République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur
et de la Recherche Scientifique
ECOLE SUPÉRIEURE EN INFORMATIQUE
8 Mai 1945 - Sidi-Bel-Abbès

الجمهورية الجزائرية الديمقراطية الشعبية
وزارة التعليم العالي والبحث العلمي
المدرسة العليا للإعلام الآلي
8 ماي 1945 - سيدي بلعباس

Neural Style Transfer With GAN

MINI PROJECT DEEP LEARNING

Team Members:

- Benghenima Hafsa
- Ghandouz Amina

Professor:

- Mrs Dif Nassima

Contents

List of Figures

1 Introduction:

In recent years, the intersection of computer vision and artistic creativity has produced fascinating developments, one of which is neural style transfer, which means the ability of deep learning models to manipulate or recreate artistic styles in images. Neural style transfer techniques aim to apply the visual appearance (style) of an image, typically an artwork, to the content of another image, such as a real photograph. While earlier approaches were based on optimization techniques (e.g Gatys), recent advancements leverage **Generative Adversarial Networks (GANs)** to achieve style transfer in a more realistic, efficient, and scalable way.

GANs, introduced by **Ian Goodfellow and his colleagues** in **June 2014**, are composed of two competing neural networks, a generator and a discriminator, they learn to produce increasingly realistic data through adversarial training. In the context of image-to-image translation, GANs have proven powerful in generating photorealistic outputs from sketches, semantic labels, or artworks. However, traditional GAN-based translation methods often require paired datasets which are scarce or unavailable in many real-world scenarios.

To overcome this limitation, **CycleGAN** introduced a framework that allows unpaired image-to-image translation using cycle-consistency loss. This enables learning a mapping between two domains even when corresponding image pairs do not exist.

In this project, we apply neural style transfer using **CycleGAN** to the **Monet2Photo** dataset that includes two distinct domains:

- Monet paintings (representing the impressionist artistic style of **Claude Monet**)
- Real-world photographs of landscapes and scenes similar in content

The objective is to learn two mappings:

- From Monet-style paintings to realistic photos
- From photos to Monet-style artworks

This task not only showcases the capabilities of unpaired style transfer models but also highlights how deep learning can bridge the gap between art and reality. Moreover, the Monet2Photo dataset provides a unique opportunity to experiment with the visual translation of impressionist textures, brushstrokes, and color palettes into the domain of modern photography.

2 Background and Related Work:

2.1 What is Style Transfer?:

Style Transfer is the process of merging the content of one image with the style of another, resulting in a new and recognizable image. In more technical terms, it is a method that allows to take the structure and layout of one image (*content*) and paint it in the unique style, textures, and patterns of another image (*style*), it's exactly like telling a machine, *"Make this photograph look like it was painted by Van Gogh"*.

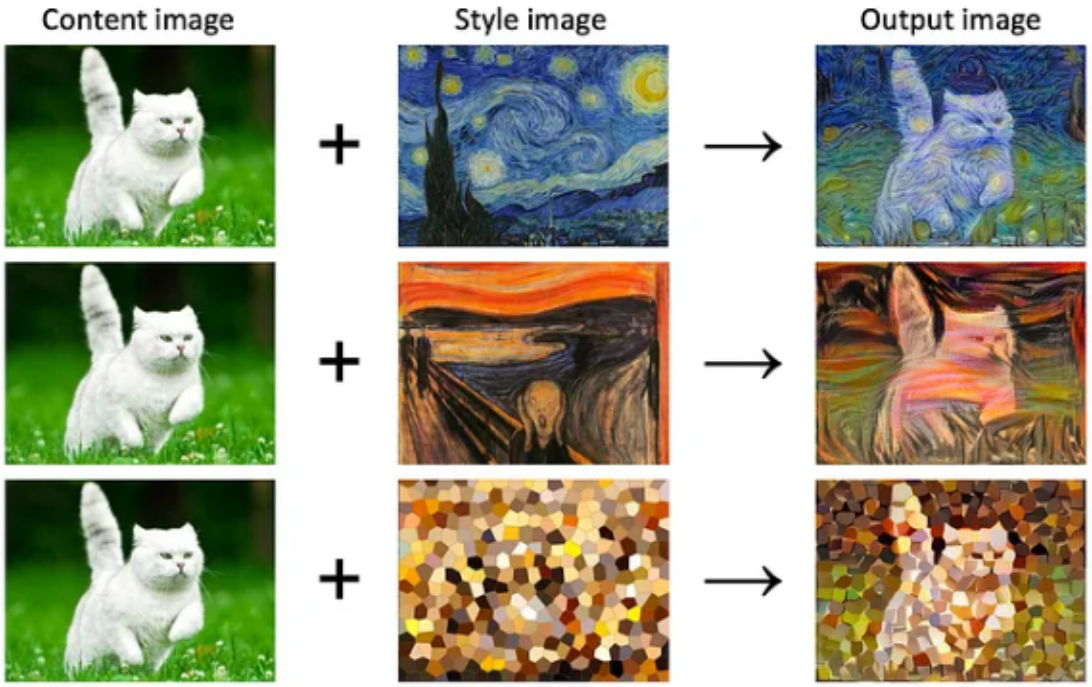


Figure 1: Style Transfer in Computer Vision.

Style transfer can be broadly categorized into:

- **Traditional Style Transfer:** Uses techniques like histogram matching, edge detection, and texture synthesis. These are often rule-based and work well only for simple cases.
- **Neural Style Transfer (NST):** In **2015**, **Gatys et al.** introduced **NST**, revolutionizing image generation by using CNNs to separate and recombine image content and style. Their method allowed computers to blend objects and artistic textures convincingly.

In the context of **style transfer**, *content* refers to the higher-level features of the image, like shapes, objects, and their spatial arrangement, it's the structure or the *what* of the image. On the other hand, style represents the lower-level features like textures, colors, and patterns, the style is the *how* or the feel of the image. For example, if we have a photo of a mountain range, the content would be the actual layout of the mountains, sky, and ground. If we apply the style of a *Van Gogh* painting, vibrant color palette would be transferred, but the mountains would still be recognizable.

In **Style Transfer**, the network needs to balance two things: *retaining the content* and *applying the style* so we use *loss functions* to measure how well it's doing this balancing act. NST optimizes for

two losses:

- **Content loss:** measures how close the generated image is to the original content.
- **Style loss:** measures how similar its textures and patterns are to the target style.

At the heart of style transfer is **feature extraction**, this is how the algorithm gets a sense of what's in the image. CNN will be as seasoned art critic that knows how to separate content from style. When an image is passed through a CNN, its early layers detect textures and colors, while deeper layers identify complex shapes and structures. This layered understanding helps distinguish between the image's content (overall layout) and style (visual patterns and textures).

2.2 What is CNN?:

Convolutional Neural Networks (CNNs) are a type of deep learning neural network architecture that is particularly well suited to image classification and object recognition tasks. A CNN starts by taking an input image, which is then transformed into a feature map through a series of convolutional and pooling layers, the convolutional layer applies a set of filters to the input image where each filter will produce a feature map that highlights a specific aspect of the input image, the pooling layer then downsamples the feature map to reduce its size, while retaining the most important information. The feature map produced by the convolutional layer is then passed through multiple additional convolutional and pooling layers, each layer learning increasingly complex features of the input image and the final output of the network is a predicted class label or probability score for each class for multi-classifiifcation tasks.

