CS424 – Group Project Report

Akeela Darryl Fattha
akeelaf.2022@scis.smu.edu.sg

Fadhel Erlangga Wibawanto
fadhelew.2022@scis.smu.edu.sg

Tan Zhi Rong
zhirong.tan.2022@scis.smu.edu.sg

Grace Angel Bisawan
gbisawan.2022@scis.smu.edu.sg

Lee Jia Heng
jiaheng.lee.2023@scis.smu.edu.sg

Abstract

This report provides an overview of Cycle-Consistent Adversarial Networks (CycleGAN), a technique for unpaired image-to-image translation. We explore the architecture, key innovations, applications, and limitations of CycleGAN models in computer vision tasks.

Contents

1. Task 1

1.1 Introduction

We attempt to create a CyleGAN using novel and modern techniques to improve the performance of the model to convert cartoon faces to realistic faces, and vice-versa. We will be using improved blocks, various attention mechanisms and normalisation techniques in our generators, as well as additional loss functions to better guide the models to learn the mapping between the two domains. We opted not to use any augmentations to our images, and we use an image size of 256x256 as our input/output size.

With our enhancements, we achieved the following results:

	FID	**IS**	**GMS**	**Avg GMS**
Raw to Cartoon	46.70193	2.41466 ± 0.24392	4.39783	3.85666
Cartoon to Raw	39.83461	3.62382 ± 0.23992	3.31548	

1.1.1 Configuration & Hyperparameters

All optimisers used Adam with a Cosine Annealing Warm Restarts scheduler. The following hyperparameters and loss weights were used for our best results:

- Random seed: 42
- Learning rate: 2e-4
- Betas: (0.5, 0.999)
- $Scheduler_{t-0}$: 10
- $Scheduler_{t-mult}$: 2
- $Scheduler_{min-lr}$: 1e-5
- λ_{cyc}: 10.0
- λ_{id}: 5.0
- λ_{gan}: 1.0
- λ_{gp}: 10.0
- λ_{fm}: 5.0

1.2 Architecture

1.2.1 Generator

Our generator incorporates several modern attention mechanisms and architectural techniques including Convolutional Block Attention Module (CBAM), Squeeze-and-Excitation (SE) blocks, and Self-Attention mechanisms for improved feature representation, generating higher-quality images with better detail preservation, style consistency, and structural coherence. Some of our key design choices are listed below:

- **Instance Normalization:** Used throughout the network instead of batch normalization, as it has been shown to produce better results for style transfer and image-to-image translation by normalizing each instance independently.
- **Reflection Padding:** Applied before convolutions to reduce boundary artifacts that can appear in generated images, particularly important for maintaining realistic edges.
- **Mixed + Improved Residual Blocks:** The strategic placement of different residual block types allows the network to benefit from complementary approaches to feature transformation. They also exhibit the following characteristics and sub-blocks:
  - **Channel Attention:** Models interdependencies between channels using both max and average pooling operations.
  - **Spatial Attention:** Focuses on important spatial regions by creating attention maps from channel-wise statistics.v
  - All blocks benefit from SE attention for channel recalibration
- **Squeeze-and-Excitation Attention:** Used in Mixed Residual Blocks, it recalibrates channel-wise feature responses by explicitly modeling interdependencies between channels, allowing the network to selectively emphasize informative features.

- **Enhanced Self-Attention:** Positioned after the residual blocks, it helps ensure global coherence in the generated images by modeling long-range dependencies that convolutional operations cannot capture efficiently. This is particularly valuable for maintaining structural integrity in facial regions.

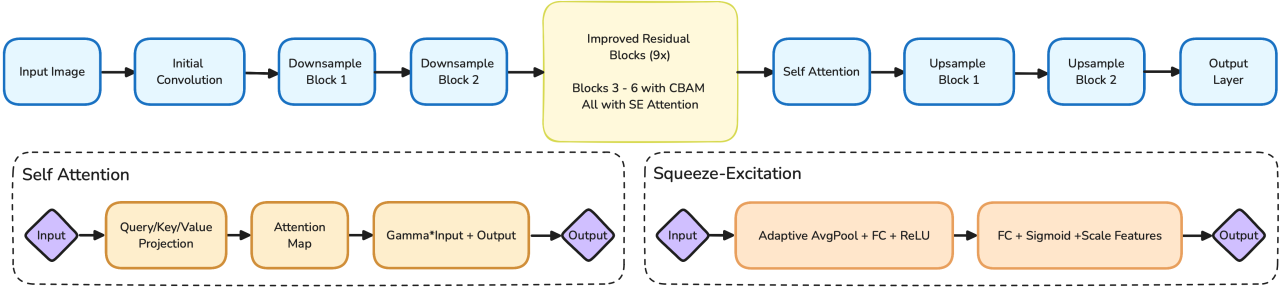
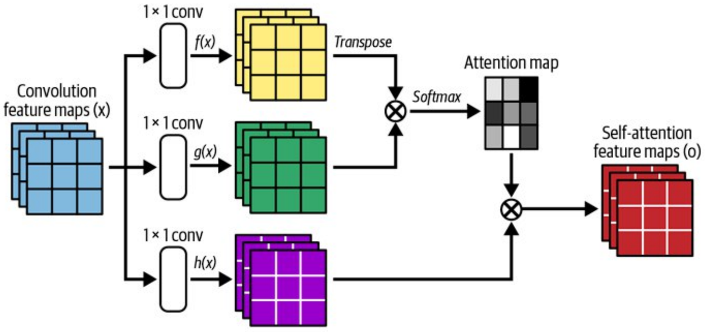
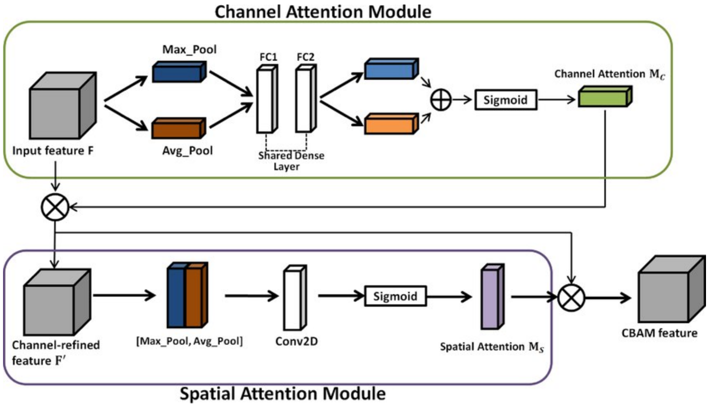


