

Image2Painting Project Report

Srikrishna Dantu

19MCME02

Shobhit Kumar

19MCME16

KPSSS Srinu

19MCME26



Original Image



Translated Van Gogh Painting

Table of Contents

Introduction	2
Architecture Explored	2
Encoder-Decoder Network	2
Neural Style Transfer	2
CycleGAN	3
CycleGAN Network Architecture	4
Generator	4
Discriminator	5
Datasets	7
Loss Functions	7
Adversarial Loss	7
Cycle Consistency Loss	8
Identity Loss	9
Total Loss	10
Training	11
Results	17
Conclusion & Future Work	22
References	23

Introduction

This project belongs to the domain where we need to generate an image (painting) given another image (captured photo) (image-to-image translation). Many architectures are available to do this work; we have listed a few of the architectures which can be useful for our project.

We chose CycleGAN as the architecture for our project. We described the reasons to chose this architecture among several other architectures in detail in the below context.

Reference to the original CycleGAN paper [\[2\]](#) for details.

Architecture Explored

Architecture that we can use:

Encoder-Decoder Network

We need to have pair of input and output pictures where the input will be a photo taken from the camera, and the output will be painting in that specific style what is in the photo.

Getting a bunch of photos is easy, but we cannot have paintings of those exact styles in photos. Vice-versa is also not possible; we can have a bunch of paintings, but getting photos of the entity in a similar environment is not easy.

So the encoder-decoder network is not best suited for our project.

Neural Style Transfer

Neural style transfer is another famous way to perform image-to-image translation, which synthesizes an image by combining the content of one image with the style of another image based on matching the *Gram matrix statistics* of pre-trained deep features (VGG-19). But it performs the translation between two specific images. What we need is to focus on the mapping between two image collections and capture correspondences between the higher-level appearance structure (color, texture, etc.).

Refer to the original paper[\[1\]](#) for details.

CycleGAN

Since, for us, getting paired training data is not possible. Cycle-consistent adversarial networks (CycleGAN)[\[2\]](#) translate an image from a source domain X to a target domain Y in the absence of paired training examples. The architecture assumes that there is some underlying relationship between both domains.

Although we lack supervision in the form of paired examples, the architecture exploits supervision at the level of sets: given one set of images in the domain X and a different set of images in the domain Y . We train a mapping $G : X \rightarrow Y$ such that the output $\hat{y} = G(x)$, $x \in X$, is indistinguishable from images $y \in Y$ by an adversary trained to classify \hat{y} apart from y .

Why is CycleGAN cyclic in architecture?

In practice, it is found that it is difficult to optimize the adversarial objective in isolation: standard procedures often lead to the well-known problem of *mode collapse*, where all input images map to the same output image, and the optimization fails to make progress.

Therefore, the architecture exploits the property that translation should be “*cycle consistent*”. Mathematically, if we have a translator $G : X \rightarrow Y$ and another translator $F : Y \rightarrow X$, then G and F , should be inverse of each other, and both mapping should be bijections.

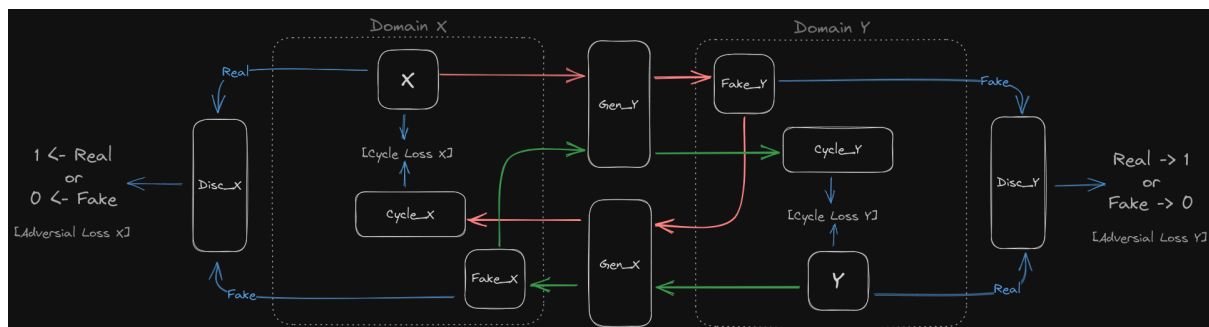
The architecture applies this structural assumption by training both mapping G and F simultaneously and adding a “*cycle consistency loss*” that encourages $F(G(x)) \approx x$ and $G(F(y)) \approx y$. Combining this loss with “*adversarial losses*” on domain X and Y yields the full objective for unpaired image-to-image translation.

[“Adversial Loss” & “Cycle Consistency Loss” to be discussed later in the report]

Refer [3] & [4] for easy explanation on CycleGAN.

CycleGAN Network Architecture

The architecture has two generators (Gen_X and Gen_Y) and their corresponding discriminator ($Disc_X$ and $Disc_Y$).



