ÖZYEĞİN UNIVERSITY
FACULTY OF ENGINEERING
DEPARTMENT OF COMPUTER SCIENCE

CS 402

2025 Spring

SENIOR PROJECT REPORT

GAN Made Pixel-Art Translation Using Deep-Learning Techniques

By
Kaan Ümit İŞMEN
Emre Batuhan YILDIRIM

Supervised By
Hasan Fehmi ATEŞ

Declaration of Own Work Statement/ (Plagiarism Statement)

Hereby I confirm that all this work is original and my own. I have clearly referenced/listed all sources as appropriate and given the sources of all pictures, data etc. that are not my own. I have not made any use of the essay(s) or other work of any other student(s) either past or present, at this or any other educational institution. I also declare that this project has not previously been submitted for assessment in any other course, degree or qualification at this or any other educational institution.

Student Name and Surname: Kaan Ümit İŞMEN - Emre Batuhan YILDIRIM

Signature:

Place, Date: 06/02/2025

Abstract

This report includes the detailed trajectory of our senior project, “GPA” gan made pixel art translation using deep-learning techniques, which was initiated and supervised by Hasan Fehmi ATEŞ. We aim to improve and utilize an existing GAN-based model to create PixelArt images. The GPA’s main objective is to take an existing image of the user and apply style transfer to a pixel art output. We observed that existing models can translate the images to a specific style but there aren’t any pixel art translators for this specific task. This will be done by modifying the existing GAN model that makes unpaired image translation. This model will be modified by adding new loss functions, fine-tuning the model for the desired output, and optimizing it to have multi-scale outputs. This project contributes to the field by addressing the gap in the pixel-art style transition and providing a novel approach to artistic image transformation. We utilized CycleGAN-Turbo model using Google Colab. We gathered an unsupervised dataset made for pixel-art generation. We have done over 10 different training experiments and made a user-friendly interface to incorporate different training checkpoints to generate Pixel-Art images from the uploaded images.

Contents

I. INTRODUCTION 4

II. BACKGROUND 6

III. PROBLEM STATEMENT 8

IV. SOLUTION APPROACH 9

V. RESULTS AND DISCUSSION 20

VI. RELATED WORK 24

VII. CONCLUSION AND FUTURE WORK 26

ACKNOWLEDGEMENTS 27

REFERENCES 28

APPENDIX 29

I. Introduction

With the evolution of technology and artificial intelligence, companies and users are expecting niche products to meet their needs. What we have chosen is a pixel-art conversion of provided pictures to be used by the people who plan to use graphics that are in pixel-art. Our aim was to shorten the period of the development stages of the pixel-art images that have a purpose in the usage of the communities which are the challenges of developing and converting user images.

The primary objective of the project to be answered is there are limited style translation models that can answer pixel art style translation operations. In the optimally designed style, translation models are structured around, for example, changing a person's hair color, etc. Initiated by Hasan Fehmi Ateş, this project aims to enhance the existing image-to-image translation models to answer the pixel art style translations in a user-friendly manner and lower the output result time for companies planning to transition an image to the pixel art style. The centerpiece of this initiative is to fine-tune the models to work in a style transition.

Example Input 1

Example Output 1

These examples of input and output are from our fine-tuned CycleGAN-Turbo[1] model that we added a palette loss function.

In this senior project as we stated in the first part (CS401), we aim to generate scalable pixel-art images by using deep learning techniques. We are using the model named as CycleGAN-turbo[1] and we are trying to compare multiple training sessions from different downscaling preferences, implemented or excluded loss functions and also by their performance to get the most optimal output based on the comparison results.

In the background we have done several procedures in order to initialize the most optimal training environment. We have gathered a unpaired dataset containing 4000 real-world and 2000 pixel-art images for training. Versioning the model for the expected outcomes. We came across a problem when we first tried our training in our local machines (RTX3070 GPU). We kept getting Out Of Memory error so we came up with a solution consists of setting the environment inside Google Colab[2]. We purchased Colab Pro+ for 500+ computation units and A100 GPU which has 40GB of VRAM. This made our trainings quite possible due to its high performance and availability of VRAM.

We are utilizing the CycleGAN-Turbo[1] model to provide a scalable, efficient, and accessible solution for Pixel-Art style transfer generation. These advancements will give an advantage to users to integrate Pixel-Art to their creative works with minimal effort. The report is organized as follows:

Background

This section presents a detailed overview of the technical approaches, tools, and related technologies that underpin our project. Rather than merely listing each component, we explain how and why each tool is used, as well as its specific role within the overall workflow.

Problem Statement

Presenting the problem, we explain what use will this project achieve and also who will benefit from it.

Solution Approach

We will be presenting our solution regarding the problem. Containing detailed explanation of the models architecture and approaches we have made.

Related Work

We are going to compare existing unsupervised image-to-image translation methods although we couldn’t find any regarding pixel-art generation.

Conclusion and Future Work

We are going to state what has been achieved so far and what could be improved in any future development phases.

Acknowledgements

This section recognizes the contributions of colleagues, advisors, and organizations that provided support, feedback, or resources for the project.

References

This section provides a list of all references cited in the report, formatted according to IEEE guidelines.

Appendix

Appendix section is for additional images related to the project like more inference images and examples from our dataset.

II. Background

This section presents a detailed overview of the technical approaches, tools, and related technologies that underpin our project. Rather than merely listing each component, we explain how and why each tool is used, as well as its specific role within the overall workflow. By describing both the purpose and functionality of every element, we aim to provide a clear understanding of how these technologies integrate to support our pixel‑art translation pipeline.

Generative Adversarial Networks (GANs)

Generative Adversarial Networks (GANs), introduced by Ian Goodfellow in 2014, present a dual‑network framework in which two neural networks—known as the generator and the discriminator—compete in a zero‑sum game. The generator's task is to synthesize images that resemble real data, while the discriminator's task is to distinguish between real and generated images. As the generator produces increasingly convincing outputs, the discriminator becomes more proficient at detecting subtle artifacts, and this feedback loop forces both networks to improve. In practice, the generator learns to approximate the true data distribution by trying to "fool" the discriminator, and the discriminator learns to refine its decision boundary to identify fakes. This adversarial training allows GANs to create high‑quality samples without explicitly modeling the underlying probability distribution.

CycleGAN‑Turbo

CycleGAN‑Turbo extends the classical CycleGAN architecture to achieve faster convergence, greater stability, and enhanced image quality during translation between unpaired domains. While CycleGAN relies on two generators and two discriminators with a cycle‑consistency constraint (an image translated to the target domain and back should closely match its original), CycleGAN‑Turbo further incorporates encoder–decoder structures in its generator networks and contrastive learning techniques in its discriminators. Encoder–decoder architectures help capture multi‑scale features more effectively, and contrastive learning encourages the discriminator to separate real and generated samples more distinctly. Additionally, CycleGAN‑Turbo integrates a Denoising Diffusion Probabilistic Model (DDPM) scheduler for inference, which enables faster image generation while preserving fine details. In our workflow, we invoke the model using two main scripts: inference_unpaired.py to generate outputs from pretrained or fine‑tuned models, and train_cyclegan_turbo.py to carry out the training routine on unpaired image datasets. Edge preprocessing for input images is handled by functions defined in image_prep.py, ensuring that contours are extracted and simplified before entering the adversarial network.

Google Colab Pro+ A100 GPU Environment

Initial experiments on a local workstation equipped with an NVIDIA RTX 3070 GPU (8 GB VRAM) consistently produced "Out Of Memory" errors when processing images at 512 × 512 resolution. To overcome this limitation, we transitioned to Google Colab Pro+, which provides an NVIDIA A100 GPU with 40 GB of VRAM. This upgrade enabled us to specify larger batch sizes and operate at higher resolutions without memory failures. The Colab environment was synchronized with Google Drive, allowing us to store checkpoints, log files, and intermediate visual results in an organized manner. Furthermore, this setup permitted parallel experimentation: collaborators could launch multiple training sessions simultaneously without resource conflicts.