Figure 1.1: Generator Architecture



Self Attention Block



1.2.2 Discriminator

Our discriminator uses a basic sequential architecture with convolutional layers that doubles in number of channels 3 times, from 64 to 512. Each convolution output is passed to a spectral norm layer, instance normalisation and a leaky relu activation function. We then branched into a Avg Pool into a FC layer for global classification and a PatchGAN block for local classification.

Given that we have used a 256x256 image, the feature map will have size of $[B, 512, 16, 16]$, the patchGAN will have size of $[B, 1, 16, 16]$. A larger patch GAN and feature map allows us to capture more details of specific features in certain areas which allows the discriminator to provide more useful information to the generator.

1.3 Data Preparation

1.3.1 Augmentations

We started with resizing images to 128x128, with a batch size of 16 before we hit a memory limit. We also applied no augmentations to the images, as a baseline to compare against later on when we add augmentations.

After adding various minor augmentations like affines, color jitters, flips, light gaussian blurs and posterising, we found that any combination of them resulted in worse metrics overall. From visual inspection, we found that the generated raw images were more noisy than without augmentations, which could contribute to the worse metrics. Furthermore, the generated cartoon images were a lot more flat than the target images, losing a lot more of the finer details and textures. Lastly, affines and any crops were clearly visible in the generated images when they otherwise should not be.

Thus, in both cases, we believe that adding the augmentations were encouraging the model to over-process the style conversions, which likely caused the reduction in metrics. Finally our best result was achieved with no

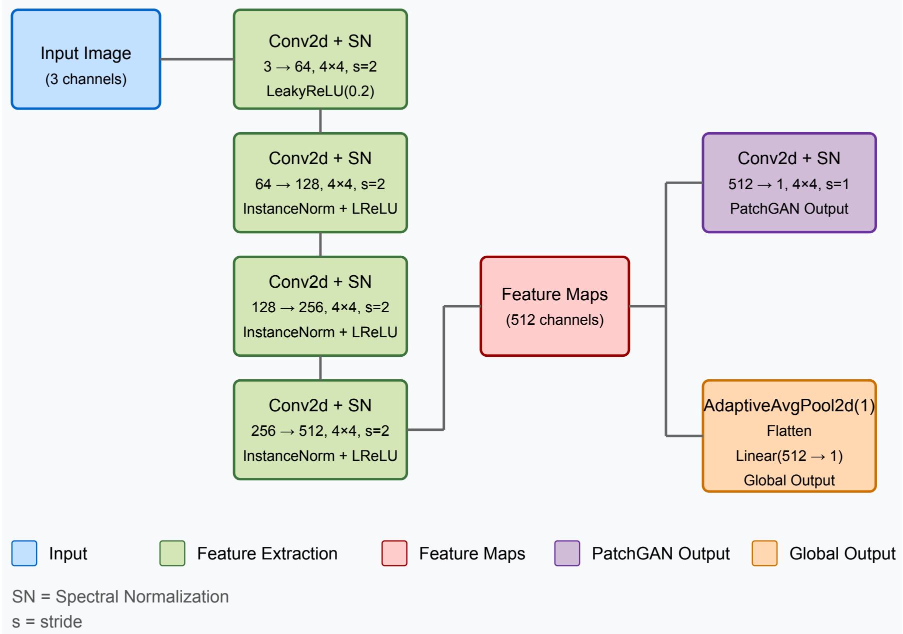


Figure 1.2: Discriminator Architecture

augmentations, and upscaling the images to 256x256 with a batch size of 3. Some sample results are shown in the below Figures.

1.3.2 Data Split

We worked with simple 99/1 split on training/validation data since we found that we needed as much data as possible for training the model to achieve our best results.