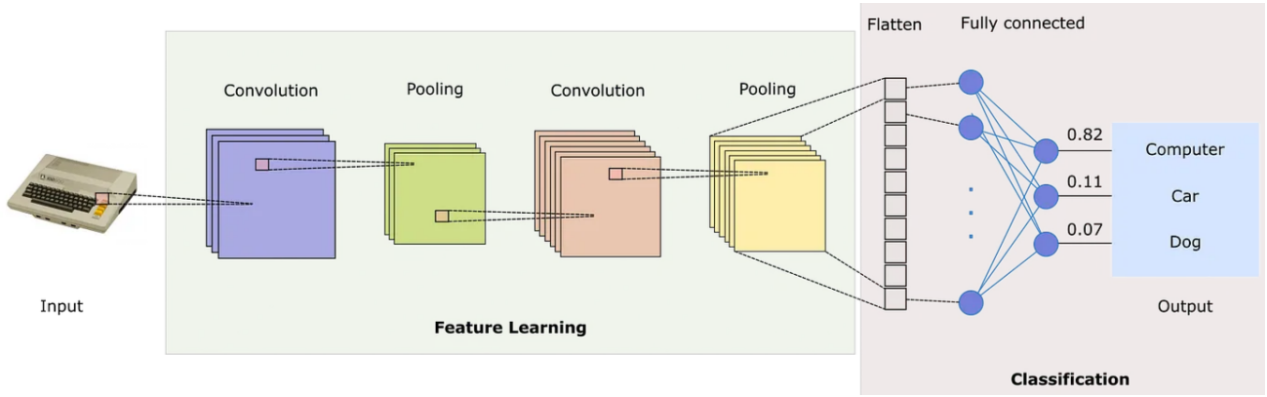


Figure 2: CNN architecture.

2.3 How do CNNs work?:

CNNs are composed of layers of artificial neurons, each responsible for transforming the input image in a specific way.

1. **Neurons:** The most basic unit in a neural network, they are composed of a sum of linear/non-linear function (the activation function).

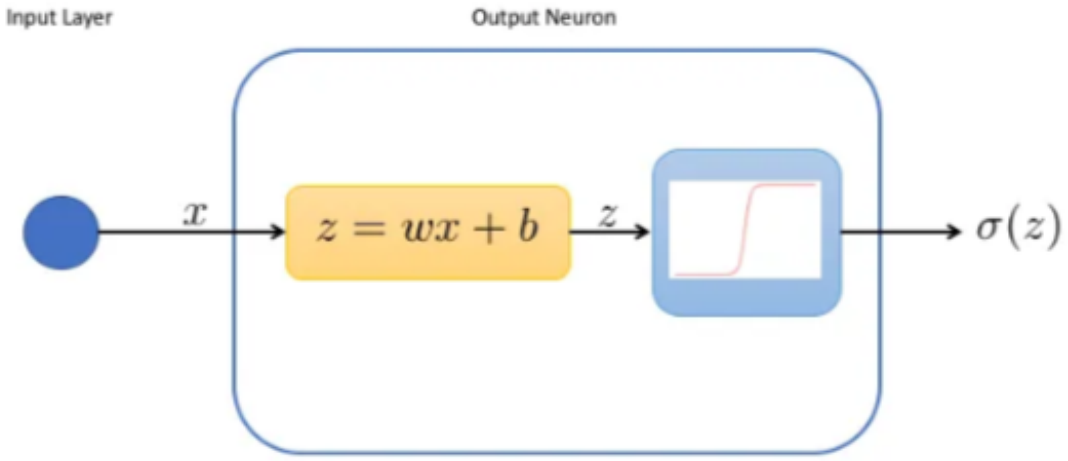


Figure 3: Neuron representation

2. **Input Layer:** each neuron in this layer corresponds to one of the input features, for example in an image classification task where the input is 32x32, the input layer would have 1024 neurons which means one neuron for each pixel.
3. **Hidden Layer:** is the layer between input and output ones, they may be more, each neuron in the hidden layer is summed by the result of the neurons in the previous layers then multiplied by non-linear function.
4. **Output Layer:** in this layer, the number of neurons corresponds to the number of the classes, for example in multi-classification task with digits 0-9, we would have 9 neurons in the output layer.

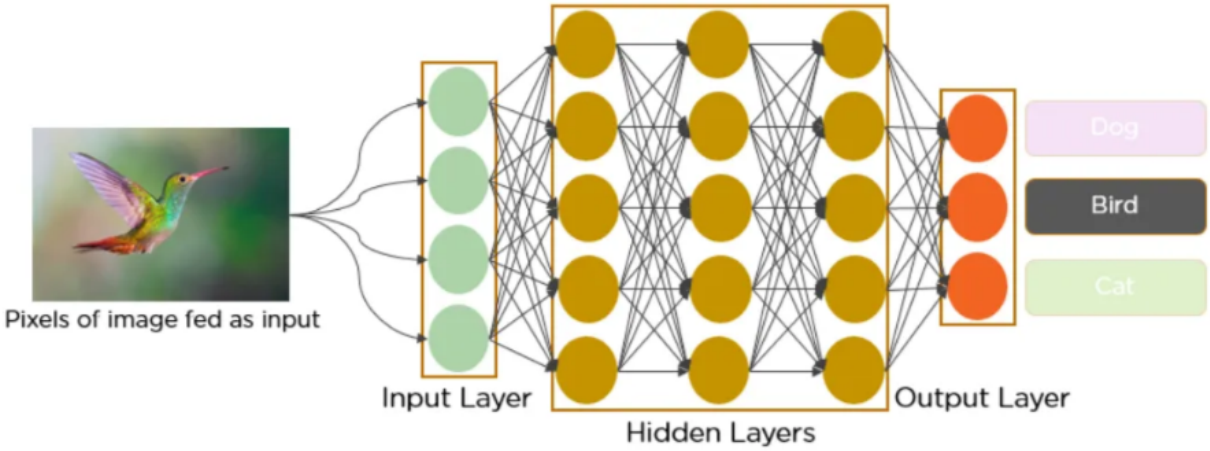


Figure 4: CNN example

Convolutional layer are what makes a CNN different from a basic neural network, they are the fundamental building blocks of CNNs, these layers perform a critical mathematical operation known as convolution. We can classify the layers of CNN into:

- **Convolutional Layer:** is responsible for extracting features from the input image, it performs a convolution operation on the input image, where a filter or kernel is applied to the image to identify and extract specific features.

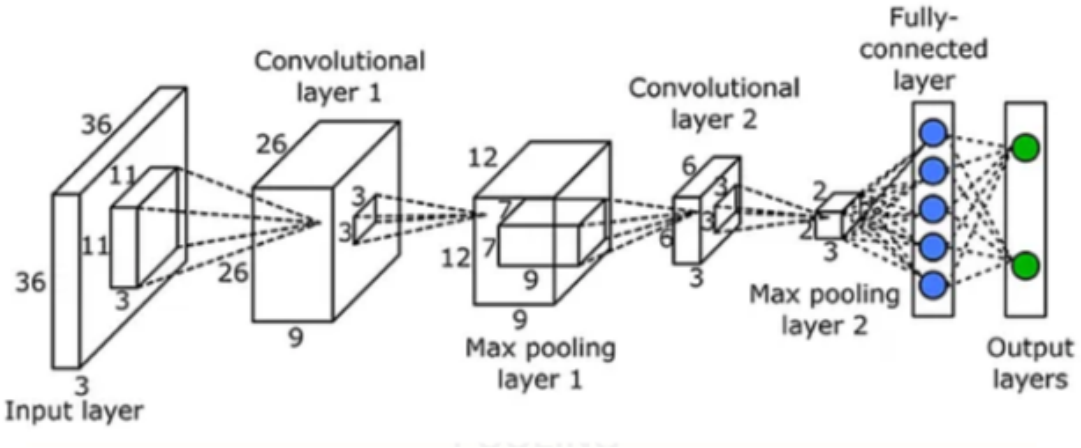


Figure 5: Convolutional Layer

- **Pooling Layer:** is responsible for reducing the spatial dimensions of the feature maps produced by the convolutional layer, it performs a down-sampling operation to reduce the size of the feature maps and reduce computational complexity.