PyTorch and PyCharm Usage

All core model development—data preprocessing, augmentation, and implementation of custom loss functions—was performed in PyTorch. We specifically employed the torchvision.transforms module to execute a pipeline of transformations (e.g., Resize, RandomCrop, HorizontalFlip) on each image, which increased the variety of input configurations during training and improved model generalization. The codebase, including scripts such as training_utils.py, train_cyclegan_turbo.py, and image_prep.py, was managed within the PyCharm IDE. PyCharm's debugging, code navigation, and project management features streamlined the iterative process of refining training loops, validating custom modules, and ensuring that each component behaved as intended.

Edge Detection: Canny with Min‑Area Filtering

Pixel‑art translation demands clear, simplified contours rather than intricate, high‑frequency details. To achieve this, we began with the classic OpenCV function cv2.Canny but modified its usage in two key ways. First, we adjusted the threshold parameters to suppress faint or spurious edges: the default configuration used lower and upper thresholds of 100 and 200, respectively; the medium configuration used 125 and 200; and the low configuration used 150 and 200. Increasing the lower threshold progressively removes weaker edge responses, so that only the most salient outlines remain. Second, we incorporated a minimum‑area filtering step via OpenCV's connectedComponentsWithStats function. This procedure measures the pixel area of every connected region in the binary edge map and discards any region whose area falls below a chosen threshold (for example, 200 pixels). By eliminating these tiny clusters of edge pixels, we obtained cleaner, more stylized edge masks that align with pixel‑art conventions and reduce unwanted noise.

Palette Loss

Conventional pixelwise loss functions—such as L1 loss or perceptual metrics like LPIPS—do not fully capture the discrete color constraints of pixel art, where palettes are limited and contrasts are sharp. To address this, we implemented a palette loss that compares the global color distributions of generated images to those of real pixel‑art examples. First, each image channel is represented by a soft histogram with sixteen bins; we then smooth these histograms using a Gaussian kernel to account for slight color variations. The palette loss is defined as the L1 distance between the real and generated histograms for each channel. By minimizing this distance, the network is guided to produce outputs whose color distributions match the sharp, limited palettes characteristic of handcrafted pixel art. As a result, the generator avoids gradual color gradients and instead favors distinct, high‑contrast color choices that adhere to authentic pixel‑art style.

Dataset Structure and Usage

Our dataset consists of two unpaired categories: (1) real‑world photographs (4,000 images) depicting a variety of scenes landscapes, buildings, and objects—and (2) pixel‑art images (2,000 samples) with tightly constrained color palettes. Because CycleGAN‑Turbo is designed for unpaired translation, we do not require matching pairs of photographs and pixel art; instead, each training iteration randomly selects one example from each domain. Prior to entering the network, all images are passed through a function

named build_transform, which rescales them to multiple target resolutions (64 × 64, 128 × 128, or 256 × 256). By organizing our data in this unpaired manner, both generators learn to map domain‑specific features without relying on explicit one‑to‑one correspondences.

III. Problem Statement

The primary objective of the project to be answered is there are limited style translation models that can answer pixel art style translation operations. In the optimally designed style, translation models are structured around, for example, changing a person's hair color, etc. Initiated by Hasan Fehmi Ateş, this project aims to enhance the existing image-to-image translation models to answer the pixel art style translations in a user-friendly manner and lower the output result time for companies planning to transition an image to the pixel art style. The centerpiece of this initiative is to fine-tune the models to work in a style transition.

We are utilizing the CycleGAN-Turbo[3] model to provide a scalable, efficient, and accessible solution for Pixel-Art style transfer generation. Achieve this objective of project, (1) our aim to add scalability for converting image across 512x512, 256x256, 128x128 and 64x64 sizes. These flexibility allows to users change the resolution of the image and if want user can choose the image sizes in the output of the training. (2) Fine-tuning already existing Canny edge detection functions in order to get lesser details due to pixel-art styling reasons which in fact have lesser details in the image without losing the original images traits. These changes allowed user to change the how much detail that they want with default (already existing edge detection thresholds without min_area function), medium (low_threshold=125, min_area=150) and low (low_threshold=150, min_area=200). (3) We also aimed to introduce a new loss function in order to enhance the image for generating pixel-art images. Which for this case was Pallette-Loss. It is a differentiable loss function designed to compare two images by evaluating their **color distribution similarity** rather than pixel-wise accuracy.

IV. Solution Approach

Our aim was to fine-tune and train the existing CycleGan-Turbo[3] to be compatible with translating an image to be pixel-art. In the first initial training we used our RTX3070 systems and faced several obstacles. One of them is out of memory error due to the fact that in the training part the already existing model uses 512x512 resolution. And the other one is finding a dataset related to our aim. After finding a related dataset we have to add scalability and downscale the 512x512 resolution with 256x256, 128x128, 64x64 and 32x32.

- **Scalability:** Our aim for developing this part to be compatible with our system is whether we can start training and decide if we can add scalability to our work. In here we have fine-tuned the already existing build_transform function in training_utils.py.

```
elif image_prep in ["resize_32", "resize_32x32"]:
    T = transforms.Compose([
        transforms.Resize((32, 32), interpolation=Image.LANCZOS),
        transforms.RandomCrop((32, 32)),
        transforms.RandomHorizontalFlip(),

    ])
```

First we tried with 32x32 for resizing of the image. If users have written for resize_32 or resize_32x32 in training starting terminal code this block will be runned. In this block, it uses:

- transforms.Resize((32, 32), interpolation=Image.LANCZOS) :

In here it resizes the input image with 32x32 pixels and with the usage of LANCZOS algorithm from Python Imaging Library Format[18], it provides high-quality downsampling

- transforms.RandomCrop((32, 32)):

Here it randomly crops an 32x32 region of the resized image but it has no visible effect since it has already been 32x32. Although it has no effect, it adds randomness during training. which helps with effective dataset usage

- transforms.RandomHorizontalFlip():

In here it flips the image horizontally with randomized chance, providing data-augmentation to help the model generalize better

These steps have been followed for the other resizing parts

```
elif image_prep in ["resize_64x64", "resize_64"]:
    T = transforms.Compose([
        transforms.Resize((64, 64), interpolation=Image.LANCZOS),
        transforms.RandomCrop((64,64)),
        transforms.RandomHorizontalFlip(),
    ])
```

```
elif image_prep in ["resize_128", "resize_128x128"]:
    T = transforms.Compose([
        transforms.Resize((128, 128), interpolation=Image.LANCZOS),
        transforms.RandomCrop((128, 128)),
        transforms.RandomHorizontalFlip(),
    ])
```

```
elif image_prep in ["resize_256", "resize_256x256"]:
    T = transforms.Compose([
        transforms.Resize((256, 256), interpolation=Image.LANCZOS)
    ])
```

We wanted our output images to be more consistent in pixel art style thus we fine-tuned the thresholds at canny_from_pil function in image_prep.py which uses thresholds to get the edge detection. By changing the lower_threshold value to higher value, it detects fewer edges and this makes it closer to a pixel-art style image. This first edge filtering fine-tune that we added in the model is the adjusting of the already existing edge thresholds and the second edge filtering fine-tuned function adding the min-area function. These fine-tune's helped us reduce the overall details in order to move to a pixel-art style look in the image. With this it helps us to preserve important edges. We kept the default Canny-thresholds also renamed as default and added two more presets which are medium (low_threshold=125) and low(low_threshold=150). However with these fine-tunes, raising the thresholds wasn't enough for the small edges in the image to disappear. To solve the issue we added the min-area function which filters the edges. This filtering does measure the pixel area of each connected component in Canny output and discards any specific region smaller than the dictated area. By combining these functions now reflect the pixel-art style more.