1.4 Methodology

1.4.1 Staged Training

Our best results came from a multi-staged supervised training approach. The idea is inspired from an application on Sketch-to-Manga colorization by Zhang et al. (2018)[1], where they utilised 2 generators using the same architecture, intended for them to have different focuses. Our methodology follows a "greedy" approach seen in **Algorithm 1** where we iteratively save checkpoints of our models, measure their metrics, and we keep the best performing weights. We then use these weights as a starting point for the next stage of training.

Our novelty stems from how we decrease the batch size, number of epochs, and evaluation threshold proportionally in each iterative stage, while using newly randomised seeds and maintaining learning rate so that our ratio of batch size to learning rate decreases[2]. This allows us to gradually refine the model's performance and achieve better results as each iteration encourages the model to learn finer details. Our intention with newly randomised seeds was to allow the optimisers to start on a different location in the loss optimisation graph, which may force it to hit a new local minima.

$$BatchSizes = [3, 2, 1] \quad Epochs = [100, 50, 30] \quad EvalThreshold = [10, 5, 3] \qquad TorchSeed = [1722426635407800, smthsmth$$

[1] Zhang, L., Li, C., Wong, T.-T., Ji, Y., & Liu, C. (2018). Two-stage sketch colorization. ACM Transactions on Graphics, 261–261. https://ttwong12.github.io/papers/colorize/colorize.pdf

[2] He, F., Liu, T., & Tao, D. (2019). Control Batch Size and Learning Rate to Generalize Well: Theoretical and Empirical Evidence. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d' Alché-Buc, E. Fox, & R. Garnett (Eds.), Advances in Neural Information Processing Systems (Vol. 32). Retrieved from https://proceedings.neurips.cc/paper_files/paper/2019/file/dc6a70712a252123c40d2adba6a11d84-Paper.pdf

Algorithm 1 Basic Training Loop Single Stage

```
procedure TRAINMODEL($D_A^w, D_B^w, G_{AB}^w, G_{BA}^w, batchSize, epochs, evalThreshold$)    ▷ weights for models
    $D_A \leftarrow D_A^w$
    $D_B \leftarrow D_B^w$
    $G_{AB} \leftarrow G_{AB}^w$
    $G_{BA} \leftarrow G_{BA}^w$
    $n \leftarrow 1$
    $W\_history \leftarrow [\quad]$
    while $n \neq epochs$ do    ▷ Run basic training loop
        $Train \quad D_A, D_B, G_{AB}, G_{BA} \quad with \quad Domain_A, Domain_B, batchSize$
        if $n \mod evalThreshold = 0$ then
            $W\_history.insert(D_A^w, D_B^w, G_{AB}^w, G_{BA}^w)$
        end if
        $n \leftarrow n + 1$
    end while
    $bestGMS \leftarrow inf$
    $bestW \leftarrow D_A^w, D_B^w, G_{AB}^w, G_{BA}^w$
    for $w_e \in W$ do    ▷ Evaluate each weight with test sets
        $D_A \leftarrow D_A^{w_e}$
        $D_B \leftarrow D_B^{w_e}$
        $G_{AB} \leftarrow G_{AB}^{w_e}$
        $G_{BA} \leftarrow G_{BA}^{w_e}$
        $epochGMS \leftarrow Compute \quad GMS$
        if $epochGMS < bestGMS$ then
            $bestGMS \leftarrow epochGMS$
            $bestW \leftarrow w_e$
        end if
    end for
    return $bestW$    ▷ Returns the best weights
end procedure
```

1.4.2 Image Pooling

An "Image Pool" was also used in training. This pool functions as a buffer to store previously generated images, which helps to reduce the risk of mode collapse where the Discriminator becomes too powerful, and prevents the Generator from learning anything new. The pool is updated with new generated fake images during training, and once the threshold is reached, we randomly sample from the available pool instead of using the latest generated image.

1.5 Loss Functions

1.5.1 Discriminator Losses

We used a combination of loss functions beyond the standard adversarial loss to improve the performance of our discriminator. Our overall loss function for the discriminator is as follows:

$$L_D = L_{GAN} + \lambda_{gp} L_{gp} \tag{1.1}$$

Gradient Penalty – L_{gp}

We also use a gradient penalty, inspired from Wasserstein GAN (WGAN)[3] in order to prevent the discriminator from learning too fast. Doing so avoids problems like mode collapse with vanishing gradients, and ensures that the generator is always learning, even if it is not performing well enough to "beat" the discriminator yet, thus improving stability of training.