CycleGAN Network Architecture

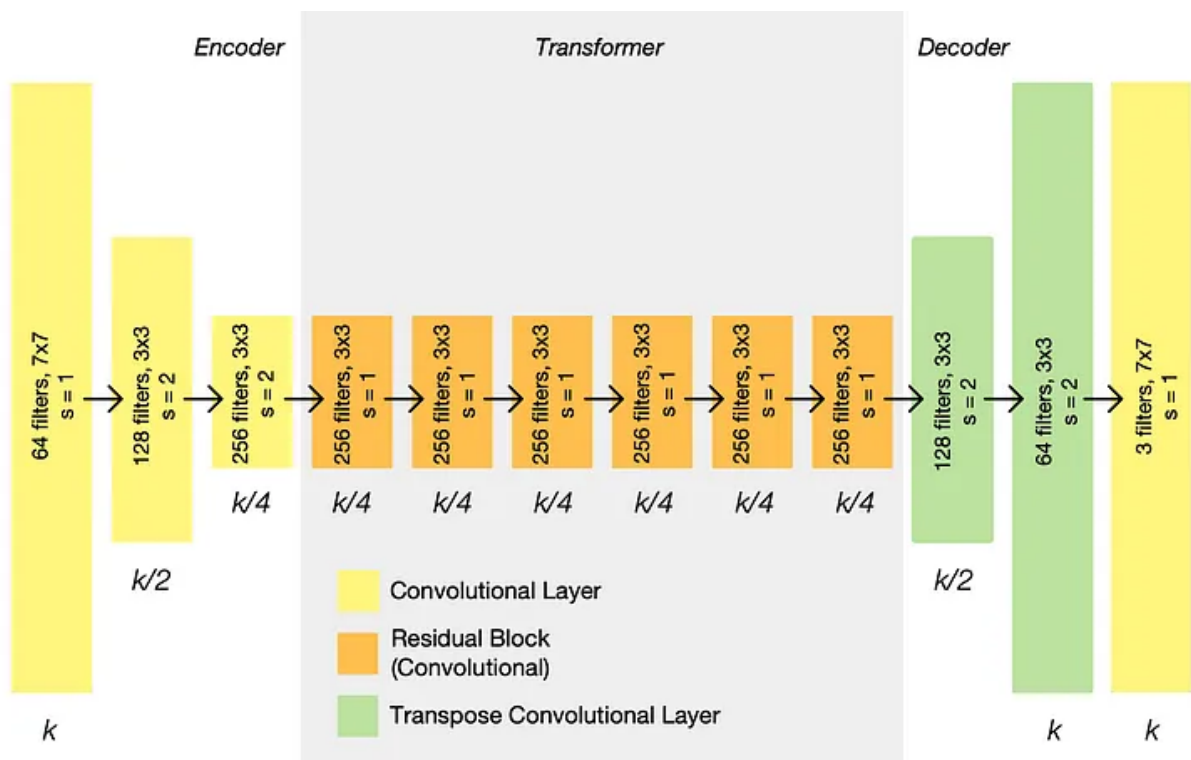
As seen in the above figure,

- We have two different domains of images (X & Y), with the image (x & y) as true images.
- Gen_Y and Gen_X are two *generators* that translate the image from the domain X to Y and the image from the domain Y to X simultaneously.
- $Disc_X$ and $Disc_Y$ are two *discriminators* that will discriminate true x with fake generated $Fake_X$ or ($Gen_X(y)$) and true y with fake generated $Fake_Y$ or ($Gen_Y(x)$) simultaneously and will output binary (1 or 0) for real or fake.

Generator

- 6 residual blocks for 128x128 training image
- 9 residual blocks for 256x256 or higher resolution training image
- C7s1-k \rightarrow 7x7 Convolution-InstanceNorm-ReLU layer with k-filters and stride 1

- $dk \rightarrow$ 3x3 Convolution-InstanceNorm-ReLU layer with k -filters and stride 2
- $uk \rightarrow$ 3x3 Convolution-InstanceNorm-ReLU layer with k -filters and stride $\frac{1}{2}$
- $Rk \rightarrow$ Residual block contains 3x3 Convolution layer with same no. of filters on both layers
- Network with 6 residual blocks
 - C7s1-32, d64, d128, 6*(R128), u64, u32, C7s1-3
- Network with 9 residual blocks
 - C7s1-32, d64, d128, 9*(R128), u64, u32, C7s1-3

