conv2d(input_nc, 64, 4, stride=2, padding=1),
                 nn.LeakyReLU(0.2, inplace=True)]

        model += [nn.Conv2d(64, 128, 4, stride=2, padding=1),
                  nn.InstanceNorm2d(128),
                  nn.LeakyReLU(0.2, inplace=True)
        ]

        model += [nn.Conv2d(128, 256, 4, stride=2, padding=1),
                  nn.InstanceNorm2d(256),
                  nn.LeakyReLU(0.2, inplace=True)
        ]

        model += [nn.Conv2d(256, 512, 4, stride=1, padding=1),
                  nn.InstanceNorm2d(512),
                  nn.LeakyReLU(0.2, inplace=True)
        ]

        model += [nn.Conv2d(512, 1, 4, stride=1, padding=1)]

        self.model = nn.Sequential(*model)

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

7.2.9 Training Loop

```
import torchvision
num_epochs = 500
def imshow(img):
    npimg = img.numpy()
    plt.imshow(np.transpose(npimg, (1, 2, 0)))
    plt.axis('off')
#     plt.show()|

```

```
# Training Loop
for epoch in range(num_epochs):
    shown = 0
    if (epoch%2):
        torch.save(netG_A2B.state_dict(), '/kaggle/working/netG_A2B.pth')
        torch.save(netG_B2A.state_dict(), '/kaggle/working/netG_B2A.pth')
        torch.save(netD_A.state_dict(), '/kaggle/working/netD_A.pth')
        torch.save(netD_B.state_dict(), '/kaggle/working/netD_B.pth')
    for i, (young_img, old_img) in enumerate(zip(o_loader, g_loader)):
        # Set model input
        real_A = Variable(young_img[0].to(device))
        real_B = Variable(old_img[0].to(device))
        # real_A = Variable(batch[0][batch[1]==0].to(device))
        # real_B = Variable(batch[0][batch[1]==1].to(device))

        # Adversarial ground truths
        # valid = Variable(torch.Tensor(np.ones((real_A.size(0), *D_A.output_shape[1:]))).to(device), requires_grad=False)
        # fake = Variable(torch.Tensor(np.zeros((real_A.size(0), *D_A.output_shape[1:]))).to(device), requires_grad=False)
        output_shape = netD_A(real_A).shape[1:]
        valid_A, invalid_A = get_adversarial_targets(real_A.size(0), device,output_shape)
        valid_B, invalid_B = get_adversarial_targets(real_B.size(0), device,output_shape)
        # ------------------
        #  Train Generators
        # ------------------
        # fake_A = fake_A.expand(-1, 3, -1, -1)
        # print(invalid_A.shape)
        # print(valid_A.shape)
        # print(netD_A(fake_A.detach()).shape)
        optimizer_G.zero_grad()

        # Identity loss
        loss_id_A = criterion_identity(netG_B2A(real_A), real_A)
        loss_id_B = criterion_identity(netG_A2B(real_B), real_B)

        loss_identity = (loss_id_A + loss_id_B) / 2

        # GAN loss
        fake_B = netG_A2B(real_A)
        loss_GAN_A2B = criterion_GAN(netD_B(fake_B), valid_B)
        fake_A = netG_B2A(real_B)
        loss_GAN_B2A = criterion_GAN(netD_A(fake_A), valid_A)

        loss_GAN = (loss_GAN_A2B + loss_GAN_B2A) / 2

        # Cycle loss
        recovered_A = netG_B2A(fake_B)
        loss_cycle_A = criterion_cycle(recovered_A, real_A)
        recovered_B = netG_A2B(fake_A)
        loss_cycle_B = criterion_cycle(recovered_B, real_B)

        loss_cycle = (loss_cycle_A + loss_cycle_B) / 2

        # Total loss
        loss_G = loss_GAN + 10.0 * loss_cycle + 5.0 * loss_identity

        loss_G.backward()
        optimizer_G.step()

        # -----------------------
        #  Train Discriminator A
        # -----------------------

        optimizer_D_A.zero_grad()

        # Real loss
        loss_real = criterion_GAN(netD_A(real_A), valid_A)
        # Fake loss (on batch of previously generated samples)
        loss_fake = criterion_GAN(netD_A(fake_A.detach()), invalid_A)
        # Total loss
        loss_D_A = (loss_real + loss_fake) / 2

        loss_D_A.backward()
        optimizer_D_A.step()

        # -----------------------
        #  Train Discriminator B
        # -----------------------

        optimizer_D_B.zero_grad()

        # Real loss
        loss_real = criterion_GAN(netD_B(real_B), valid_B)
        # Fake loss (on batch of previously generated samples)
        loss_fake = criterion_GAN(netD_B(fake_B.detach()), invalid_B)
        # Total loss
        loss_D_B = (loss_real + loss_fake) / 2

        loss_D_B.backward()
        optimizer_D_B.step()

        # --------------
        #  Log Progress
        # --------------
        if epoch % 10 == 0 and epoch !=0 and shown == 0:
            # Save model checkpoints
            torch.save(netG_A2B.state_dict(), f'/kaggle/working/netG_A2B_epoch{epoch}.pth')
            torch.save(netG_B2A.state_dict(), f'/kaggle/working/netG_B2A_epoch{epoch}.pth')
            torch.save(netD_A.state_dict(), f'/kaggle/working/netD_A_epoch{epoch}.pth')
```

```
            torch.save(netD_B.state_dict(), f'/kaggle/working/netD_B_epoch{epoch}.pth')
        if (i%100==0):
            print(f"[Epoch {epoch+1}/{num_epochs}] [Batch {i+1}/{len(g_loader)}] "
                  f"[D loss: {loss_D_A.item() + loss_D_B.item()}] "
                  f"[G loss: {loss_G.item()}, adv: {loss_GAN.item()}, cycle: {loss_cycle.item()}, identity: {loss_identity.item()}]")
```

References

[1] Ian J. Goodfellow et al. *Generative Adversarial Networks*. 2014. arXiv: 1406.2661 [stat.ML]. URL: https://arxiv.org/abs/1406.2661.

[2] Qibin Hou, Daquan Zhou, and Jiashi Feng. *Coordinate Attention for Efficient Mobile Network Design*. 2021. arXiv: 2103.02907 [cs.CV]. URL: https://arxiv.org/abs/2103.02907.

[3] Jie Hu et al. *Squeeze-and-Excitation Networks*. 2019. arXiv: 1709.01507 [cs.CV]. URL: https://arxiv.org/abs/1709.01507.

[4] Tero Karras, Samuli Laine, and Timo Aila. *A Style-Based Generator Architecture for Generative Adversarial Networks*. 2019. arXiv: 1812.04948 [cs.NE]. URL: https://arxiv.org/abs/1812.04948.

[5] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: 1706.03762 [cs.CL]. URL: https://arxiv.org/abs/1706.03762.

[6] Sanghyun Woo et al. *CBAM: Convolutional Block Attention Module*. 2018. arXiv: 1807.06521 [cs.CV]. URL: https://arxiv.org/abs/1807.06521.

[7] Jun-Yan Zhu et al. *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks*. 2020. arXiv: 1703.10593 [cs.CV]. URL: https://arxiv.org/abs/1703.10593.