With these we needed to adjust the loss-functions. Originally the generators in the old train_cyclegan_turbo.py were using three separate backward-passes to generate images. These were cycle consistency, gan based adversarial term and finally for identity it uses LPIPS loss. These losses have each individual optimizer steps and scheduler updates.These, optimizer steps and generator updates were split into three separate back-propagation steps which are (1) Cycle-Consistency loss which reconstruct A->B->A and B->A->B, (2) Adversarial GAN loss which fools each discriminator on generated A and B images, lastly Identity loss which does makes the generator output the same image when no style change is needed. These multi-step duplicates graph retention calls which increases memory usage and can lead to imbalanced gradient signals since each objective is applied in isolation thus, it increases peak GPU usage. In our fine-tuned version, we combined these losses in a single backward pass. This reduces peak memory usage but not the total usage. But adding palette loss to these changes bumps the GPU memory needed for computation which gives an "out of memory" error. Because of this we implemented these changes separately. To be consistent with the aim of the pixel-art style, color schemes tend to be limited and simplified colors. Thus we added differentiable palette loss based on soft histograms. In the soft histogram we compute histogram for each channel which for both the generated and real images. At first, we put the pixel values into a fixed number of bins(for example 16) using a small Gausian kernel to allow gradidents to flow. Then it takes L1 distance for between two histograms and uses it as our palette loss. By penalizing differences in the soft histogram, the generator is encouraged to match the overall distribution of the target. This approach effectively nudges the model toward a simpler, higher-contrast palette, in line with pixel-art style. To look at more details

- **Edge Threshold and Min-Area:** We have kept the already existing canny image thresholds as default, added medium and low thresholds. Aim of this changes, to reduce the details of the images since pixel art style relies on simplified edge maps. Raising Canny threshold automatically gets rid of weak edges otherwise creates a lot more details which gets away from the idea of the pixel-art style. Min area filtering

Goal: By raising the lower canny threshold, we force the detector ignore thin edges due to pixel-art styling concerns

```
1   import numpy as np
2   from PIL import Image
3   import cv2
4
5
6   def canny_from_pil(image, low_threshold=100, high_threshold=200):
7       image = np.array(image)
8       image = cv2.Canny(image, low_threshold, high_threshold)
9       image = image[:, :, None]
10      image = np.concatenate([image, image, image], axis=2)
11      control_image = Image.fromarray(image)
12      return control_image
```

In the original canny_from_pil function, the low_threshold values set to 100. That means any gradient magnitude below 100 were discarded. However magnitudes between 100 and 200 which is high_threshold still can be seen in the output image. Due to pixel-art styling concerns, we want to remove more faint lines and keep the more stronger ones. To achieve this, we simply raised the low_threshold parameter. Below is the exact code snipped that we fine-tuned

```
5 usages  new *
def canny_from_pil(image, low_threshold=100, high_threshold=200):
    image = np.array(image)
    image = cv2.Canny(image, low_threshold, high_threshold)
    image = image[:, :, None]
    image = np.concatenate([image, image, image], axis=2)
    return Image.fromarray(image)

2 usages  new *
def filtered_canny(image, low_threshold, high_threshold, min_area):
    gray = cv2.cvtColor(np.array(image), cv2.COLOR_RGB2GRAY)
    edges = cv2.Canny(gray, low_threshold, high_threshold)
    num_labels, labels, stats, _ = cv2.connectedComponentsWithStats(edges, connectivity=8)
    filtering = np.zeros_like(edges)
    for label in range(1, num_labels):
        if stats[label, cv2.CC_STAT_AREA] >= min_area:
            filtering[labels == label] = 255
    rgb = np.stack([filtering]*3, axis=2)
    return Image.fromarray(rgb)

new *
def default(image):
    return canny_from_pil(image)

new *
def medium(image, low_threshold=125, high_threshold=200, min_area=100):
    return filtered_canny(image, low_threshold, high_threshold, min_area)

new *
def low(image, low_threshold=150, high_threshold=200, min_area=200):
    return filtered_canny(image, low_threshold, high_threshold, min_area)
```

With our threshold change functions we can achieve gradient filtering which does different lower threshold values. We can change which edges that we can keep and discard by simply changing the low_thresholds.

- medium (low_threshold=125) → any gradient value below 125 is discarded and above 125 have been kept
- low(low_threshold=150) → any gradient value below 150 is discarded and above 150 have been kept

After the threshold adjustment, a few small clusters of edge pixels can remain because in this it looks at the strong edges and eight connected with strong edges it still keeps it. We remove these by discarding any connected component whose area is below the predetermined min_area. In the original code there wasn't any filtered_canny function so it is implemented by us. In below we can see step by step explanation of the overall code that we fine-tuned and added:

A) gray = cv2.cvtColor(np.array(image), cv2.COLOR_RGB2GRAY)

with this np.array(image) turns the input image, which is a PIL image, into a numpy array with shape (H,W,3). cv2.cvtColor(cv2.COLOR_RGB2GRAY) converts that three channel array to a two channel array. cv.Canny waits for grayscale (single-input) input to compute gradients. If it passed the RGB image, OpenCV would merge channels internally and results could be inconsistent. By converting explicitly, we know exactly which gradients we measure.

B) edges = cv2.Canny(gray, low_threshold, high_threshold)

B1) pixels with gradients over than high_threshold becomes strong edge
B2) pixels with gradients between low_threshold and high_treshold become weak edge which stays if the one of the connected edges is strong edge
B3) pixels with gradients lower than low_threshold immediately discarded

C) num_labels, labels, stats, _ = cv2.connectedComponentsWithStats(edges, connectivity=8)

cv2.connectedComponentsWithStats scans for a binary edge map for every connected group of 255-valued pixels.

D) filtering = np.zeros_like(edges)

This creates a brand new array, filtering of the same shape (H,W) and type edges, but filled with zeros. In the end, filtering becomes our final edge map filtering become our final edge map before stacking in to RGB

E) for label in range(1, num_labels):

```
    if stats[label, cv2.CC_STAT_AREA] >= min_area:
        filtering[labels == label] = 255
```

This part effectively removes all small, isolated edge speckels. We start from label equals one due to label zero is the background. Then it reads the area of the label and checks whether the area of the label is big enough to keep. After that it uses boolean indexing for finding all connected pixels belonging to that component and setting them to the value 255.

F) rgb = np.stack([filtering] * 3, axis=2)

Neural network pipeline expects a three channel image by replacing the single channel mask into R, G, B this maintains compatibility without changing the downstream code. Filtering (H, W) is single channel then it multiplies with three which creates a list of three identical references. np.stack stacks them along a new third dimension and produces the shape (H, W, 3)

G) return Image.fromarray(rgb)

This takes the three channel rgb turns back into a PIL image whether it’s medium or low filter applied.

3) <u>Loss Functions</u>: In the first train_cyclegan_turbo.py, the generator underwent updates in three distinct phases. which are accelerator.backward(), then called optimizer_gen.step() and lr_scheduler_gen.step(). Here we can take a look at the original codes

A1) Cycle-Consistency Loss

Ensures that translating an image from domain A to domain B and then back to domain A recovers the original image (and likewise for B→A→B). By penalizing the difference between the reconstructed image and its original (often using L1 distance plus a perceptual metric), this loss forces the generators to learn bijective mappings rather than arbitrary mappings. In practice, it prevents mode collapse and keeps structure.

$$\mathcal{L}_{\text{cycle}} = \mathbb{E}_{a\sim p(A)}\Big[||G_{B\to A}(G_{A\to B}(a))-a||_1+\lambda_{\text{perc}}\, d_{\text{LPIPS}}(G_{B\to A}(G_{A\to B}(a)),a)\Big]+\mathbb{E}_{b\sim p(B)}\Big[||G_{A\to B}(G_{B\to A}(b))-b||_1+\lambda_{\text{perc}}\, d_{\text{LPIPS}}(G_{A\to B}(G_{B\to A}(b)),b)\Big]$$

Objective: Motivate the model to replicate input sequences in:

- A →fake B → reconstructed A
- B → fake A → reconstructed B.

```python
"""
Cycle Objective
"""
# A -> fake B -> rec A
cyc_fake_b = CycleGAN_Turbo.forward_with_networks(img_a, "a2b", vae_enc, unet, vae_dec, noise_scheduler_1step, timesteps, fixed_a2b_emb)
cyc_rec_a = CycleGAN_Turbo.forward_with_networks(cyc_fake_b, "b2a", vae_enc, unet, vae_dec, noise_scheduler_1step, timesteps, fixed_b2a_emb)
loss_cycle_a = crit_cycle(cyc_rec_a, img_a) * args.lambda_cycle
loss_cycle_a += net_lpips(cyc_rec_a, img_a).mean() * args.lambda_cycle_lpips
# B -> fake A -> rec B
cyc_fake_a = CycleGAN_Turbo.forward_with_networks(img_b, "b2a", vae_enc, unet, vae_dec, noise_scheduler_1step, timesteps, fixed_b2a_emb)
cyc_rec_b = CycleGAN_Turbo.forward_with_networks(cyc_fake_a, "a2b", vae_enc, unet, vae_dec, noise_scheduler_1step, timesteps, fixed_a2b_emb)
loss_cycle_b = crit_cycle(cyc_rec_b, img_b) * args.lambda_cycle
loss_cycle_b += net_lpips(cyc_rec_b, img_b).mean() * args.lambda_cycle_lpips
accelerator.backward(loss_cycle_a + loss_cycle_b, retain_graph=False)
if accelerator.sync_gradients:
    accelerator.clip_grad_norm_(params_gen, args.max_grad_norm)

optimizer_gen.step()
lr_scheduler_gen.step()
optimizer_gen.zero_grad()
```

Although we set retain_graph=false ,which tells PyThorch to release the computational graph and its intermediate activations immediately after invokes loss.backward(), LPIPS certain immediate activations still needed to be retained. After backpropagation of loss_cycle_a+loss_cycle_b, parts of the computational graph either remained allocated or were recomputed thereafter, resulting in increased peak memory usage

A2) GAN Adversarial Loss

GAN Adversarial Loss: Comes from a minimax game between a generator G and a discriminator D. The generator's objective is to produce images that the discriminator cannot distinguish from real samples, while the discriminator's objective is to correctly classify real versus generated images

discriminator loss: $\mathcal{L}_D = -\mathbb{E}_{x\sim p_{\text{data}}(x)}[\log D(x)] - \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))]$
generator loss: $\mathcal{L}_G = -\mathbb{E}_{z\sim p_z(z)}[\log D(G(z))]$

Objective: Fool each discriminator into beliving generated images as authentic.

```python
"""
Generator Objective (GAN) for task a->b and b->a (fake inputs)
"""
fake_a = CycleGAN_Turbo.forward_with_networks(img_b, "b2a", vae_enc, unet, vae_dec, noise_scheduler_1step, timesteps, fixed_b2a_emb)
fake_b = CycleGAN_Turbo.forward_with_networks(img_a, "a2b", vae_enc, unet, vae_dec, noise_scheduler_1step, timesteps, fixed_a2b_emb)
loss_gan_a = net_disc_a(fake_b, for_G=True).mean() * args.lambda_gan
loss_gan_b = net_disc_b(fake_a, for_G=True).mean() * args.lambda_gan
accelerator.backward(loss_gan_a + loss_gan_b, retain_graph=False)
if accelerator.sync_gradients:
    accelerator.clip_grad_norm_(params_gen, args.max_grad_norm)
optimizer_gen.step()
lr_scheduler_gen.step()
optimizer_gen.zero_grad()
optimizer_disc.zero_grad()
```

Here, every backward pass requires another optimizer update. The computing graph is constructed and dismantled, incurring additional memory and computational costs. Furthermore, adversarial gradients applied independently of cycle consistency gradients may result in gradient imbalance.

A3) Identity (LPIPS) Loss

Combines a simple identity mapping penalty with a learned perceptual metric (LPIPS). The identity portion ensures that if an image already belongs to the target style, the generator should produce an identical output. On top of that, the LPIPS component measures perceptual similarity by comparing deep‑feature activations in a pretrained network. By penalizing differences in those high‑level features, LPIPS helps

preserve fine details and global consistency so identity mapping retains both pixel-level and perceptual likeness.

$$d_{\text{LPIPS}}(x, y) = \sum_{l} \frac{1}{H_l W_l} \sum_{h=1}^{H_l} \sum_{w=1}^{W_l} \|\mathbf{w}_l \odot (\phi_l(x)_{hw} - \phi_l(y)_{hw})\|_2^2$$

Objective: Ensure the generator maintains identity when a style alteration is unnecessary.

- A → A
- B → B

```
"""
Identity Objective
"""
idt_a = CycleGAN_Turbo.forward_with_networks(img_b, "a2b", vae_enc, unet, vae_dec, noise_scheduler_1step, timesteps, fixed_a2b_emb)
loss_idt_a = crit_idt(idt_a, img_b) * args.lambda_idt
loss_idt_a += net_lpips(idt_a, img_b).mean() * args.lambda_idt_lpips
idt_b = CycleGAN_Turbo.forward_with_networks(img_a, "b2a", vae_enc, unet, vae_dec, noise_scheduler_1step, timesteps, fixed_b2a_emb)
loss_idt_b = crit_idt(idt_b, img_a) * args.lambda_idt
loss_idt_b += net_lpips(idt_b, img_a).mean() * args.lambda_idt_lpips
loss_g_idt = loss_idt_a + loss_idt_b
accelerator.backward(loss_g_idt, retain_graph=False)
if accelerator.sync_gradients:
    accelerator.clip_grad_norm_(params_gen, args.max_grad_norm)
optimizer_gen.step()
lr_scheduler_gen.step()
optimizer_gen.zero_grad()
```

A third backward call and optimizer update. This raises memory consumption even more; once more, gradients from cycle or GAN losses have no bearing on the identity update in a given batch.

To address these issues, we combine all three generator objectives; cycle, GAN, and identity into a single loss. The steps are:

1. Calculate every component independently; do not call accelerator.backward(...) until all are totaled.
2. Sum them into one scalar called total_gen_loss..
3. On total_gen_loss, do a single backward pass then one optimizer step and one scheduler step.

```
total_gen_lost = ((loss_cycle_a + loss_cycle_b) + (loss_gan_a + loss_gan_b) + (loss_idt_a + loss_idt_b))
```

```
accelerator.backward(total_gen_lost)
if accelerator.sync_gradients:
    accelerator.clip_grad_norm_(params_gen, args.max_grad_norm)
optimizer_gen.step()
lr_scheduler_gen.step()
optimizer_gen.zero_grad()
optimizer_disc.zero_grad()
```

3.4 Implementation Details & Minor Adjustments

- No retain_graph=True : Because cycle, GAN, and LPIPS share core forward computations (e.g., the UNet encoder/decoder), a single backward pass can reuse those activations. We removed any retain_graph=True flags that the original code used to allow multiple backward calls.

- Gradient Clipping: We still perform gradient clipping once immediately after accelerator.backward(total_gen_loss) to prevent exploding gradients.

  if accelerator.sync_gradients:
  accelerator.clip_grad_norm_(params_gen, args.max_grad_norm)

- Single Zero‑Grad : We call zero_grad() only after the combined update to avoid clearing gradients prematurely.

  optimizer_gen.zero_grad()

  Edge‑Filtering Impact:

- The generator's forward passes (cyc_fake_b, fake_b, idt_a, etc.) now receive "edge‑filtered" inputs from the data pipeline.

- This does not change the loss formulas themselves—only the inputs used for computing those losses.

- Because palette loss is excluded here, this combined approach remains within GPU memory even with edge filtering active.

4) Palette Loss: We have implemented a Palette Loss function which is a differentiable loss function designed to compare two images by evaluating their color distribution similarity rather than pixel-wise accuracy. Unlike standard image losses (e.g., MSE or L1 pixel-wise loss), the palette loss measures differences in the overall color distribution through differentiable histograms. It consists of two main steps:

1. Soft Histogram Computation

To measure color distribution similarity in a differentiable manner, the palette loss uses a **soft histogram**, constructed as follows:

- **Input:**
  Given an image tensor x of shape (B,C,H,W) with pixel values in range [0,1], it aims to create a histogram representation of the pixel intensities for each channel.