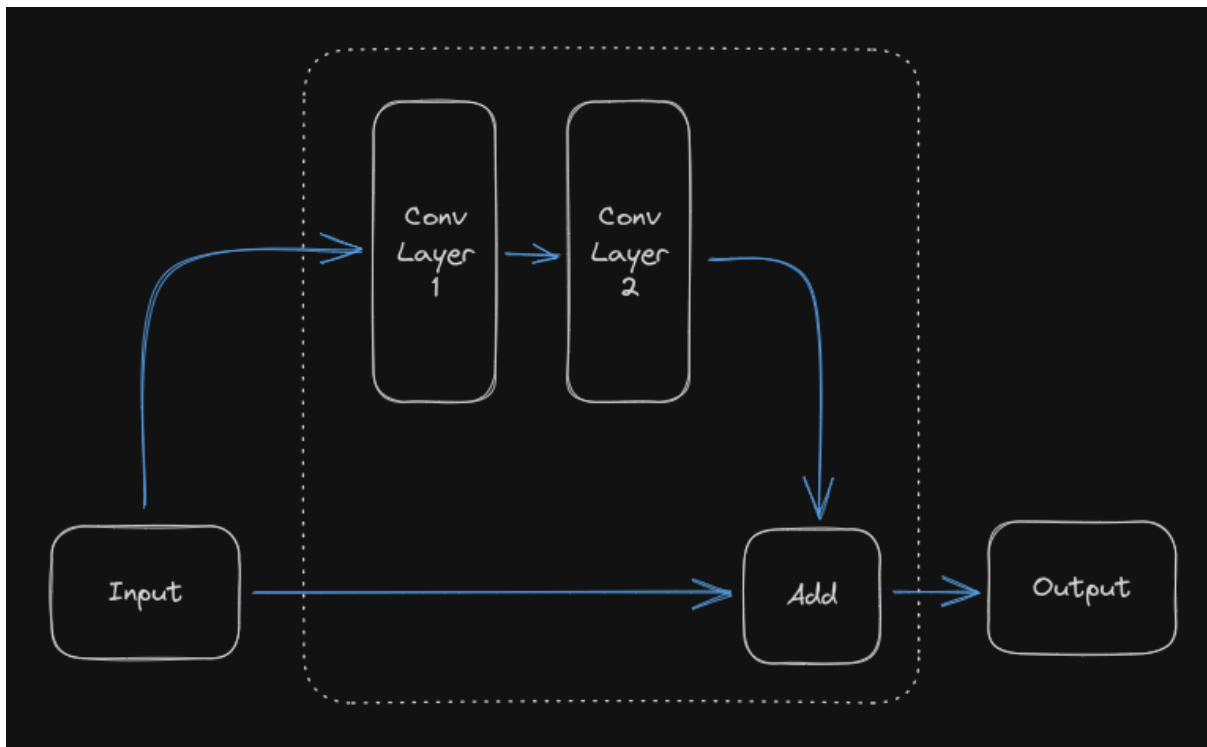


Generator Architecture for 6 ResNet Block | Image Credits: [Sarah Wolf - Toward Data Science](#)

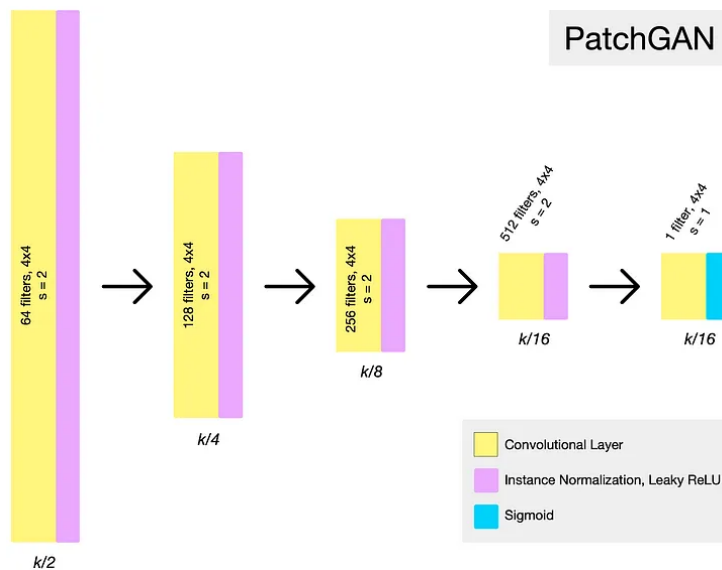
Discriminator

- 70x70 PatchGAN
- $Ck \rightarrow$ 4x4 Convolution-InstanceNorm-ReLU layer with k -filters and stride 2
- Discriminator Network

- C64, C128, C256, C512
- After the last layer, apply a Convolution to produce 1-dimensional output
- Do not use InstanceNorm for the first C64 layer
- Use Leaky-ReLU with slope 0.2



Residual Block Architecture in Generator



Discriminator Architecture | Image Credits: [Sarah Wolf – Toward Data Science](#)

Datasets

We are following two datasets (X & Y domains); both datasets are taken from Kaggle, where X is the input captured image[5] and Y is the output painting[6].

We will work with Vincent Van Gogh paintings and aim to generate a Van Gogh version of any input-captured image.