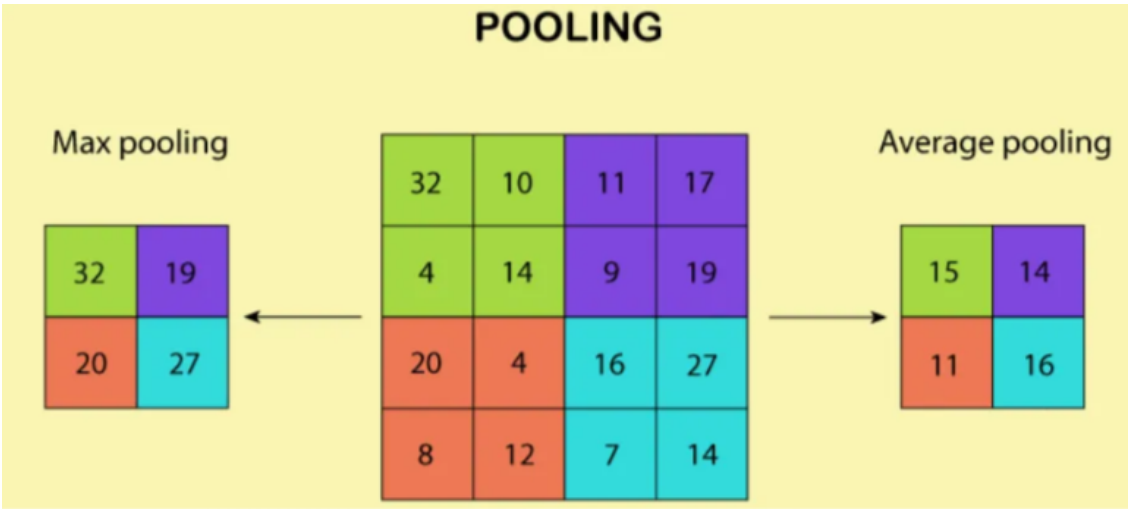


Figure 6: MaxPooling Layer

- **Activation Layer:** applies a non-linear activation function, such as the ReLU to the output of the pooling layer, this function helps to introduce non-linearity into the model, allowing it to learn more complex representations of the input data.

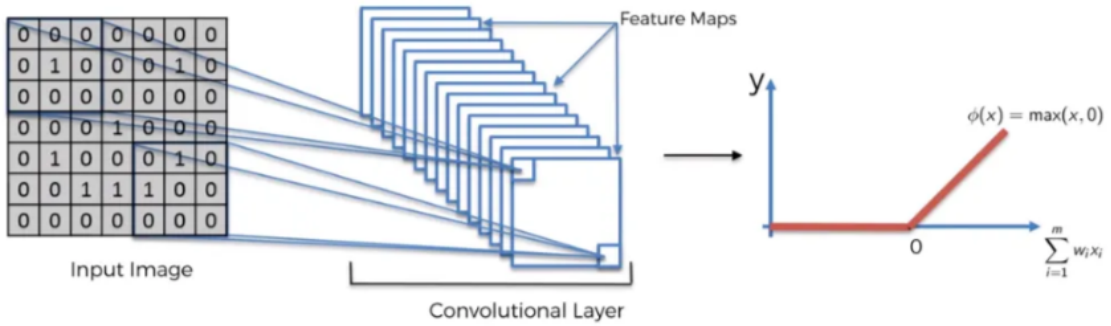


Figure 7: Activation Layer

- **Fully Connected Layer:** is a traditional neural network layer that connects all the neurons in the previous layer to all the neurons in the next layer, it's responsible for combining the features learned by the convolutional and pooling layers to make a prediction.

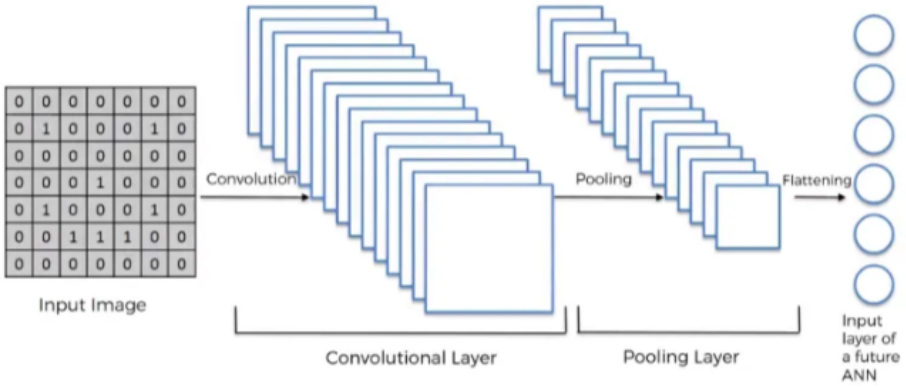


Figure 8: Fully Connected Layer

- **Normalization Layer:** performs normalization operations such as batch normalization to ensure that the activations of each layer are well-conditioned and prevent overfitting.
- **Dropout Layer:** is used to prevent overfitting by randomly dropping out neurons during training which helps to ensure that the model does not memorize the training data but instead generalizes to new, unseen data.
- **Dense Layer:** after the convolutional and pooling layers have extracted features from the input image, this layer can then be used to combine those features and make a final prediction. In a CNN, the dense layer is usually the final layer and is used to produce the output predictions. The activations from the previous layers are flattened and passed as inputs to the dense layer which performs a weighted sum of the inputs and applies an activation function to produce the final output.

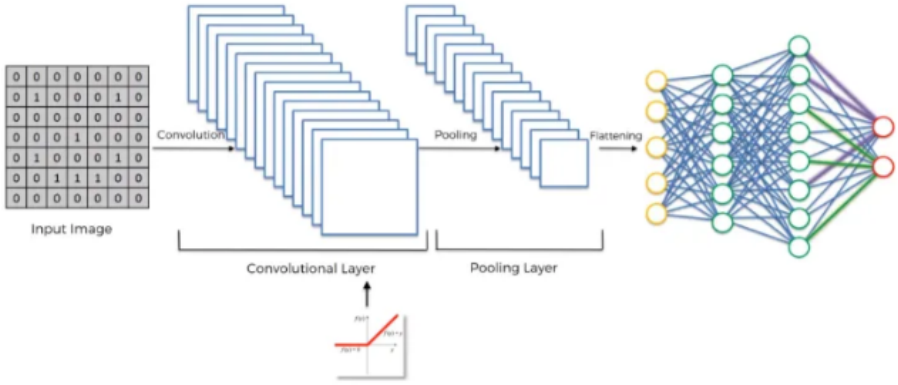


Figure 9: Dense layer

CNNs are a powerful deep learning architecture well-suited to image classification and object recognition tasks with its ability to automatically extract relevant features, handle noisy images, and leverage pre-trained models.