Relativistic – L_{GAN}

We extend the typical adversarial loss by using a relativistic loss, which compares the logits of discriminating the real and fake images in a relative manner on a global scope with Binary Cross Entropy Loss. Furthermore,

[3] Arjovsky, M., Chintala, S., & Bottou, L. (2017, July 17). Wasserstein Generative Adversarial networks. PMLR. https://proceedings.mlr.press/v70/arjovsky17a.html

we took inspiration from PatchGAN[4] which whether the images is real or fake based on smaller patches, a local scope. Thus, we have another source of information for determining real or fake, which is computed with MSE loss. The equation for our relativistic loss for our "real" side is as follows, and the opposite will be applied for the "fake" side i.e. $D_{real} <=> D_{fake}, True <=> False$:

$$L_{GAN} = 0.7 \cdot MSE(D_{real} - \bar{D}_{fake}, True)_{patch} + 0.3 \cdot BCE(D_{real} - \bar{D}_{fake}, True)_{global} \tag{1.2}$$

1.5.2 Generator Losses

The generator also makes use of the **Relativistic** loss, which is an extension of the standard adversarial GAN loss. Our overall loss function for the generator is as follows:

$$L_G = L_{GAN} + \lambda_{cyc} L_{cyc} + \lambda_{id} L_{id} + \lambda_{feat} L_{feat} \tag{1.3}$$

Cycle Consistency – L_{cyc}

As we would like the fake images to be able revertible to its original image i.e. the pixels are identical, we maintain using L1 Loss for our Cycle Consistency.

Identity – L_{id}

Similarly, we also use L1 Loss for our Identity loss since images in the same domain should not be altered.

Feature – L_{feat}

Another extension that we incorporated is the Feature Loss, which serves as a perceptual loss that compares the features of the generated image with the target image. Although traditionally used with pre-trained networks, we used the features extracted from the training discriminator instead. These features are compared using L1 loss, which helps to ensure that the generated images are perceptually similar to the target images.

[4] Isola, P., Zhu, J., Zhou, T., & Efros, A. A. (2016). Image-to-Image Translation with Conditional Adversarial Networks. arXiv (Cornell University). https://doi.org/10.48550/arxiv.1611.07004