Loss Functions

There are primarily three types of loss functions involved in the CycleGAN architecture, as discussed in the original paper.

1. Adversarial Loss
2. Cycle Consistency Loss
3. Identity Loss

Let's discuss these loss functions in detail.

Adversarial Loss

Adversarial losses are applied to both mapping functions. For the generator Gen_Y and its discriminator $Disc_Y$, and the generator Gen_X and its discriminator $Disc_X$.

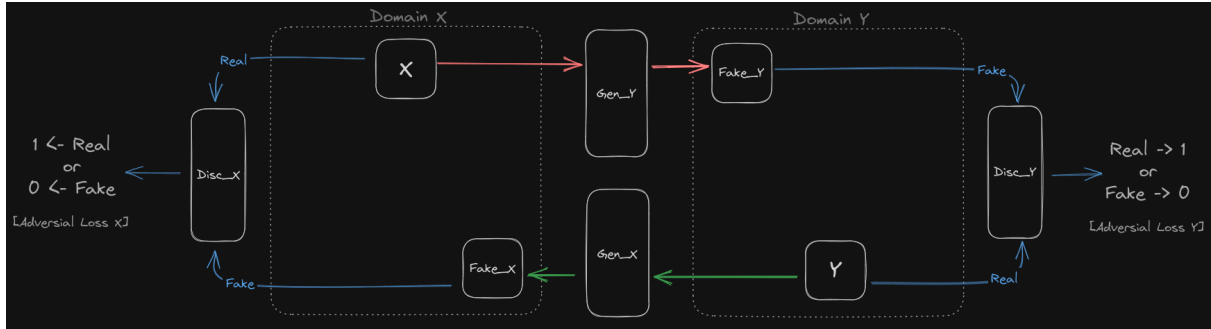
Here for Gen_Y & $Disc_Y$, Gen_Y tries to generate an image $Fake_Y$ from X that looks similar to images from the domain Y , while $Disc_Y$ aims to distinguish between translated image $Fake_Y$ and real image from domain Y .

The adversarial loss for Gen_Y & $Disc_Y$ is given as L_{GAN-Y} , where, Gen_Y aims to minimize $E_{x \sim X}[(Disc_Y(Fake_Y) - 1)^2]$ against $Disc_Y$ which aims to minimize $E_{y \sim Y}[(Disc_Y(y) - 1)^2] + E_{x \sim X}[(Disc_Y(Fake_Y) - 0)^2]$.

Similarly for Gen_X & $Disc_X$, Gen_X tries to generate an image $Fake_X$ from Y that looks similar to images from the domain X , while $Disc_X$ aims to distinguish between translated image $Fake_X$ and real image from domain X .

The adversarial loss for Gen_X & $Disc_X$ is given as L_{GAN-X} , where, Gen_X aims to minimize $E_{y \sim Y}[(Disc_X(Fake_X) - 1)^2]$ against $Disc_X$ which aims to minimize $E_{x \sim X}[(Disc_X(x) - 1)^2] + E_{y \sim Y}[(Disc_X(Fake_X) - 0)^2]$.

How adversarial loss works in CycleGAN is shown in the figure below.



Adversarial Loss in CycleGAN Architecture

Cycle Consistency Loss

As discussed earlier, the adversarial loss is not enough to guarantee that the learned generator can map the input X to desired output Y as it may face a problem of *mode collapse*, where all input images map to the same output image, and the optimization fails to make progress.

Thus the author argues that the learned mapping function should be cycle consistent, where for each X , the image translation cycle should be able to bring X back to the original image, i.e.

$$X \rightarrow Gen_Y(X) \rightarrow Fake_Y \rightarrow Gen_X(Fake_Y) \rightarrow Cycle_X \approx X$$