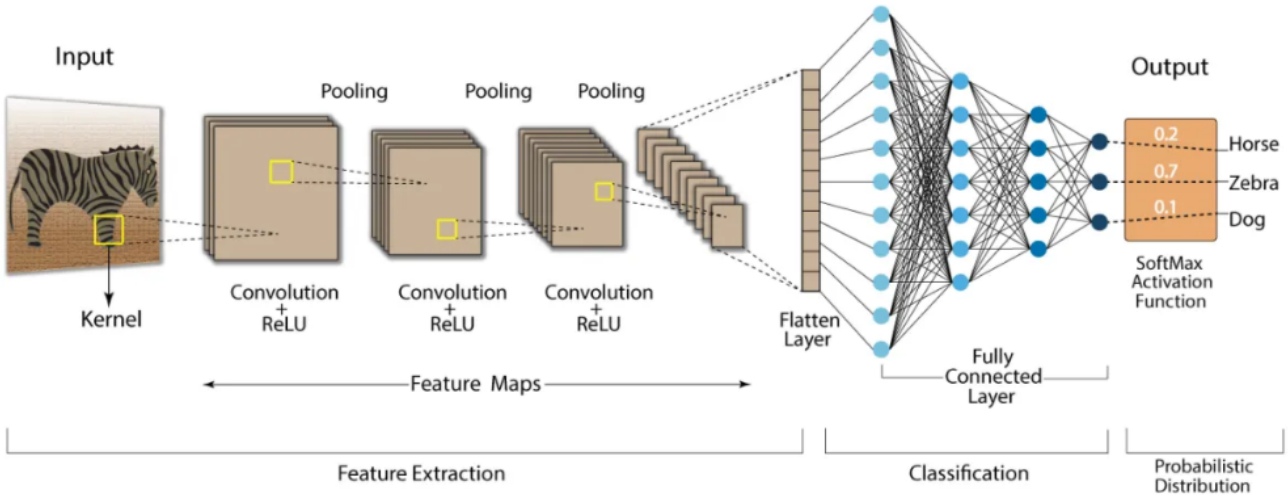


Figure 10: CNN representation

Despite their strong capabilities in feature extraction and classification tasks, *CNNs* face limitations when it comes to generating new and high-quality images. Traditional CNNs are primarily discriminative models which means they excel at recognizing and analyzing existing data, but they are not designed to create new content and this becomes a challenge in tasks like image synthesis or style transfer where realism and creativity are crucial. To overcome these limitations, **Generative Adversarial Networks** (*GANs*) were introduced. Unlike CNNs, GANs are generative models capable of producing realistic and high-resolution images by learning the underlying data distribution, making them a powerful alternative in the field of generative image modeling.

In *CNNs*, one of the most powerful features is their ability to extract hierarchical representations from images which makes them excellent for tasks like classification and style-content separation. In

GANs, especially in the generator, feature extraction is also crucial, it helps the generator learn the structure and texture of images to produce realistic outputs but there's a key difference:

- In CNNs, feature extraction is the main goal, the model is often used to understand or classify an image.
- In GANs, feature extraction is a means to an end, it helps the generator/discriminator learn how to create or judge images, not just understand them.

Which means that feature extraction is central to both, but it plays a supporting role in *GANs* (for generation and discrimination), and a primary role in *CNNs* (for recognition and analysis).

2.4 What is GAN?:

2.4.1 Introduction:

Deep learning models can be broadly classified into:

- **Generative models** learn the distribution of the data itself. They model the joint probability *P(x,y)* and can generate new data samples that are similar to the training data. An example of a generative model is a **Generative Adversarial Network** (*GAN*), which consists of a generator that creates data and a discriminator that evaluates the authenticity of the generated data.
- **Discriminative models** learn the decision boundary between different classes, they focus on modeling the conditional probability where y is the label and x is the input data. These models are used primarily for classification tasks. A common example of a discriminative model is a *CNN*, which is widely used for image classification tasks where the model classifies images into predefined categories.

2.4.2 What is GAN?:

Generative Adversarial Networks are a class of deep learning frameworks designed by **Ian Goodfellow and his colleagues** in *2014*. They consist of two interlinked neural networks:

- **the generator:** responsible for creating data from random noise.

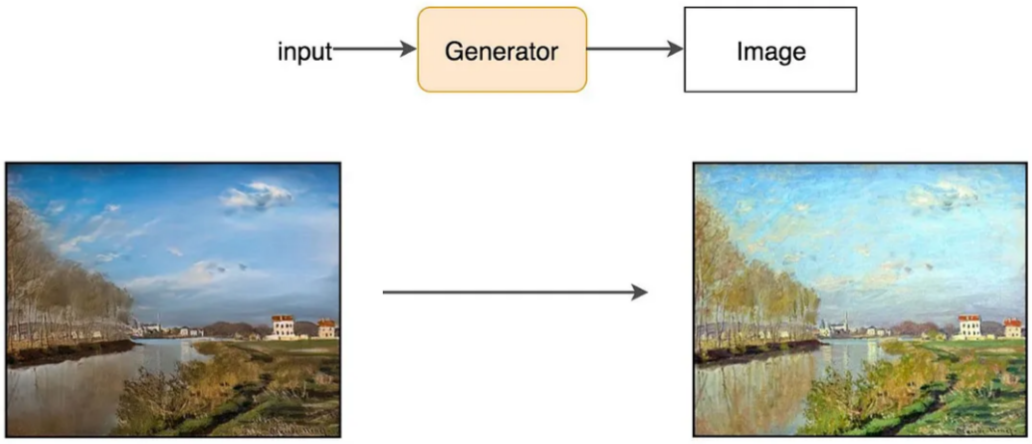


Figure 11: Generator example

- **the discriminator:** whose task is to differentiate between genuine and simulated data.

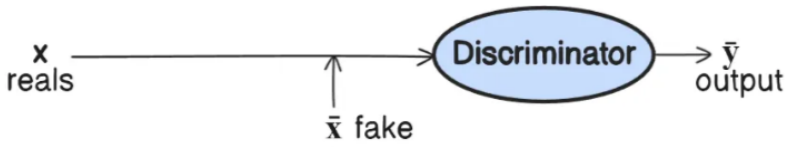


Figure 12: Discriminator example

They are trained simultaneously by competing against each other and this interplay results in a unique training process, with GANs operating in an "adversarial" manner formulated as a supervised learning challenge, the *Generator* tries to produce data that the *Discriminator* can't distinguish from real data, while it tries to get better at differentiating real data from fake data. The two networks are in essence competing in a game: the *Generator* aims to produce convincing fake data, and the *Discriminator* aims to tell real from fake which leads to the Generator creating increasingly better data over time and allows GANs to produce highly realistic synthetic data.

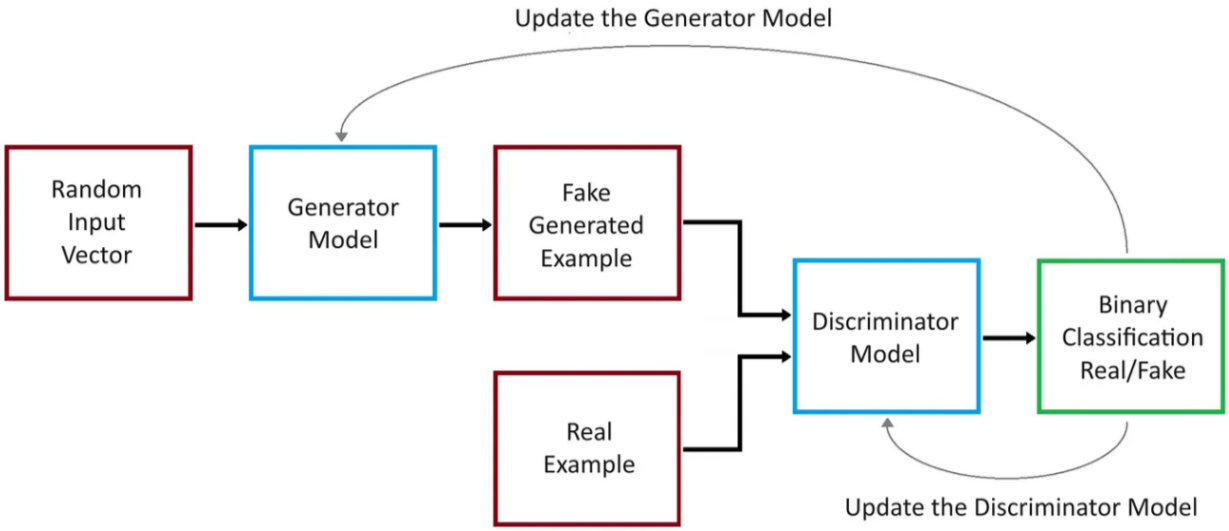


Figure 13: GAN Architecture

2.4.3 Loss Functions:

In *GANs*, the loss function is the core part used to guide the optimization of the generator and the discriminator. The goal of a *GAN* is to generate realistic data through the generator so that the discriminator cannot distinguish between real data and generated data. The loss function of a *GAN* usually includes the loss of the generator and the loss of the discriminator so there's two main loss functions guide the training process:

- **The discriminator loss:** evaluates how well the discriminator can distinguish between real and generated (fake) data, it's trained to maximize the probability of correctly identifying real images as real and generated images as fake. For real samples, the discriminator should output a value close to 1 (indicating the probability of being true), and for generated samples, the discriminator

should output a value close to 0 (indicating the probability of being false), the discriminator loss function can be written as:

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

Where:

- x pdata(x) represents the real data sampled from the real data distribution.
- z pz(z) is the noise sampled from the noise distribution, which is input to the generator to generate fake data G(z).
- D(x) is the output of the discriminator for real data (close to 1), and D(G(z)) is the output of the discriminator for generated data (close to 0).