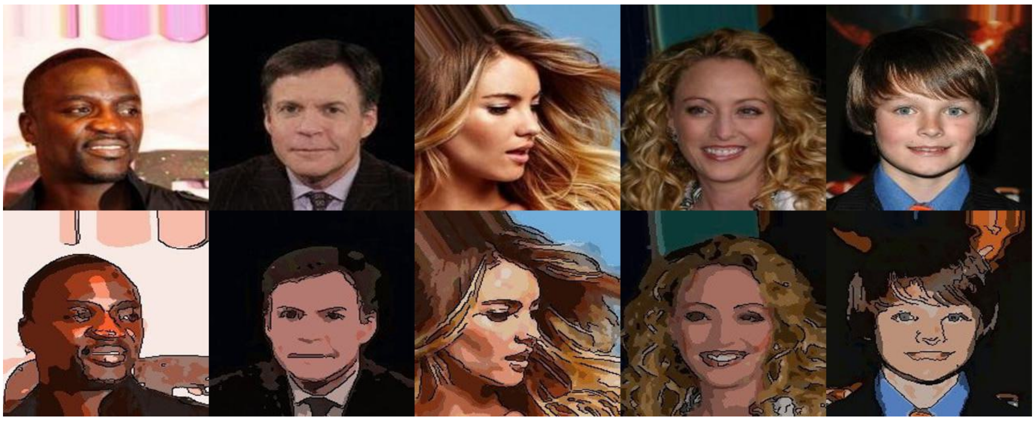


Figure 1.3: Original (resized 256x256). Top row: Raw, Bottom row: Cartoon

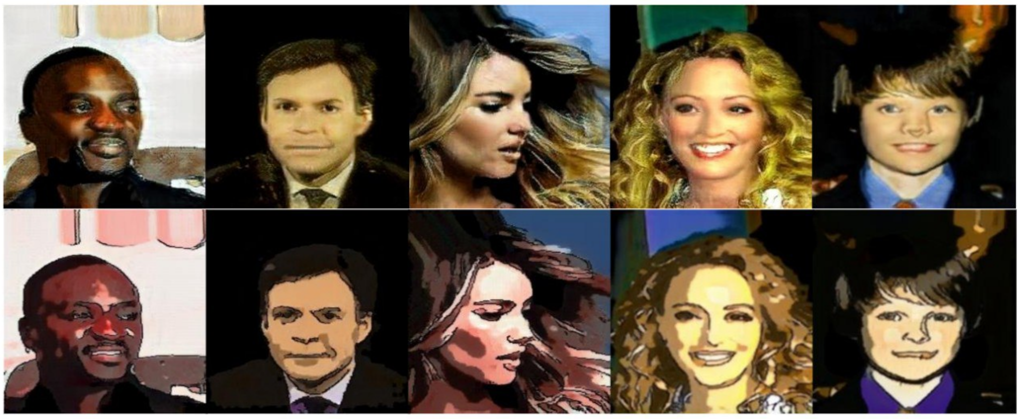


Figure 1.4: 128x128 Avg GMS: 4.47829

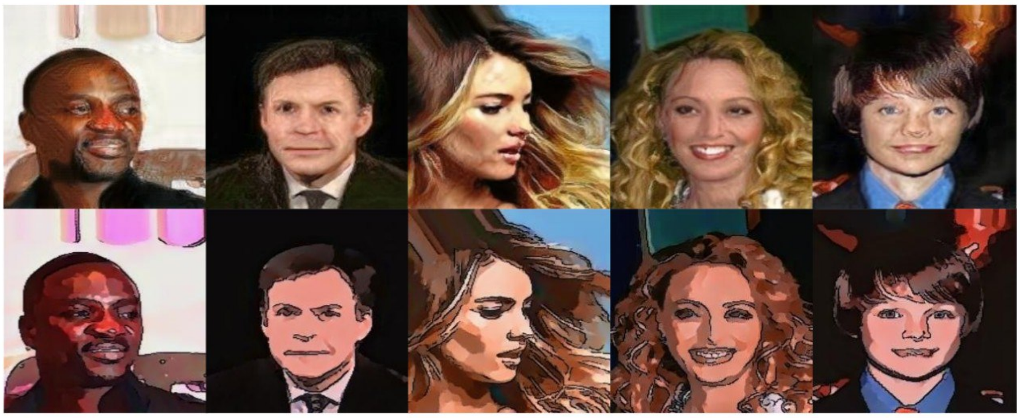


Figure 1.5: 256x256, Stage 1: Batch Size 3, Avg GMS: 3.72523

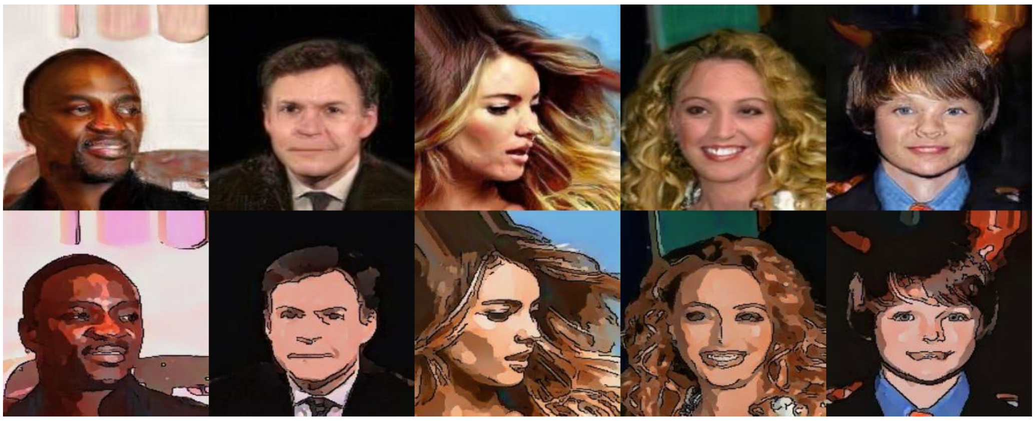


Figure 1.6: 256x256, Stage 2: Batch Size 2, Avg GMS: 3.42415

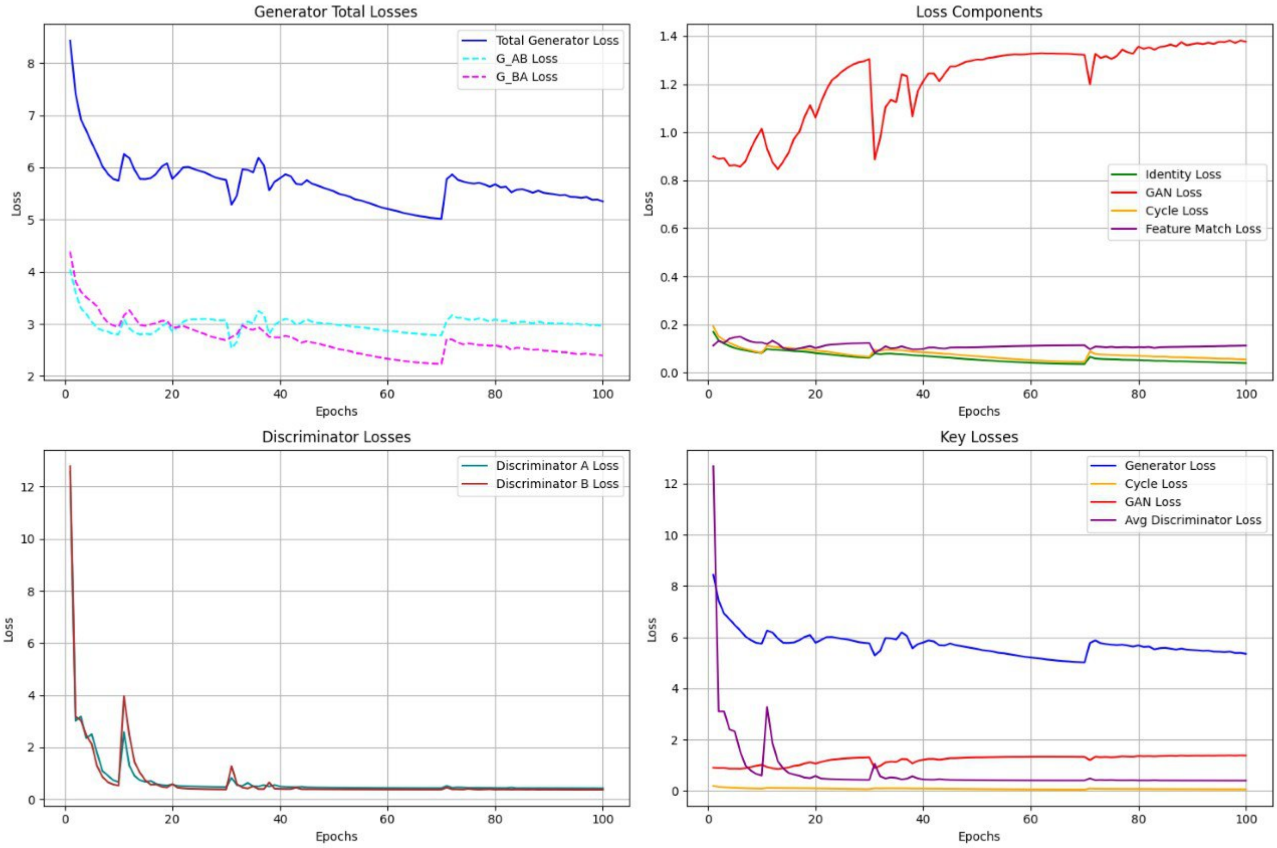


Figure 1.7: Overall loss curves

2. Task 2

2.1 Introduction

2.1.1 Configuration

2.2 Architecture

2.2.1 Generator

We used a similar architecture as the one used in Task 1 (1.2.1), with the following modification specific to catering for vast difference in domains.

Vector Quantization Block is used to convert the input image into a discrete representation, which is then passed through a series of residual blocks. The output of the residual blocks is then passed through a decoder to reconstruct the image. The vector quantization block is used to learn a set of discrete codes that can be used to represent the input image, allowing for more efficient encoding and decoding.