The cycle consistency loss for X & $Cycle_X$ is given as

$$L_{Cycle-X} = E_{x \sim X}[||Cycle_X - X||_1]$$

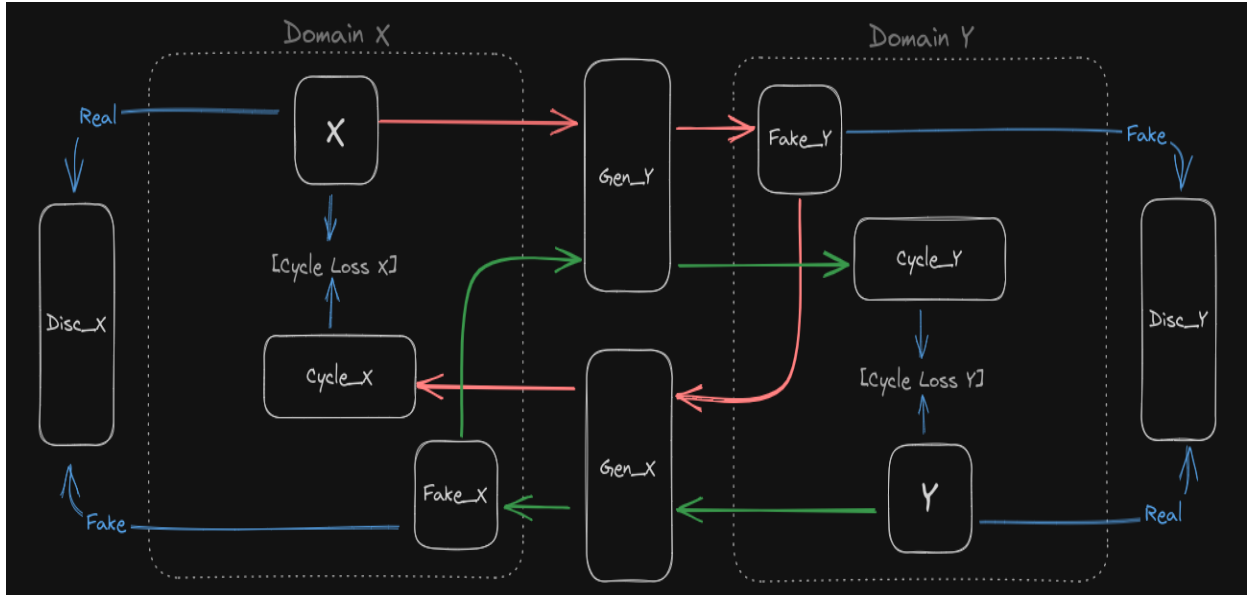
Similarly for each Y , the image translation cycle should be able to bring Y back to the original image, i.e.

$$Y \rightarrow Gen_X(Y) \rightarrow Fake_X \rightarrow Gen_Y(Fake_X) \rightarrow Cycle_Y \approx Y$$

The cycle consistency loss for Y & $Cycle_Y$ is given as

$$L_{Cycle-Y} = E_{y \sim Y} [||Cycle_Y - Y||_1]$$

How cycle consistency loss works in CycleGAN is shown in the figure below.



Cycle Consistency Loss in CycleGAN Architecture

Identity Loss

The author introduced an additional loss to encourage the mapping to preserve color composition between the input and output. We regularize the generator to be near identity mapping when real samples of the target domain are provided as the input to the generator.

Let's say, Y from the output domain is provided as the input to Gen_Y in place of X , which should generate an identity mapping of Y called $Identity_Y$.

The identity loss for Y & $Identity_Y$ is given as

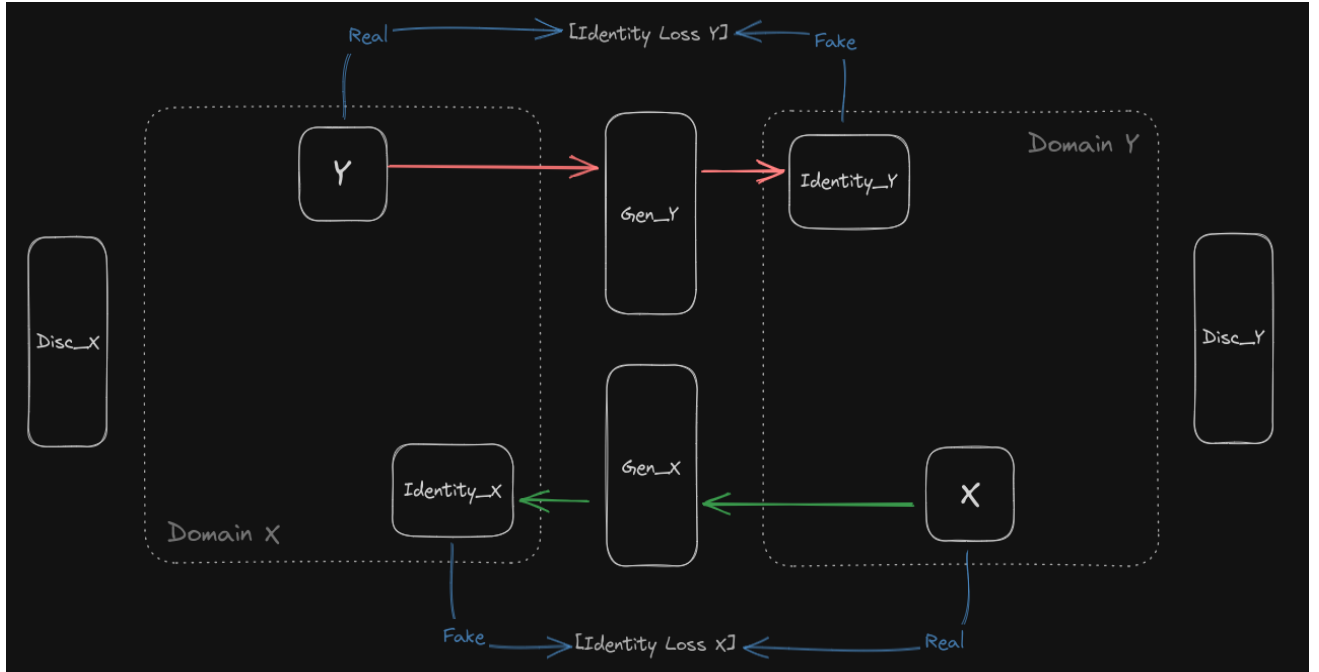
$$L_{Identity-Y} = E_{y \sim Y} [||Identity_Y - Y||_1]$$

Similarly X from the input domain is provided as the input to Gen_X in place of Y , which should generate an identity mapping of X called $Identity_X$.