- **The generator loss:** measures how successfully the generator can fool the discriminator, it hopes to make the discriminator think that the generated data is *real* wo it aims to minimize this loss by producing images that are realistic enough to be classified as real by the discriminator so the loss function of the generator is opposite to that of the discriminator, it can be written as:

$$\mathcal{L}_G = -\mathbb{E}_{z\sim p_z(z)}[\log(D(G(z)))] \tag{2}$$

Here, the generator tries to minimize this loss function, that is, to make the D(G(z)) given by the discriminator as close to 1 as possible, which means that the generator hopes that the generated data can deceive the discriminator.

The interplay between these two losses creates a dynamic adversarial training process where both models improve simultaneously.

Overall loss: During the training process of *GAN*, the discriminator and the generator are adversarial. The discriminator maximizes its ability to distinguish between real data and generated data, while the generator minimizes the ability of its generated data to be distinguished by the discriminator. The loss function of *GAN* is usually interpreted as an optimization problem of a two-person zero-sum game. The final loss function can be written as:

$$\min_G \max_D \; V(D,G) = \mathbb{E}_{x\sim p_{\text{data}}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))] \tag{3}$$

- Discriminator loss reflects whether it can correctly distinguish between real data and generated data. When the discriminator loss is low, it means that it can distinguish between the two well.
- Generator loss reflects whether the data it generates is realistic enough to deceive the discriminator. When the generator loss is low, it means that the data generated by the generator has made it difficult for the discriminator to distinguish between true and false.

2.4.4 Variants of GANs:

- **Conditional GANs (C-GANs):** GANs that generate data based on additional input information (e.g., class labels, text, or attributes) which allows control over the generated output.
- **CycleGAN:** Designed for image-to-image translation tasks where paired training data is not available (e.g., translating horses to zebras without having exact matching images).

- **DC-GAN:** Introduces convolutional and transposed convolutional layers in place of fully connected layers for better image quality and more stable training.
- **Wasserstein GAN (W-GAN):** Uses the *Wasserstein distance* instead of the traditional loss, which improves training stability and helps mitigate mode collapse (where the generator produces limited varieties of outputs).

2.5 What is CycleGAN?:

2.5.1 Introduction:

CycleGAN is a powerful variant of *Generative Adversarial Networks* (*GANs*) that enables image-to-image translation without the need for paired training data. Traditional supervised GAN approaches require aligned image pairs (e.g., a photo and its corresponding painting), which can be expensive or impossible to obtain. So, what *CycleGAN* does differently from a standard *GAN* is that it doesn't generate images from random noise, instead, it uses a given image to get a different version of that image, this is the image-to-image translation that allows *CycleGAN* to change a horse into a zebra. However, image-to-image translation is not a feature that is unique to *CycleGAN* but it's one of the first models to allow for unpaired image-to-image training. What that means is that we don't have to have a picture of a horse and a picture of what that horse would look like as a zebra in the dataset. Instead, we can have a bunch of horses and a bunch of zebras separately. This is useful in situations where we aren't able to get paired data.

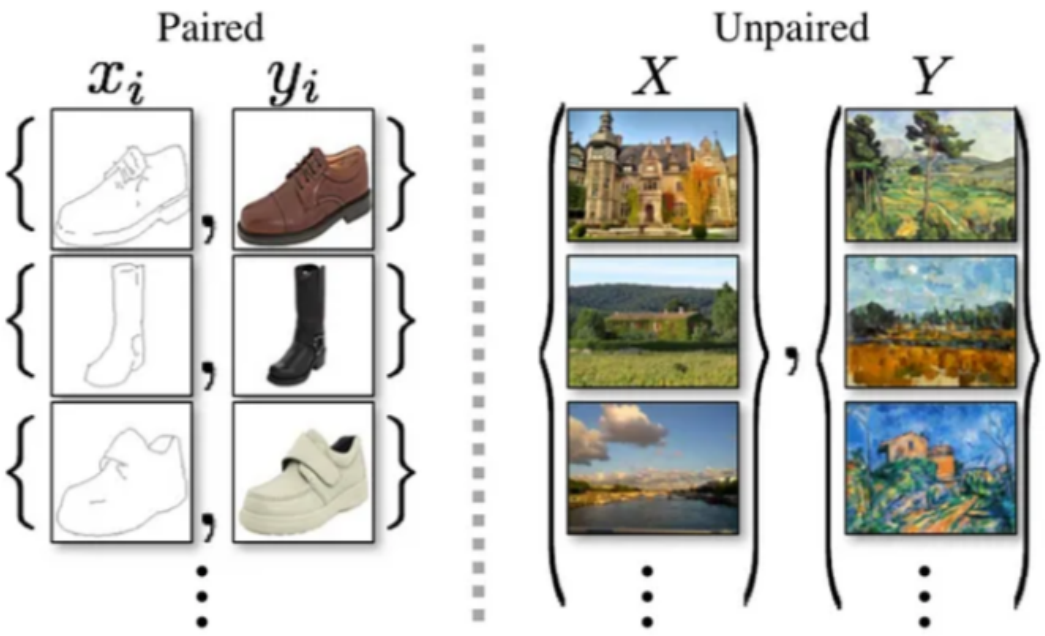


Figure 14: Paired and unpaired dataset

While traditional *GANs* consist of a single generator and discriminator and require paired datasets, *CycleGAN* removes the need for aligned image pairs, instead, it uses two generators (one for each direction of translation, e.g., A $\rightarrow$ B and B $\rightarrow$ A) and two discriminators (one for each domain). A key innovation in *CycleGAN* is the *cycle consistency loss*, which ensures that translating an image to the

target domain and back results in the original image (i.e., A → B → A ≈ A). This architecture allows *CycleGAN* to learn meaningful mappings between domains even without direct correspondence, making it particularly useful in real-world applications like artistic style transfer, medical imaging, and domain adaptation.

2.5.2 Why we use CycleGAN?:

The *Cycle-GAN* architecture was proposed in the paper, *Unpaired image-to-image Translation Cycle-Consistent Adversarial Networks. Jan-Yan Zhu and his colleagues (2017)* suggested that *Pix2Pix* can produce truly amazing result but the challenge is in training data, to convert one image to another image is a bit difficult as well as time-consuming, infeasible or even sometimes impossible based on which image we were trying to translate(one to one match between two images). This is where *CycleGAN* comes in, the key idea behind it is that they allow us to point the model at two unpaired collection of the images.