- **Bin Centers:**
  The histogram bins are represented by evenly spaced **bin centers** between a minimum and maximum value (typically 0 and 1).
  For example, for 16 bins, bin centers might be:
  0.0, 0.066, 0.133, …, 0.933, 1.0
- **Soft Assignment:**
  Each pixel contributes to each histogram bin using a Gaussian-based **soft-assignment** method. This contrasts with traditional histograms, where each pixel belongs strictly to one bin.

  The contribution of a pixel to a bin b is calculated using a Gaussian function:

  $$\text{weight}_{i,b} = \exp\left(-\frac{(x_i - \text{bin_center}_b)^2}{2\sigma^2}\right)$$

  - x_i is the pixel intensity,
  - bin_center_b is the intensity at the center of bin b,

  σ is a hyperparameter determining the softness of assignment (smaller σ gives sharper assignment).

- **Histogram Construction:**
  The final **soft histogram** for each channel is obtained by summing these weights across all pixels:

  $$\text{histogram}_{c,b} = \sum_i \text{weight}_{i,b}$$

- **Normalization:**
  The histogram is then normalized per channel to sum to 1:

  $$\text{histogram}_{c,b}^{\text{normalized}} = \frac{\text{histogram}_{c,b}}{\sum_b \text{histogram}_{c,b} + \epsilon}$$

  This normalization ensures the histogram represents a probability distribution of colors within each

channel.

2. Palette Loss Calculation

Once soft histograms are computed for the **real** and **fake** images (typically a generated output and a ground-truth target), the palette loss measures their difference:

- **L1 Distance Between Histograms:**
  The palette loss is simply the L1 distance between the two normalized histograms, averaged over batches and channels:

$$\text{Palette Loss} = \frac{1}{B \cdot C} \sum_{b=1}^{\text{bins}} \left|\text{histogram}_b^{\text{fake}} - \text{histogram}_b^{\text{real}}\right|$$

This loss captures the **overall difference in color distributions** rather than exact pixel matching. It encourages the generated image to **match the color palette of the target image**, even if the precise spatial details differ.

Tools Used:

1) Pycharm: Usage of pycharm relies upon already existing and can be easily downloaded libraries for the environment
2) Pytorch: The general academic papers in the public use PyTorch library and it can be implemented for developing mode
3) Google Colab A100 GPU: The outline of the existing model developed optimally for this GPU that has 40GB VRAM
4) LoRA[12]: Low-Rank Adaptation of Large Language Models, LoRA[12] allows us to train some dense layers in a neural network indirectly by optimizing rank decomposition matrices of the dense layers' change during adaptation instead, while keeping the pre-trained weights frozen [4]
5) CycleGAN Turbo: Our base model [3]
6) Google Drive: Our model file container for Google Colab

Feasibility:

1. Operational: Evaluation of the algorithm can be utilized with careful management of the components used in the development environment to ensure compatibility and seamless communication between the components
2. Financial: Except for GPU which is a primary cost, using open-source tools minimizes the financial impact of the project
3. Engineering Standards and Skill Utilization: Skills utilized as following:
    - Data Knowledge
    - Machine Learning
    - Deep Learning
    - Training Capability
    - Image-to-Image Translation
    - Generative Adversarial Networks (GANs)[4]

Advantages:

This project will speed up the process of Pixel-Art conversion of images compared to hand-work. It will enhance our knowledge of GAN[4] architectures and image translation tasks by building on existing deep learning projects. Because this project is scalable through 256x256, 128x128, 64x64, 32x32 and scalable edge filtering with min-area and different thresholds we can observe that the outer styling of the image can be close to hand drawn pixel-art style. Not only these features also addition of the palette loss makes the pixel-art stylization adds another point of the pixel-art style.

Disadvantages:

Since the user base is quite niche, the potential audience for Pixel-Art conversion tools is relatively small which affects the appeal of a broader user base. Also data collection will be a difficult task since there isn't any proper datasets available for us to use and the available datasets are limited. Also the due to long training times and high GPU need before and after the fine-tuned version, taken time for the training gets bit higher, palette loss and edge loss changes applied differently

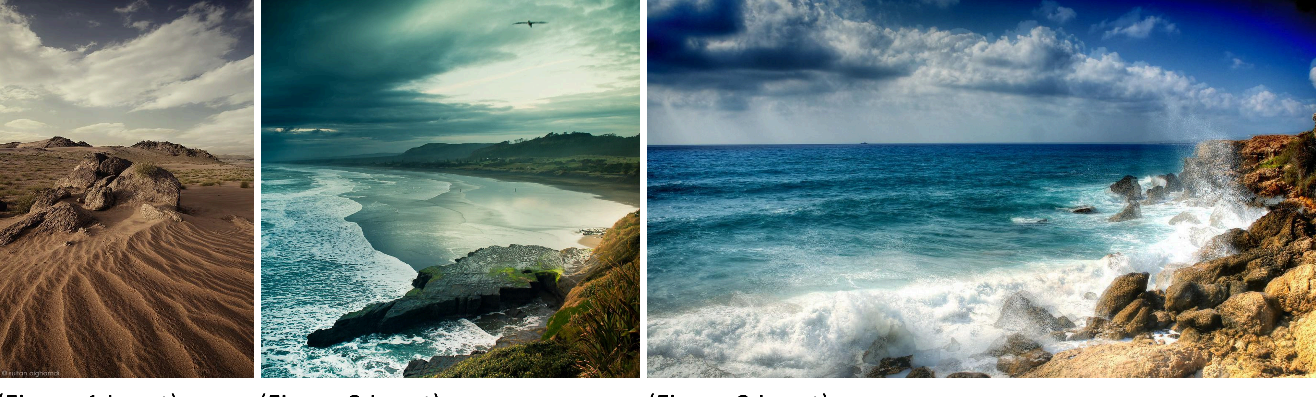
We have incorporated knowledge from CS454 (Introduction to Artificial Nerual Networks and Machine Learning) and CS466 (Introduction to Deep Learning) courses mostly. It helped us understand the principles of a Deep Learning model, how can we utilize trainings for our needs, what to optimize and how can we use inference principle when we finilized our trainings.

V. Results and Discussion

Since we have done a certain amount of modifications, we had to initiate multiple amounts of training. Here is a list of every training that we have done so far:

- 256x256 default
- 128x128 default
- 64x64 default
- 256x256 palette loss
- 128x128 palette loss
- 64x64 palette loss
- 256x256 medium edge threshold
- 128x128 medium edge threshold
- 128x128 low edge threshold
- 64x64 medium edge threshold
- 64x64 low edge threshold

Our test folder contained 3 landscape images in order to validate every desired step. In our case it was every 300 steps. They are the following 3 images:



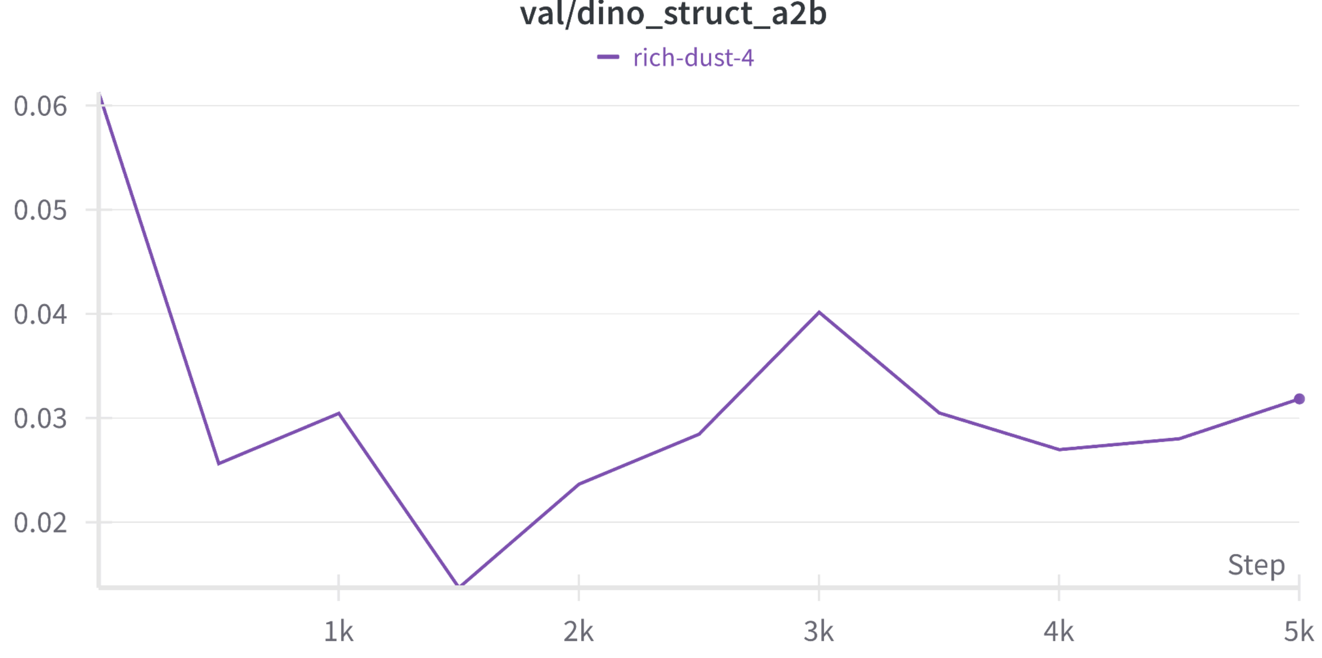
(Figure 1 Input) (Figure 2 Input) (Figure 3 Input)

In the first phase we wanted to observe the models unmodified output so we had done a 5000 step training. Here are the outputs in the 5000th step:

(Figure 4 Output) (Figure 5 Output) (Figure 6 Output)

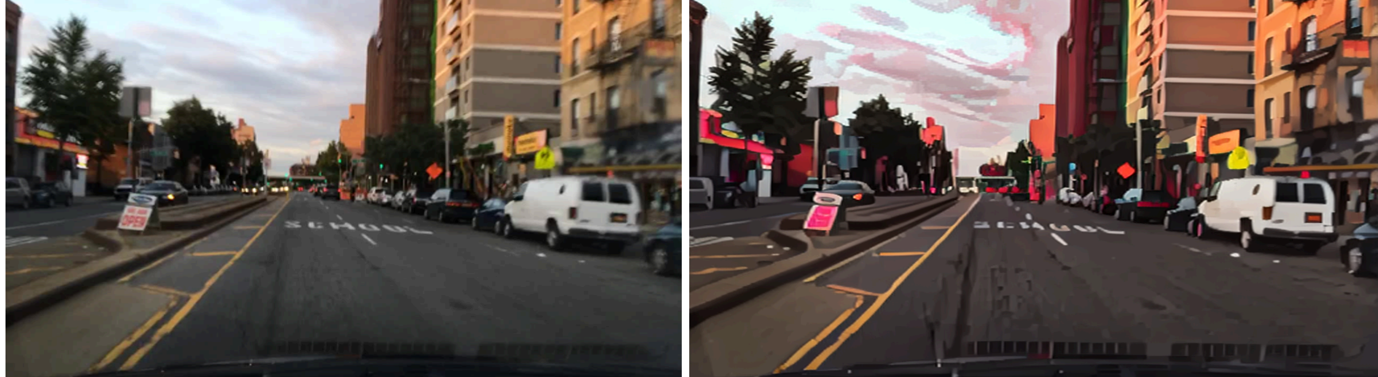
Although FID is a standard metric for generative model evaluation, due to resource and runtime limitations, we restricted our test set to 3 images per evaluation cycle. As FID requires a larger sample size to be meaningful (typically ≥500 images), we instead rely on **DINO feature distance** (dino(a2b)) as a primary metric.

The DINO metric offers a robust measure of **semantic similarity** between input and translated images, making it particularly suitable for pixel-art and style-transfer tasks where structure preservation is important.



This is our dino_struct chart regarding our 5000 step original model training. It can be observed that there is a slight variation between 0.05 and 0.01 values. This interval is quite acceptable in terms of semantic similarity between inputs and outputs.

Then, we took our latest checkpoint and tried to run an inference to our pretrained model. Here are the raw input and output:



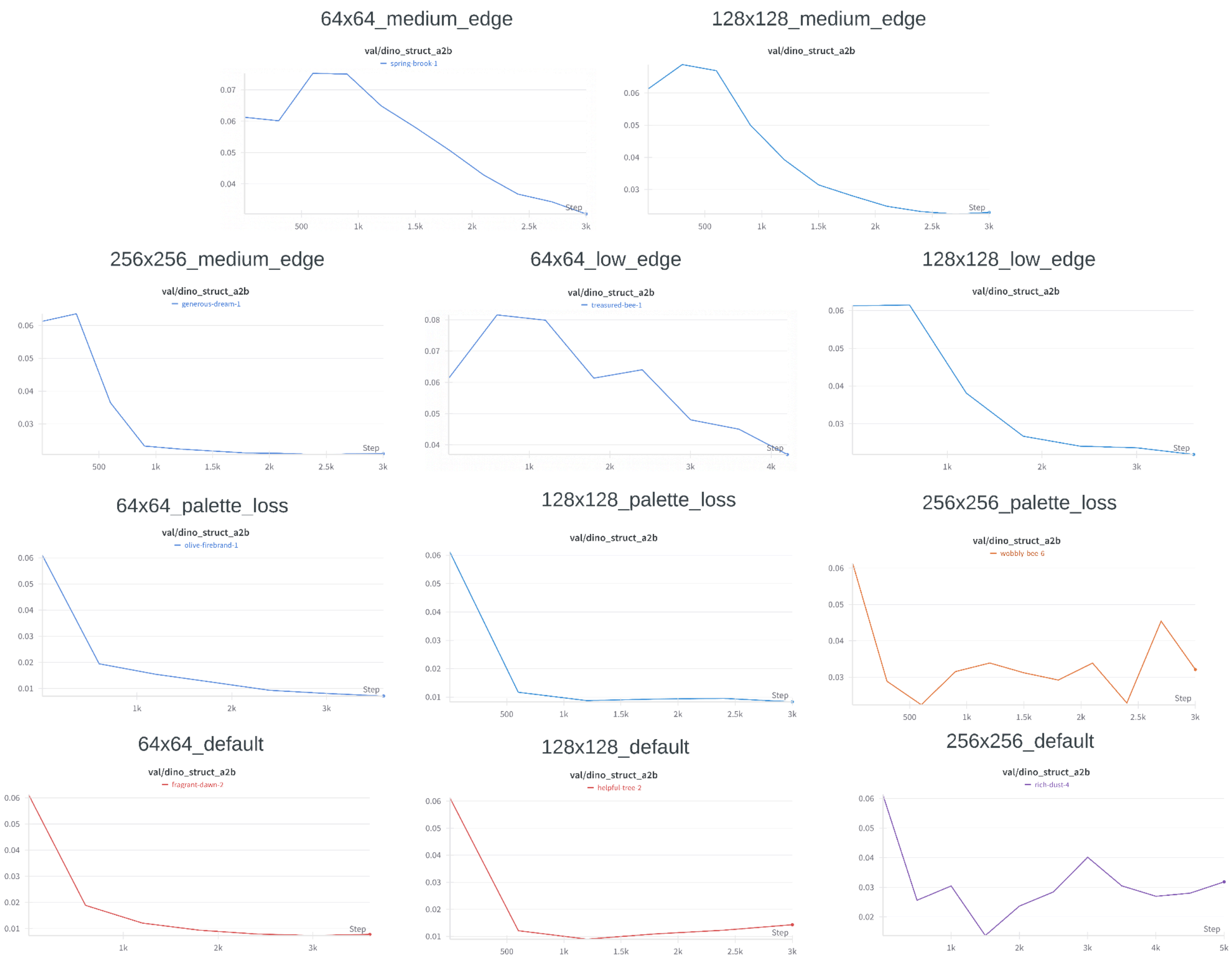
(Figure 7 Input) (Figure 8 Output)

Instead of evaluating each training by their own, we have prepared a single inference scheme for every training we have done. We have selected a colorful image of our campus to show how the trained models handle the pixel-art generation process. The table is as following:

(Figure 9 Every Inference Output of Our Trained Models)

Since art products are quite subjective, we wanted to have multiple options for pixel art generation. We are quite satisfied with palette_loss function outputs. They preserve the color nicely and pixelate the image just like it is in a retro game.