The identity loss for X & $Identity_X$ is given as

$$L_{Identity-X} = E_{x \sim X} [||Identity_X - X||_1]$$

How identity loss works in CycleGAN is shown in the figure below.



Identity Loss in CycleGAN Architecture

Total Loss

The total loss for CycleGAN architecture is the combination of all losses, i.e., adversarial loss, cycle consistency loss, and identity loss, which is given as

$$\begin{aligned} L_{CycleGAN} = & L_{GAN-X} + L_{GAN-Y} \\ & + (L_{Cycle-X} + L_{Cycle-Y}) * \lambda_{Cycle} \\ & + (L_{Identity-X} + L_{Identity-Y}) * \lambda_{Identity} \end{aligned}$$

Where, λ_{Cycle} & $\lambda_{Identity}$ are two parameters which control the relative importance of cycle consistency loss and identity loss on the overall loss function.

Training

Before going on with training, initial configurations need to be pre-configured, where we use these configuration variables in later stages of training.

Details of variables used in training:

Name	Details
DEVICE	Set to "cuda" if any GPU devices are provided as hardware units for computing the training; else we set it to "cpu". Cuda enables the special parallel computations used in Pytorch.
TRAIN_DIR_X	Training directory path of X domain.
TRAIN_DIR_Y	Training directory path of Y domain.
BATCH_SIZE	<i>Batch size</i> ; set to 1 for batch normalization.
LEARNING_RATE	<i>Learning rate</i> is the η which is used in backpropagation. $\eta = 0.0002$
NUM_WORKERS	To increase the number of processes.
NUM_EPOCHS	Number of epochs.
LOAD_MODEL	Boolean variable; whether to load the model from the previous checkpoint or not.
SAVE_MODEL	Boolean variable; to save the checkpoint of all the Generators and Discriminators.
CHECKPOINT_GEN_	To save the generator checkpoints.
CHECKPOINT_CRITIC_	To save the critic or discriminator checkpoints.
LOAD_CHECKPOINT_	To load the previously saved checkpoints.

LAMBDA_CYCLE	Lambda value λ_{Cycle} introduced to include the cycle consistency loss into the total loss.
LAMBDA_IDENTITY	Lambda value $\lambda_{Identity}$ introduced to include the identity loss into the total loss.

save_checkpoint(model, optimizer, filename):

- To save the checkpoints of the trained networks (Both Generators Gen_X and $Gen_{Y'}$ and Discriminators $Disc_X$ and $Disc_Y$).
- It takes the model as input to save its network weights and other contents in a tar file given by the file name as a parameter.

load_checkpoint(checkpoint_file, model, filename):

- To load the saved checkpoints of the trained networks (Both Generators Gen_X and $Gen_{Y'}$ and Discriminators $Disc_X$ and $Disc_Y$) which are stored in the tar files given in as file name parameters.
- It takes the model and optimizers as inputs to load its network weights and all its contents.

XYDataset Class:

- To take the input examples from both X and Y domains with proper indexing and root file path of their respective directories.
- Getitem method does this part of the input loading and preprocessing (augmentations provided as in order).
- Data Augmentation methods used:
 - Resizing the input images to 256x256
 - Horizontal flipping with p-value set to 0.5
 - Normalization.

Main Function:

- Initialize the generators of X and Y as gen_X (generates X domain fake images) and gen_Y (generates Y domain fake images).

- Initialize the discriminators of X and Y as `disc_X` (discriminates X domain whether fake or real), and `disc_Y` (discriminates Y domain whether fake or real).
- Optimizers for both the generators and discriminators, we use *Adam optimizer* with β_1 and β_2 with 0.05 and 0.999 as their respective values. We also provide the learning rate as set earlier to 0.0002.
- For cycle consistency loss and identity loss, we use L1 loss, and For adversarial loss, we use mean squared error.
- Checking conditions for loading the models from the saved checkpoints where the trained generator and discriminator were saved.
- Dataset initialized to load the training data into the variable using the `XYDataset` class.
- Initializing the dataset loader variable using the built-in `DataLoader` of PyTorch library with the below-given parameters. It helps to load and iterate over training examples.

```
dataset          # To load the dataset
batch_size = BATCH_SIZE # Batches of input
shuffle = True      # To shuffle the dataset
num_workers = NUM_WORKERS # To allow multiprocessing
pin_memory = True    # To load the dataset as cuda
                    tensors
```

- Initializing the scalers to train the weights with float operations, with `float16`, `g_scaler`, and `d_scaler` for the generators and discriminator.
- Iterate over all the epochs for the training.
 - Train function to do the training for the generators and discriminators with the loss functions.
 - To save the checkpoints of the trained network models.