which will learn additional features, as animal and pokemon do not have a direct translation.

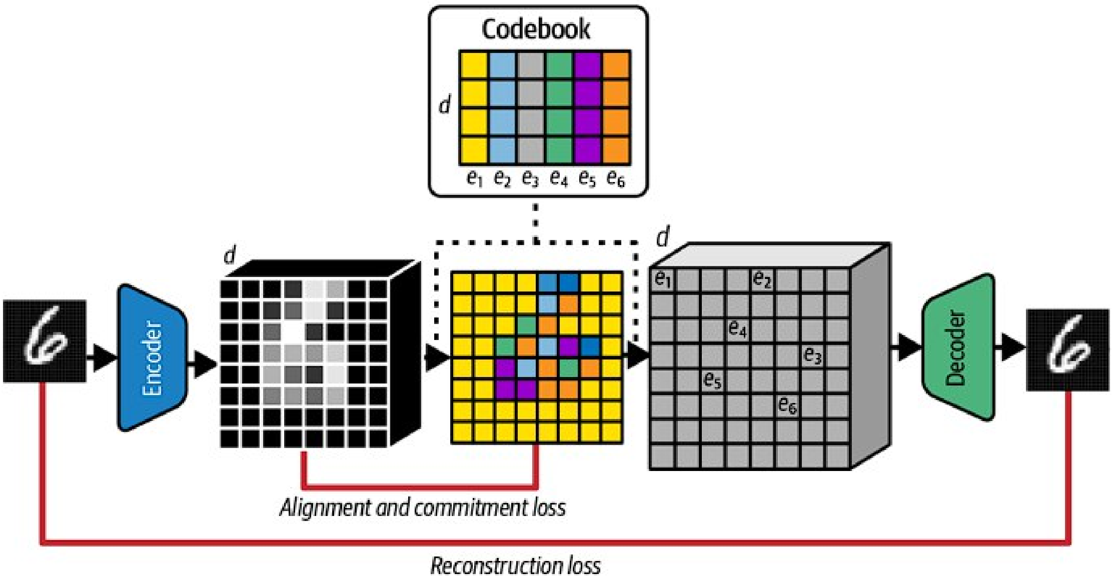


Figure 2.1: Vector Quantization Block

2.2.2 Discriminator

We used the same general architecture as the one used in Task 1 (1.2.2), with a few modifications specific to catering for the smaller images.

Since our Pokémon sprite sizes are only 96x96, we removed the last downsampling layer of the discriminator so bottleneck/features will be capped at $[B, 256, 12, 12]$. The reasons are two-fold:

- The structure of an animal and pokemon being very different, so we will not want to make the discriminator too powerful.
- A $[B, 256, 6, 6]$ feature map will not provide enough useful information to the generator either, thus a $[B, 256, 12, 12]$ feature map can better help the generator to identify the Pokémon sprite style.

2.3 Data Preparation

2.3.1 Collection

Due to our task of transfering the styles from a real-life animal to that of a Pokémon sprite, we were unable to find a paired dataset between these 2 domains. Thus, we resorted to utilising 2 individual datasets: one for the animal images, another for the Pokémon sprites, and then combining them together.