After seeing the results, we wanted to observe each models dino_struct graph so we can compare their performances regarding the semantic similarity of inputs and outputs. Its table is available as following:



(Figure 10 Every Dino-Struct Table of Our Trained Models)

We are satisfied from the outputs and metrics that have been recorded so far. Although we are well aware that further step trainings can achieve much better results regarding the pixel-art generation process. Future work will be discussed later in this paper.

VI. Related Work

Unpaired Image Translation Methodologies

Unpaired image translation techniques have swiftly acquired prominence in creative visual production problems because they obviate the necessity for labeled, paired information. CycleGAN is a notable technique in this field that facilitates domain-to-domain translation by imposing cycle-consistency loss, ensuring that a picture changed from domain A to B and subsequently reverted to A retains its structural integrity. Although CycleGAN has demonstrated effectivity in numerous style transfer applications, it encounters difficulties in minimalist or stylized domains like pixel art, where abstraction and simplification are paramount. The original CycleGAN architecture often maintains intricate details and does not reduce visual detail. In response, CycleGAN-Turbo[3] was developed, featuring additions like an encoder-decoder generator, a contrastive discriminator, and a DDPM-based inference scheduler to augment training efficiency and stability. Nonetheless, CycleGAN-Turbo is deficient in domain-specific methods like edge abstraction and color quantization, which are essential for pixel-art stylization. Therefore, our approach is founded on the CycleGAN-Turbo architecture, incorporating task-specific preprocessing and tailored loss functions to address the distinct visual requirements of pixel painting.

Investigations on Pixel Art Utilizing CycleGAN

A prominent study examining CycleGAN's applicability to pixel-art generation is the research conducted by Seo et al., entitled “Image-to-Pixel-Art Translation Based on CycleGAN.” The authors expanded a dataset from 5,000 to 10,000 pictures by mirroring and utilized L1 and LPIPS losses to direct the generator. Although the results were aesthetically pleasing, the model was deficient in explicit edge filtering, resulting in noisy and uneven contours. Furthermore, the absence of palette control led to unsatisfactory color gradients in the output. This represents a significant deficiency in pixel art, where strict color limitations and precise outlines are needed. We rectify these deficiencies by using an adaptive Canny edge recognition algorithm with threshold presets and minimum area filtering, effectively eliminating minor, redundant structures from the edge maps. Furthermore, we employ a palette loss function that evaluates soft histograms of both produced and authentic pixel-art images, directing the model to utilize fixed color palettes and minimizing post-processing requirements. These enhancements yield outputs that more accurately embody the stylized, low-resolution aesthetic characteristic of genuine pixel art.

Proprietary and Open-Source Pixel Art Software

Alongside academic methodologies, numerous commercial and open-source programs have been created for pixel-art conversion. Nevertheless, most of these tools execute just fundamental pixelization, lacking functionalities such as style-driven transformation, edge structure regulation, or palette administration. Therefore, while the outputs may initially appear to be pixel art, they frequently lack the artistic coherence, contour precision, and visual simplicity characteristic of genuine pixel-art aesthetics.

An explicit illustration of these constraints is shown in the Hugging Face application “NoCrypt/pixelization,” which underwent thorough testing during our project. This utility downsamples high-resolution photos to low-resolution formats (e.g., 32×32 or 64×64) and implements direct pixelation. Nonetheless, throughout this process, no structural filtering is implemented, no edge simplification is executed, and color palettes remain unconstrained. Moreover, the model demonstrates considerable quality

deterioration at resolutions of 256×256 and above, signifying its restricted scalability. Although structural integrity is somewhat maintained, it lacks adequate stylistic control and abstraction to be considered authentic pixel art.

Conversely, our technology does not merely execute pixelization. Initially, it employs adaptive edge detection and minimum-area filtering to streamline structural features, subsequently utilizing a palette loss function to direct the output's color distribution towards a specified color palette. The resulting graphics are both pixelated and artistically styled while maintaining semantic consistency. Furthermore, our system functions consistently across a range of sizes, from 32×32 to 256×256, rendering it appropriate for both low-resolution art creation and extensive visual assignments. Unlike most tools that simply downscale photographs and utilize block-style quantization, our method executes this transformation in a deliberate, style-focused, and learning fashion.

VII. Conclusion and Future Work

We have achieved successful image style transfers regarding pixel-art generation. We learned much about convolutional neural networks. How does existing models work and how they can be finely tuned in order to generate desired outputs through various amounts of trainings. Since our working conditions were highly undercutted by computational power and limited external GPU sources, we couldn't achieve the best results. Even so, we have made formidable progress regarding the matter. Implemented a variety of methods and observed their differences by their outputs. Since art themed images are a subjective topic, many of our trained model checkpoints can be suitable for pixel-art generation morley developed furtherly for more accurate and optimal pixel-art image generations.

On the economic and social impact side, our project holds substantial promise by enabling efficient and accessible pixel-art generation from diverse image inputs. This project can benefit digital artists, indie game developers, and much more. By automating the creation of pixel-art images, creators can significantly reduce both production costs and time, which fosters innovation and economic opportunities, especially when it comes to indie and small-scale projects. Also, allowing users without extensive artistic skills to produce visually appealing pixel art images is one of the social aspect of this project.

About world and society health, environmental, and safety, our project shows good results. By means of digitalization, artistic processes naturally lower material waste related to conventional art supplies, so benefiting the environment. Furthermore, fast prototyping visual assets can encourage better workflows for digital artists, so lessening of physical strain from hand-made creation. But our efforts also raise possible security and safety issues. Unauthorized users run dangers to communities, especially underprivileged or vulnerable groups, by using the instrument to create damaging or inappropriate images.

Legally, our work begs questions about content ownership and intellectual property rights. If input images are not legally licensed or sourced, created images could unintentionally violate copyright materials. As the project grows, clearly defined policies, content moderation systems, and intellectual property law compliance will be especially important.

Ethically, our main concern is possible abuse of pixel-art generating technologies. Issues include possible cultural appropriation, creation of misleading or damaging material, and illegal portrayal of people. Dealing with these problems and creating a reliable, safe user environment will depend critically on well defined ethical guidelines, community standards, and responsible use policies.

Future work directions for this project include enhancing computational efficiency through model optimization techniques, enabling broader accessibility on consumer-grade hardware. Also broadening the dataset and initiating longer step training will help generate more robust and reliable pixel-art images. Additionally, incorporating robust content moderation systems and user verification could mitigate misuse risks. Refining the neural network architectures will further improve the quality and versatility of generated outputs. Exploring real-time generation capabilities and integrating interactive user interfaces for customizable pixel-art production would significantly enhance user engagement and tool applicability.

VIII. Acknowledgements

This section expresses our heartfelt gratitude to those who have supported and contributed to our project. We extend our profound thanks to Professor Hasan Fehmi ATEŞ, who has been instrumental throughout the research of this project. His guidance was invaluable; he provided numerous paths and offered continuous advice, helping us navigate through various challenges and inspiring us to achieve our objectives. His contributions were not just educational but motivational, deeply influencing the successful progression of our work.