Training Function:

Takes input loader, networks initialized as `gen_X`, `gen_Y`, `disc_X`, and `disc_Y`, optimizers, loss variables, and scalers.

Iterate over all the loaded examples in the loader variable.

Discriminator Training:

- Generate a fake "x" domain image from the gen_X.
- Let the disc_X discriminate a real "x" domain image and get the mean output as "D_X_real".
- Let the disc_X discriminate the "fake_x" image, mean value as output in "D_X_fake".
- The error is calculated as:

$Disc_X$ which aims to minimize D_X_loss

$$E_{x \sim X}[(Disc_X(x) - 1)^2] + E_{x \sim X}[(Disc_X(Fake_X) - 0)^2]$$

`D_X_real_loss = mse(D_X_real, torch.ones_like(D_X_real))`

`D_X_fake_loss = mse(D_X_fake, torch.zeros_like(D_X_fake))`

`D_X_loss = D_X_real_loss + D_X_fake_loss`

- Let the disc_Y discriminate a real "y" domain image, and get the mean output as "D_Y_real".
- Let the disc_Y discriminate the "fake_y" image, mean value as output in "D_Y_fake".
- The error is calculated as:

$Disc_Y$ which aims to minimize D_Y_loss

$$E_{y \sim Y}[(Disc_Y(y) - 1)^2] + E_{y \sim Y}[(Disc_Y(Fake_Y) - 0)^2]$$

`D_Y_real_loss = mse(D_Y_real, torch.ones_like(D_Y_real))`

`D_Y_fake_loss = mse(D_Y_fake, torch.zeros_like(D_Y_fake))`

`D_Y_loss = D_Y_real_loss + D_Y_fake_loss`

- Total loss is calculated as:

$$D_loss = (D_X_loss + D_Y_loss) / 2$$

- Backpropagation is implemented based on the D_loss value with the "adam" optimizer.

Generator Training:

- The generated fake images are both "fake_x" and "fake_y" by the generators gen_X and gen_Y, respectively.

- Get the discriminant mean value from the discriminator as D_X_fake and D_Y_fake from disc_X and disc_Y for the inputs fake_x and fake_y, respectively.

- Calculate the adversarial losses of the generator gen_X.

The error L_{GAN-X} is calculated as:

$$L_{GAN-X} = E_{x \sim X} [(Disc_X(Fake_X) - 1)^2]$$

```
loss_G_X = mse(D_X_fake, torch.ones_like(D_X_fake))
```

- Calculate the adversarial losses of the generator gen_Y.

The error L_{GAN-Y} is calculated as:

$$L_{GAN-Y} = E_{y \sim Y} [(Disc_Y(Fake_Y) - 1)^2]$$

```
loss_G_Y = mse(D_Y_fake, torch.ones_like(D_Y_fake))
```

- Calculate the cycle consistency loss for gen_X as:

$$L_{Cycle-X} = E_{x \sim X} [||Cycle_X - X||_1]$$

```
cycle_x = gen_X(fake_y)
```

```
cycle_x_loss = l1(x, cycle_x)
```

- Calculate the cycle consistency loss for gen_Y as:

$$L_{Cycle-Y} = E_{y \sim Y} [||Cycle_Y - Y||_1]$$

```
cycle_y = gen_Y(fake_x)
```

```
cycle_y_loss = l1(y, cycle_y)
```

- Identity loss as follows for the gen_X and gen_Y:

$$L_{Identity-X} = E_{x \sim X} [||Identity_X - X||_1]$$

```
identity_x = gen_X(x)
```

```
identity_x_loss = l1(x, identity_x)
```

$$L_{Identity-Y} = E_{y \sim Y} [||Identity_Y - Y||_1]$$

```
identity_y = gen_Y(y)
```

```
identity_y_loss = l1(y, identity_y)
```

- Adding all the generator losses together:

```
G_loss = (loss_G_Y + loss_G_X + cycle_y_loss * LAMBDA_CYCLE  
+ cycle_x_loss * LAMBDA_CYCLE + identity_x_loss *  
LAMBDA_IDENTITY + identity_y_loss * LAMBDA_IDENTITY)
```

- Back propagation for the generators gen_X and gen_Y is performed based on this added up G_loss variable value with adam optimizer.
- For certain epochs (where epoch id gives remainder 0 when divided with NUM_EPOCHS, here 150), we save the images of both the “x” domain and fake output “y” domain images in the saved_images directory.