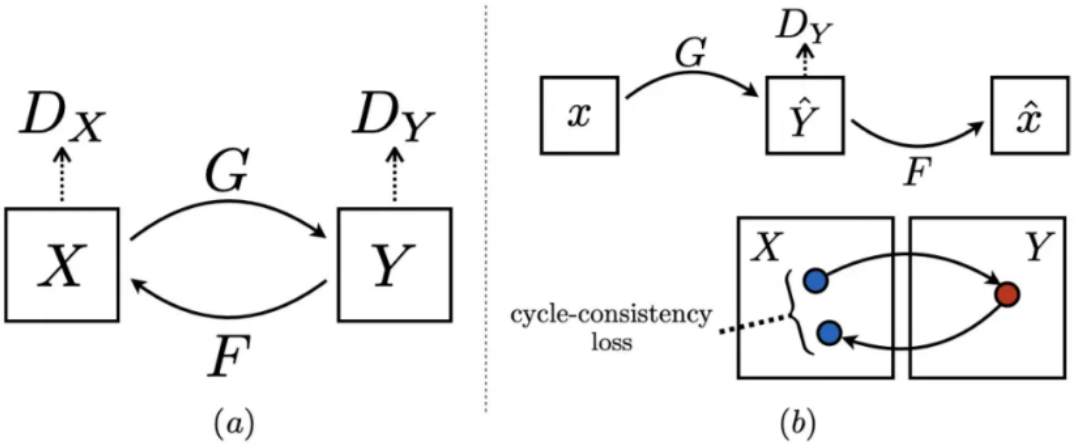


Figure 15: Simple translation of Cycle GAN

- **Role of G:** G is trying to translate X into outputs, which are fed through Dy to check whether they are real or fake according to Domain Y.
- **Role of F:** F is trying to translate Y into outputs, which are fed through Dx to check if they are indistinguishable from Domain X.

The Cycle-consistency loss is like if our input image is A from domain X is transformed into a target image or output image B from domain Y via Generator G, then image B from domain Y is translated back to domain X via Generator F. So that time the difference between these two images is called as the *CycleConsistency loss*. This approach requires creating two pairs of generators and discriminators: one for A2B (real photo to monet image conversion) and another for B2A (monet image to real photo conversion).

2.6 The Generator architecture:

As we did mention before, *CycleGAN* contains two generators, each one has three parts:

1. **Encoder (convolutional block):** is responsible for extracting features. The initial step is separating the highlights from a picture that is done a convolution network. As input, a CNN

takes a picture, size of filter window that we move over input picture to excerpt out features and the Stride size to choose the amount we will move filter window after each progression. Every convolution layer prompts the extraction of dynamically more elevated level highlights. This stage includes three 2D convolutional layers followed by an instance normalization layer and an activation function (ReLU).

2. **Transformer (residual block):** We have utilized 9 layers of resnet blocks, it's a neural network layer that comprises of two convolution layers where a buildup of info is added to the yield. This is done to guarantee properties of the contribution of past layers are accessible for later layers also with the goal that their yield doesn't digress much from unique information, in any case, the qualities of unique pictures won't be held in the yield and results will be unexpected. The encoder output passes through this stage which mainly consists of 6 to 9 residual blocks (we use 9). Each block is a set of 2D convolutional layers, with every two layers followed by an instance normalization layer.

3. **Decoder (transposed convolutional block):** The decoding step is the specific inverse of Step 1, we will work back the low-level features once again from the element vector. This is finished by applying a deconvolution (or transpose convolution) layer. The transformer output will pass through this stage, which consists of two upsampling blocks. Each upsampling block is a transposed convolution layer followed by a ReLU activation function. These two deconvolution blocks increase the size of representation of the processed images coming from the transformer to its original value.

The generator features a final 2D convolutional layer that uses the tanh activation function. This layer allows the generation of images of size equal to the size of the original input images.

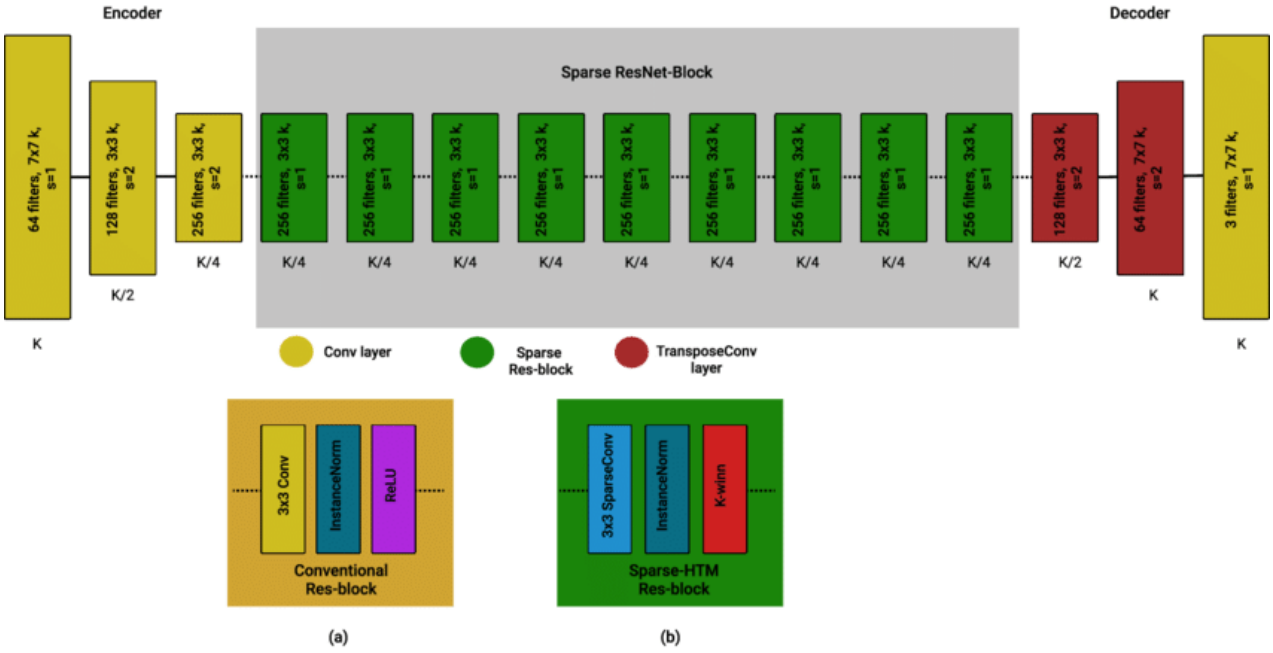


Figure 16: CycleGAN Generator

2.7 The Discriminator architecture:

A *CycleGAN discriminator* is a typical *CNN* that includes multiple convolutional layers. This network takes an input image and classifies it as real or fake so the CycleGAN discriminator is different from that used in the regular GAN, the latter maps input from a 256 by 256 image to a single scalar output, which represents *real* or *fake*, where the CycleGAN discriminator maps from the 256 by 256 image to an N by N array of outputs X. In that array, each Xij signifies whether the patch ij in the image is real or fake. The diagram below shows the typical architecture of a CycleGAN discriminator.