Animal Dataset

The animal dataset [1] we used consists of 5400 images of animals from 90 different classes / types of animals. We decided to use all images from the animal dataset. (Usability: 10.0 Total Image: 5.4k)

Pokémon Sprite Dataset

We decided to use Pokémon sprites as comapared to the original Pokémon art as we there is a distinct style to how Pokémon sprites are. Furthermore, using the original Pokémon art would not be suitable for a "style transfer" as we felt that the original art is at the prerogative of the individual artists when they created them. Thus, the styles across Pokémon will not be consistent, and our model will find it very difficult to learn the mapping between the two domains. Our failed attempts can be found in the Appendix 2.5

The Pokémon sprite dataset [2] we used consists of many different variations of the same Pokémon, namely those front-facing, back facing, shiny and normal. We have decided to use the normal and front-facing versions as those are the most suitable given that our animal datasets also exhibit the same characteristics. (Usability 5.29, Total Image: 10.2k)

2.3.2 Pre-Processing

Due to the domains being vastly different, and the images are of different formats, we had to pre-process the images before feeding them into the model so that it will learn what it needs to learn.

Background Removal

The Pokémon sprites from the dataset are placed on a white background, whereas the animal images are not. Thus, we need to remove the background of the animal images so that the model can focus on the animals themselves rather than including how to convert the background to white.

To do so, we used a pretrained model (VGG16) to remove the background of the animal and convert them into white background found in `clean_background.py`.

No Tightly Paired Images

We **randomly** paired each animal to a **random** Pokémon and we ensure that each Pokémon is not used too many times, as the different Pokémons are not equally represented in quantity. Since we have 1025 unique Pokémon, with different quantity of front-facing Pokémons, and 5400 different animal images, we try to ensure that each type of Pokémon sprite is at most represented 6 times. Finally, we have a total of 5400 animal-Pokémon paired images.

Failure From Tight Pairs Before choosing to adopt a more "unsupervised learning" approach, we tried to mimic a "supervised learning" by mapping the animals to a specific Pokémon based on how similar their general designs are. However, we found that the results were **not satisfactory**. Although it does look like a Pokémon sprite from afar, it has a much worse GMS score of 6.5 as compared to the non-tightly paired dataset of 5.

2.3.3 Training/Validation

A simple 90/10 train-test split. So, we will have 4860 train images and 540 test images. The train (4860 images) is split further to a 99/1 split, where 4811 images will be used for training, while 49 will be used for validation (visualisation of how the pokemon will look like)

[1] Animal Image Dataset (90 Different Animals) – Zooming in on Wildlife: 5400 Animal Images Across 90 Diverse Classes https://www.kaggle.com/datasets/iamsouravbanerjee/animal-image-dataset-90-different-animals

[2] Pokemon sprite images – Total 10,437 Pokemon sprite images in 96x96 resolution. https://www.kaggle.com/datasets/yehongjiang/pokemon-sprites-images