EXTRA NOTES:

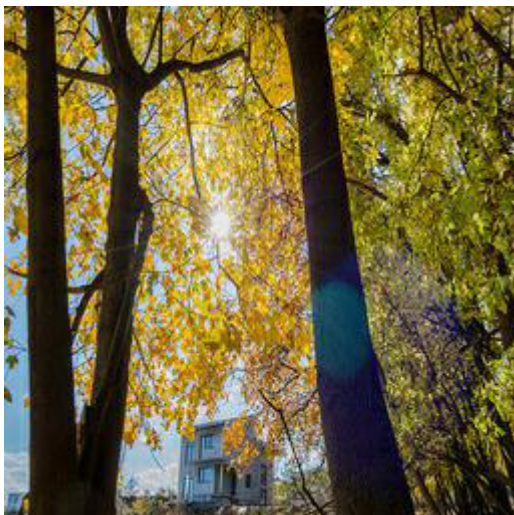
The network was trained on a Kaggle free tier instance with GPU TESLA P100, 2 Cores, and 13 GB RAM.

The code for building architecture, training & testing, and the results can be viewed at: https://github.com/sklohani/Image2Painting_CycleGAN

Results

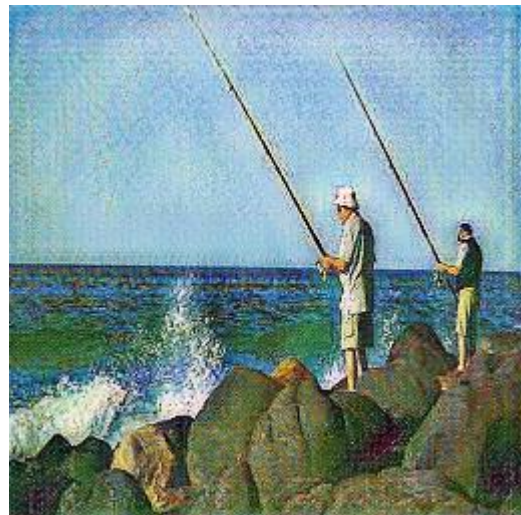
The results obtained from this project are shown below. The input images are taken from various sources like the Kaggle dataset used for training and testing (X domain), images captured from the camera, Monet paintings, etc.

The left-hand side image is the input image given to the trained generator (Gen_y) and the right-hand side image is the translated output or the generated Van Gogh Painting.



Images from the train dataset (X domain)

Translated Van Gogh Painting



Images from the test dataset (X domain)

Translated Van Gogh Painting



School of Computer & Information Sciences, UoH



Translated Van Gogh Painting







Administration Block, UoH



Translated Van Gogh Painting



School of Life Science, UoH	Translated Van Gogh Painting
	
Image of Hand-drawn painting of Joker	Translated Van Gogh Painting
	
A beautiful lake in Warangal	Translated Van Gogh Painting



Monet Paintings

Translated Van Gogh Painting



Van Gogh Paintings

Translated Identity Van Gogh Painting

Conclusion & Future Work


The project aims to generate paintings in a specific style from input images. The report discusses different architectures that can be used for this purpose, including the Encoder-Decoder Network, Neural Style Transfer, and the CycleGAN. Due to the absence of paired images for training and the desire to utilize the relationship between the picture and painting domains, the CycleGAN architecture was chosen for implementation. The report provides a detailed explanation of the CycleGAN architecture, highlighting its cyclic nature and how both generators and discriminators work together to produce the desired painting output. It also delves into the various loss functions employed in this architecture, such as Adversarial Loss, Cycle Consistency Loss, and Identity Loss. It explains how these losses are calculated and combined to enhance the network's efficiency and robustness.

The Kaggle dataset used for training and testing the network is presented, along with a detailed explanation of the training process of the network involving both generators and discriminators.

The report concludes by presenting the results obtained from the CycleGAN. It showcases the generated paintings for input images sourced from diverse origins, including training and testing data, images captured from a camera, Monet paintings, and even the Van Gogh painting itself. While the model's performance is generally satisfactory, it struggles with input images unrelated to nature, such as people or buildings. Notably, there are some exceptions to this trend. The bias towards nature scenes in the training dataset (X domain) might contribute to this disparity in performance.

Our future plan is to train the network for more epochs and introduce an image buffer that stores 50 previously generated images and update the discriminators using image buffer rather than the ones produced by the latest generative networks.

References

- [1] [\[1508.06576\] A Neural Algorithm of Artistic Style](#)
- [2] [\[1703.10593\] Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks](#)
- [3] [CycleGAN: Learning to Translate Images \(Without Paired Training Data\)](#)
[| by Sarah Wolf](#)
- [4]  [Unpaired Image-Image Translation using CycleGANs](#)
- [5] [I'm Something of a Painter Myself | Kaggle](#)
- [6] [Best Artworks of All Time | Kaggle](#)