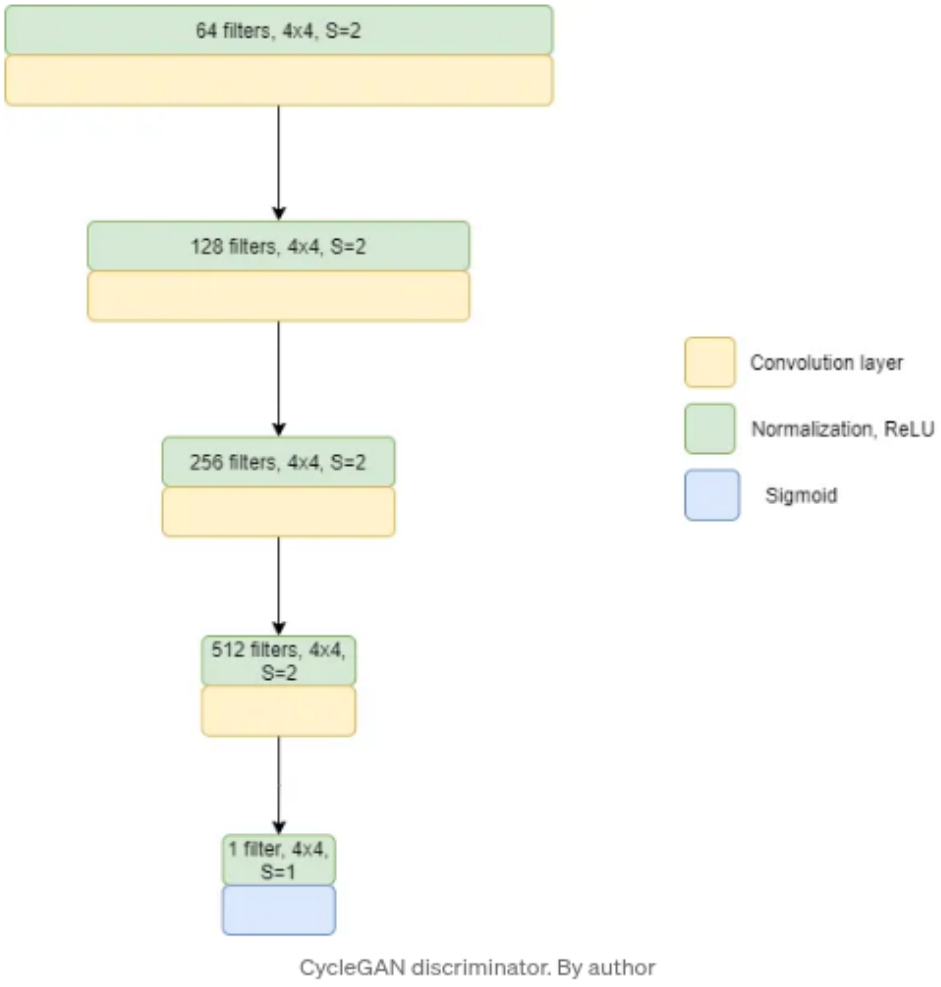


Figure 17: CycleGAN discriminator

CycleGAN uses a *PatchGAN* rather than a *DCNN* as it is discriminator model which helps it to classify patches of image as real or fake instead of an entire image. The discriminator is convolutionally run across the image where in we average all the responses to give the final output. The network outputs a single feature map of real and fake predictions that is averaged to give a single score. 70x70 patch size is considered to be effective across different image-to-image translation task. Hence Patch GAN has the ability to predict whether each 70 x 70 patch in the input image is real or fake. The discriminator model described in the paper uses blocks of **Conv2D-InstanceNorm-LeakyReLU** layers with 4x4 filters and 2x2 strides. The architecture is *C64-C128-C256-C512*. After the last layer a convolution is applied

to produce a 1D output. Instance normalization which involves standardizing the values on each feature map in order to remove image specific contrast information is used instead of batch normalization.

2.8 Loss Function:

The next target is to design the loss function, it can be seen having some parts:

1. The Discriminators must accept all the images of all categories.
2. The Discriminators should reject all the images which the generator tries to fool them.
3. The Generators try to approve all images from discriminator to fool them.
4. The generated image returns the original image, so the generator generates a fake image using A→B then we should almost certainly return to a unique picture utilizing another Generator B→A which means it must fulfill cyclic-consistency.

2.8.1 Adversarial Loss:

This loss is similar to the one used in the regular *GAN*. However, in *CycleGAN*, adversarial loss is applied to both generators that are trying to generate images of their corresponding domains. A generator aims to minimize the loss against its discriminator in order to finally generate real images. It's calculated as follows:

$$\mathcal{L}_{GAN}(G, D_y, X, Y) = \mathbb{E}_{y \sim p_{data}(y)}[\log(D_y(Y))] + \mathbb{E}_{x \sim p_{data}(x)}[\log(1 - D_y(G(x)))] \tag{4}$$

where G tries to generate images $G(x)$ that look similar to images from domain Y , while D_y aims to distinguish between translated samples $G(x)$ and real samples y. G aims to minimize this objective against an adversary D that tries to maximize it, i.e., min_G, max_{D_Y}, $\mathcal{L}_{GAN}(G, D_Y, X, Y)$.

2.8.2 Cyclic Consistency loss:

One of the most critical loss is the **Cyclic loss** where we can achieve the original image using another generator and the difference between the initial and last image should be as small as possible. It's calculated by the sum of GAN forward and backward cycle loss:

$$\mathcal{L}_{cyc}(G, F) = \mathbb{E}_{x \sim p_{data}(x)}[\|F(G(x)) - x\|_1] + \mathbb{E}_{y \sim p_{data}(y)}[\|G(F(y)) - y\|_1] \tag{5}$$

Adversarial training tries to make the output of the generator look like it came from the target domain. For example:

- for $G : X \to Y$, it makes sure $G(x)$ looks like it's from domain Y.
- And for $F : Y \to X$, it makes sure $F(y)$ looks like it's from domain X.

But the problem is, the generator might learn to map all photos of cats to random photos of dogs as long as these dogs do look like real dogs, so the discriminator is happy, but we don't want this, we're looking for each specific cat to become a specific-looking dog with consistent structure. This problem

may happen because the adversarial loss doesn't force the output to relate meaningfully to the input. As a solution, they did introduce the **Cycle Consistency Loss**, if we translate a cat to a dog using G, then translate it back using F should give use the original cat, means $F(G(x)) \approx x$ which forces the generator to remember details about the input and discourages random mappings.

2.8.3 Identity Loss:

This loss encourages the generators to preserve color composition between the input and output, it's applied by feeding real samples from the target domain into the generator and penalizing deviation. It's only used as a *regularization term* and calculated by:

$$\mathcal{L}_{identity}(G, F) = \mathbb{E}_{y \sim p_{data}(y)}[\|G(y) - y\|_1] + \mathbb{E}_{x \sim p_{data}(x)}[\|F(x) - x\|_1] \tag{6}$$

2.8.4 Total Generator Loss:

It combines *adversarial loss*, *cycle consistency loss* and *identity loss*, weighted by hyperparameters as followed:

$$\mathcal{L}_G = \mathcal{L}_{GAN}(G, D_Y, X, Y) + \mathcal{L}_{GAN}(F, D_X, Y, X) + \lambda_{cyc}\mathcal{L}_{cyc}(G, F) + \lambda_{id}\mathcal{L}_{identity}(G, F) \tag{7}$$

2.8.5 Discriminator Loss:

We're interested in the discriminator loss but for a different reason than the generator's loss, because the discriminator helps guide the generator during training so if the discriminator is too weak or too strong, the generator won't learn meaningful mappings, that's why even though we ultimately care more about the quality of the generator's outputs, the discriminator's loss tells us that a very low discriminator loss might means it's too good and rejecting everything which helps the generator improve. It's calculated by:

$$\mathcal{L}_{D_A} = \frac{1}{2}\left[(D_A(\text{real}_A) - 1)^2 + (D_A(\text{fake}_A))^2\right] \tag{8}$$

2.9 Artistic Style Transfer: Transforming Photographs into Paintings:

One popular application of *CycleGAN* is in the transformation of artistic images, specifically, making real-world photographs appear as if they were painted by renowned artists. When we look at a photo, we might ask ourselves:

How would Monet paint this scene?

3 Dataset Description:

The **Monet2Photo dataset** is a well-known dataset used for artistic style transfer, specifically designed for testing the effectiveness of *CycleGAN* in transforming photos into the style of famous artists like *Claude Monet*. It consists of two primary domains: *Monet* and *Photo* images.

Monet2Photo dataset consists of *8231* images, *1193* (*14%*) *Monet Paintings* and *7038* (*86%*) *Natural Photos* which it's splitted into train and test subsets where training set size is *89%* and only *11%* for the test set. We have two main folders, trainA and testA refer to *Monet images* where trainB and testB refer to *Photo images* with *256x256* images size.