IX. References

[1] Transforming Real-life Photos into Pixel Art with SDXL - r/StableDiffusion - defensez0one (https://www.reddit.com/r/StableDiffusion/comments/17bkr4m/transforming_reallife_photos_into_pixel_art_with/?rdt=57910)

[2]Junyoung Seo, Gyuseong Lee, Seokju Cho, Jiyoung Lee, Seungryong Kim “MIDMs: Matching Interleaved Diffusion Models for Exemplar-based Image Translation”

[3] Gaurav Parmar, Taesung Park, Srinivasa Narasimhan, Jun-Yan Zhu “One-Step Image Translation with Text-to-Image Models”

[4] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio “Generative Adversarial Nets”

[5]Zhou Wang, Member, IEEE, Alan C. Bovik, Fellow, IEEE. Hamid R. Sheikh, Student Member, IEEE, and Eero P. Simoncelli, Senior Member, IEEE “Image Quality Assessment: From Error Visibility to Structural Similarity”

[6] Pan Zhang1 ∗ , Bo Zhang2, Dong Chen2, Lu Yuan3, Fang Wen2 1 University of Science and Technology of China 2 Microsoft Research Asia 3 Microsoft Cloud+AI “Cross-domain Correspondence Learning for Exemplar-based Image Translation”

[7] Xiaojuan Qi CUHK, Qifeng Chen Intel Labs, Jiaya JiaCUHK, Vladlen Koltun Intel Labs”Semi-parametric Image Synthesis”

[8] Roey Mechrez⋆ , Itamar Talmi⋆, Lihi Zelnik-Manor “The Contextual Loss for Image Transformation with Non-Aligned Data”

[9] Yunjey Choi1,2 Minje Choi1,2 Munyoung Kim2,3 Jung-Woo Ha2 Sunghun Kim2,4 Jaegul Choo1,2, 1 Korea University 2 Clova AI Research, NAVER Corp., 3 The College of New Jersey 4 Hong Kong University of Science & Technology “StarGAN v2: Diverse Image Synthesis for Multiple Domains”

[10]T. Karras, T. Aila, S. Laine, and J. Lehtinen. Progressive growing of GANs for improved quality, stability, and varia-tion. In ICLR, 2018.)

[11]Hu, E.J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., Chen, W.: Lora: Low-rank adaptation of large language models. In: International Conference on Learning Representations (ICLR) (2022)

[12]Zhang, L., Rao, A., Agrawala, M.: Adding conditional control to text-to-image diffusion models. In: IEEE International Conference on Computer Vision (ICCV) (2023)

[13] Image-to-pixel-art Translation Based on CycleGAN DOI: 10.54254/2753-8818/83/2025.20028

[14] Curated Pixel Art 512x512 https://www.kaggle.com/datasets/artvandaley/curated-pixel-art-512x512

[15] Landscape Pictures https://www.kaggle.com/datasets/arnaud58/landscape-pictures?select=00000000_%283%29.jpg

[16] Denoising Diffusion Probabilistic Model [2006.11239] Denoising Diffusion Probabilistic Models

[17] An Introduction to Variational Autoencoders https://doi.org/10.1561/2200000056

[18] Python Imaging Library Python: Pillow (a fork of PIL) | GeeksforGeeks

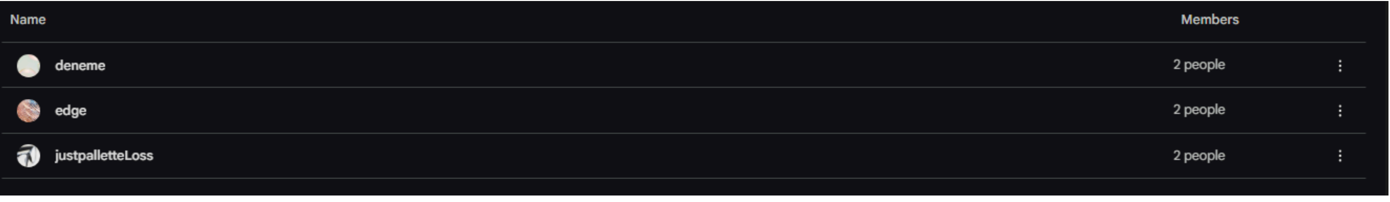
[19] NumPY NumPy

[20] Canny Canny Edge Detection Step by Step in Python — Computer Vision | by Sofiane Sahir | TDS Archive | Medium

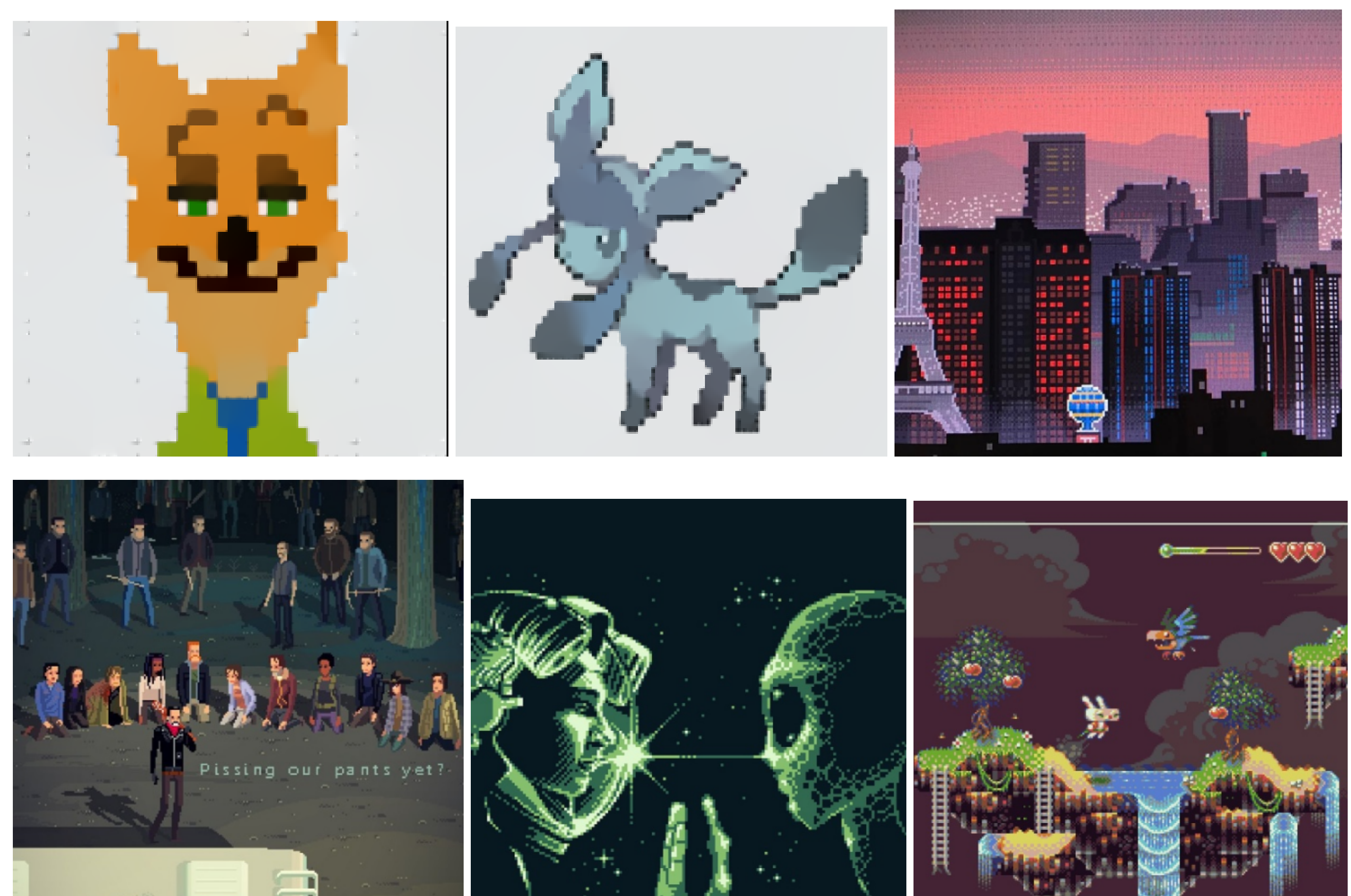
[21] DDPM https://arxiv.org/abs/2006.11239

[22] colab colab.google

IX. Appendix



Our 3 shareddrives in order to execute the model via Google Colab.



Some examples of our dataset containing over 2000 pixel-art images.