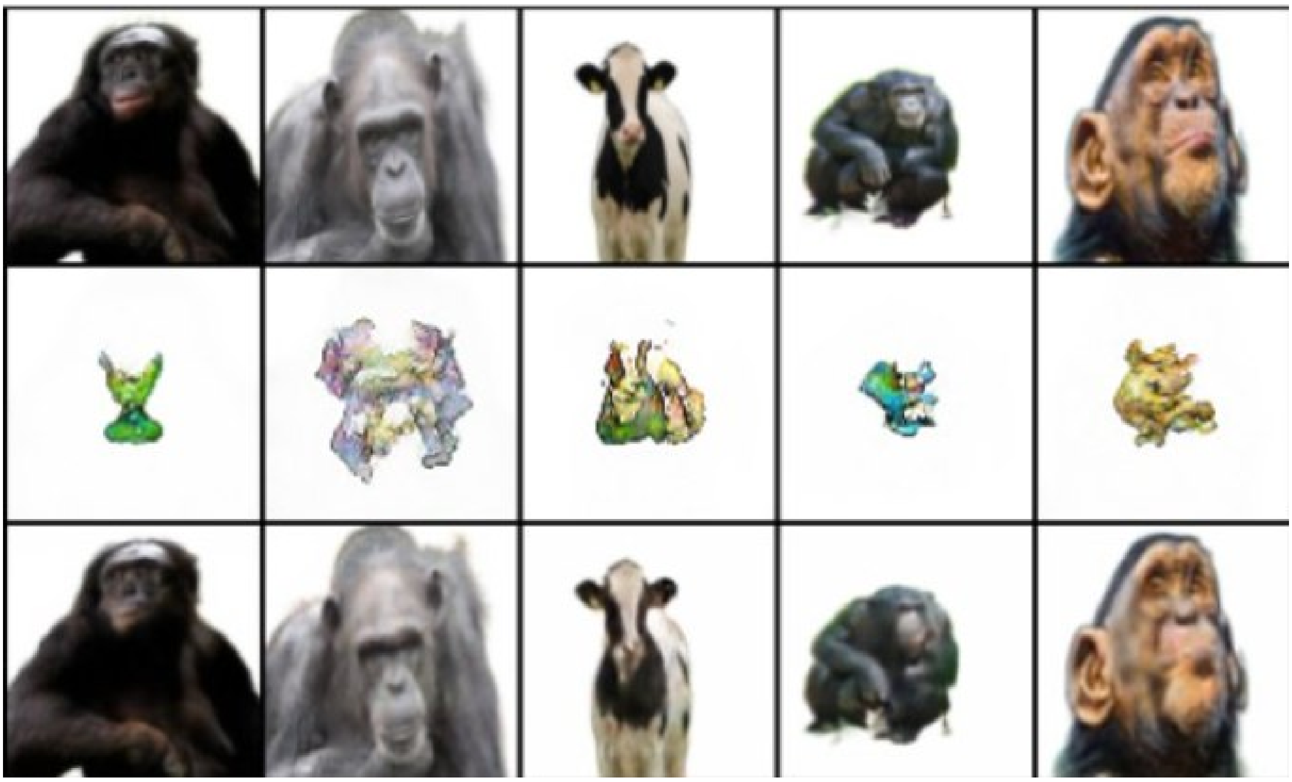


Figure 2.2: Unsatisfactory results from tightly paired dataset

2.4 Loss Functions

Our overall loss functions for CycleGAN are as follows:

$$L_{total} = L_{GAN} + \lambda_{cyc} L_{cyc} + \lambda_{id} L_{id} + \lambda_{edge} L_{edge} + \lambda_{color} L_{color} \tag{2.1}$$

2.4.1 Discriminator

Patch

Least Squares

2.4.2 Generator

Adversarial

Use basic

Cycle Consistency

Identity

Edge Consistency

Color Consistency

2.4.3 Adaptive Loss Weighting

2.5 Appendix

Pokémon Official Art

Other dataset tried https://www.kaggle.com/datasets/hesselaar/all-pokemon-official-images This data consisted of 1010 official pokemon art. However, after training, the results seems to be really bad.

Notice that in 2.3 the generated image does not resemble Pokémon art at all. We suspect that it is trying to colour the animal images in but it does now know exactly in what colour to use. We extensively researched on the reasons why and got cooked.

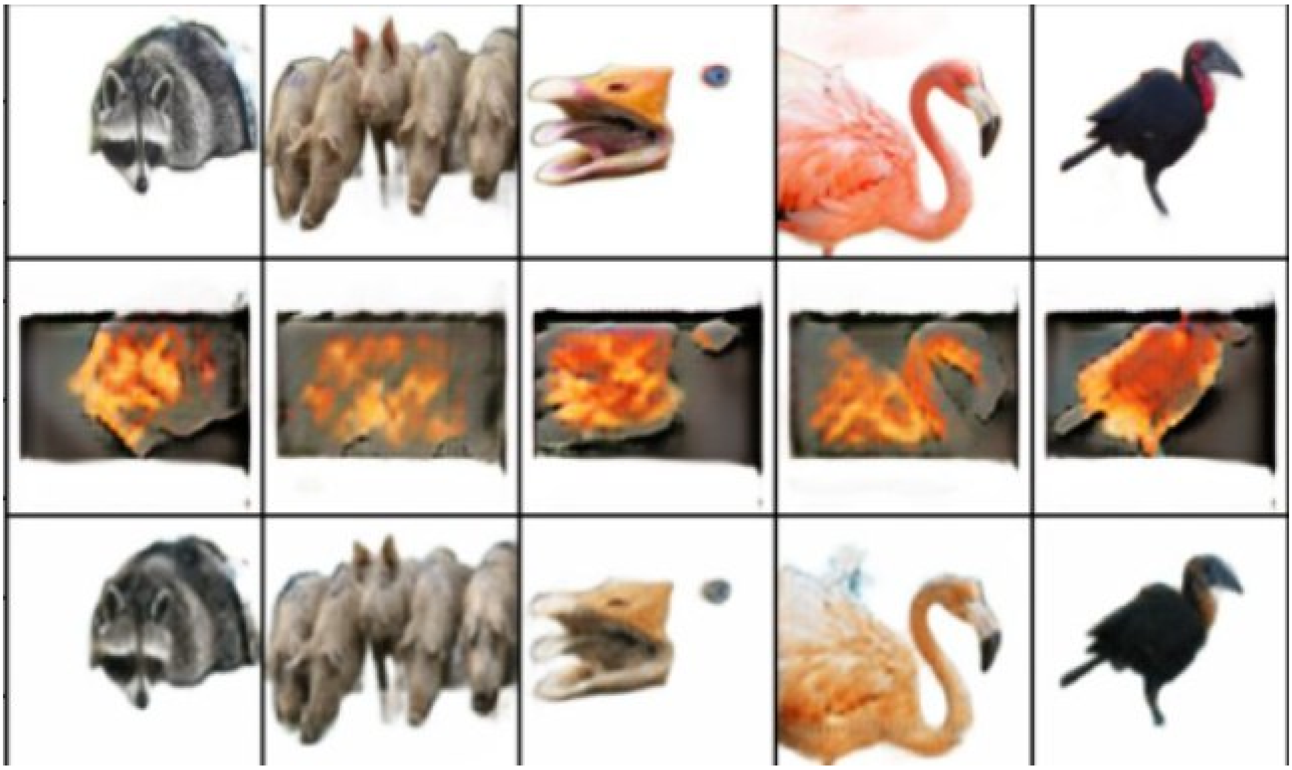


Figure 2.3: Random sample 1