In training set, we have:

```
Total images in `/trainA`: 1072
Most common image sizes (in sample): [((256, 256), )]
Total images in `trainB`: 6287
Most common image sizes (in sample): [((256, 256), )]
```

Figure 18: Train images

3.1 Sample Images:

Figure 19: Monet images

Figure 20: Photo images

3.2 Data pre-processing:

Most of the images (Monet and photo images) size is *256x256* but to ensure consistency during training we decided to resize all the images to a uniform size (*256x256* pixels) the converts it to *PyTorch tensor* and scales pixel values from *[0, 255]* to *[0, 1]* before the normalization step where we normalized the obtained image tensor to the range *[-1, 1]*

```
# resize to 256x256 and normalize to [-1, 1]
transform = transforms.Compose([
    transforms.Resize((256, 256)),
    transforms.ToTensor(),  # converts to [0,1]
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))  # converts to [-1, 1]
])
```

Figure 21: Image Transformation Pipeline

After the normalization step, we defines a custom dataset class to load unpaired images from the both domains: Monet paintings (domain A) and real photos (domain B) then wrap it with a **PyTorch Dataloader** to iterate in batches where each batch contains one Monet image and one real photo with randomized order for better training.

```
dataset = ImageDataset("/kaggle/input/monet2photo/trainA", "/kaggle/input/monet2photo/trainB", transform=transform)
dataloader = DataLoader(dataset, batch_size=1, shuffle=True)
```

Figure 22: Dataloader

4 Methodology:

4.1 Model Architecture:

Following the architecture that was proposed in the paper, we used 2 generators, each one attempts to learn the mapping between one image domain and the other (monet painting and real photos):

- $G : Photo \rightarrow Monet$
- $F : Monet \rightarrow Photo$

```python
class ResnetBlock(nn.Module):
    def __init__(self, dim):
        super().__init__()
        self.block = nn.Sequential(
            nn.ReflectionPad2d(1), # padding to preserve spatial size
            nn.Conv2d(dim, dim, 3), # first conv layer
            nn.InstanceNorm2d(dim), # normalization
            nn.ReLU(inplace=True), # activation with ReLU
            nn.ReflectionPad2d(1), # padding
            nn.Conv2d(dim, dim, 3), # second conv layer
            nn.InstanceNorm2d(dim), # normalization
        )

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

# full ResNet generator --part 1
class ResnetGenerator(nn.Module):
    def __init__(self, in_channels=3, out_channels=3, n_blocks=9):
        super().__init__()

        # encoder (initial conv block)
        model = [
            nn.ReflectionPad2d(3), # padding
            nn.Conv2d(in_channels, 64, 7), # large kernel in order to capture more features
            nn.InstanceNorm2d(64),
            nn.ReLU(inplace=True),
        ]
        # encoder --part 2: downsampling layers
        in_features = 64
        for _ in range(2):
            out_features = in_features * 2
            model += [
                nn.Conv2d(in_features, out_features, 3, stride=2, padding=1),
                nn.InstanceNorm2d(out_features),
                nn.ReLU(inplace=True),
            ]
            in_features = out_features

        # transformation layers (ResNet blocks)
        for _ in range(n_blocks): # n_block = 9
            model += [ResnetBlock(in_features)]

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

        # output layer
        model += [
            nn.ReflectionPad2d(3), # padding
            nn.Conv2d(64, out_channels, 7), # out_channels = 3
            nn.Tanh() # for normalization the output to [-1, 1]
        ]

        self.model = nn.Sequential(*model)

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

Figure 23: Generator Implimentation

We used 2 discriminators also:

- D_{Monet}: Distinguishes real Monet paintings from generated ones.
- D_{Photo}: Distinguishes real photographs from generated ones. They are patch-based ($PatchGAN$) to focus on local image details.

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

        # helper function to define a conv layer
        def block(in_feat, out_feat, norm=True):
            layers = [nn.Conv2d(in_feat, out_feat, 4, stride=2, padding=1)]
            if norm:
                layers.append(nn.InstanceNorm2d(out_feat)) # apply normalization if needed
            layers.append(nn.LeakyReLU(0.2, inplace=True)) # ReLU activation
            return layers

        # the discriminator model as a series of conv layers
        self.model = nn.Sequential(
            *block(in_channels, 64, norm=False), # first layer without normalization (as mentionned on the original paper)
            *block(64, 128),
            *block(128, 256),
            *block(256, 512),
            nn.Conv2d(512, 1, 4, padding=1) # final output layer with single channel output (real or fake)
        )

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

Figure 24: Discriminator Implimentation

4.2 Model initialization:

```python
G_A2B = ResnetGenerator().to(device)
G_B2A = ResnetGenerator().to(device)
D_A = PatchDiscriminator().to(device)
D_B = PatchDiscriminator().to(device)

optimizer_G = optim.Adam(
    list(G_A2B.parameters()) + list(G_B2A.parameters()),
    lr=0.0002,
    betas=(0.5, 0.999)
)

optimizer_D_A = optim.Adam(D_A.parameters(), lr=0.0002, betas=(0.5, 0.999))
optimizer_D_B = optim.Adam(D_B.parameters(), lr=0.0002, betas=(0.5, 0.999))
```

Figure 25: Model initialization

We initialized $CycleGAN$ model components and optimizers as described in the original paper, using:

- G_{A2B}: Generator that transforms images from domain A (Monet paintings) to domain B (photo).
- G_{B2A}: Generator that transforms images from B to A.
- D_A: Discriminator that judges whether an image is a real B image or a fake one generated by G_{B2A}
- D_B: Discriminator that judges whether an image is a real A image or a fake one generated by G_{A2B}

We used *Adam optimizer* to update the weights of the models during training with:

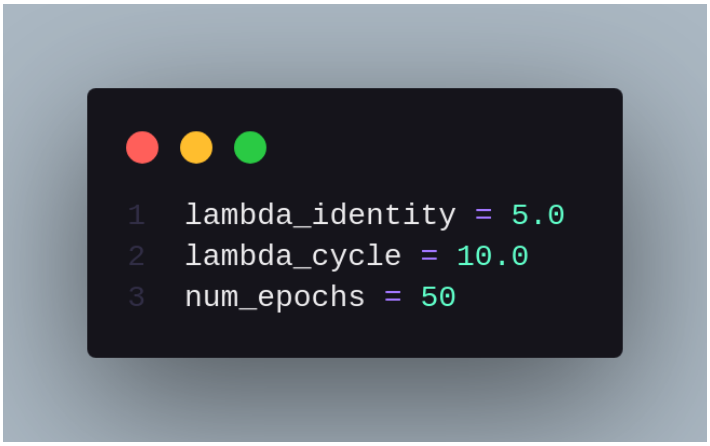


Figure 26: Hyperparameters

4.3 Loss functions:

4.3.1 Adversarial Loss:

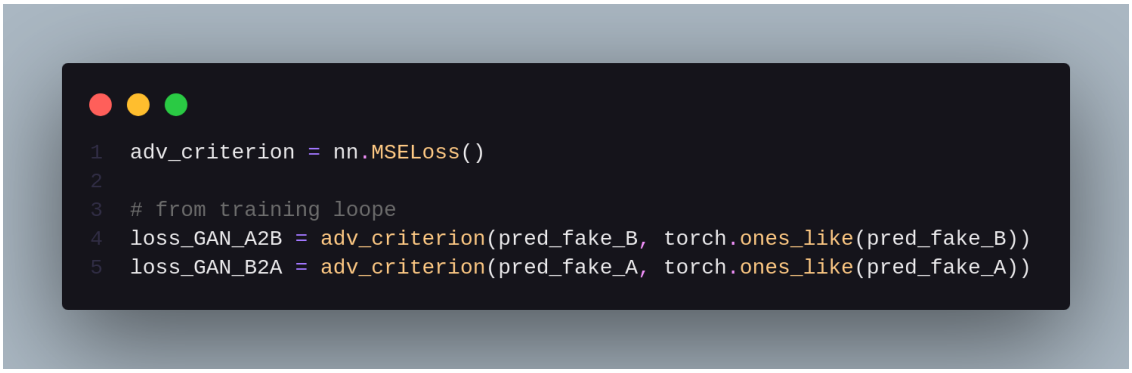
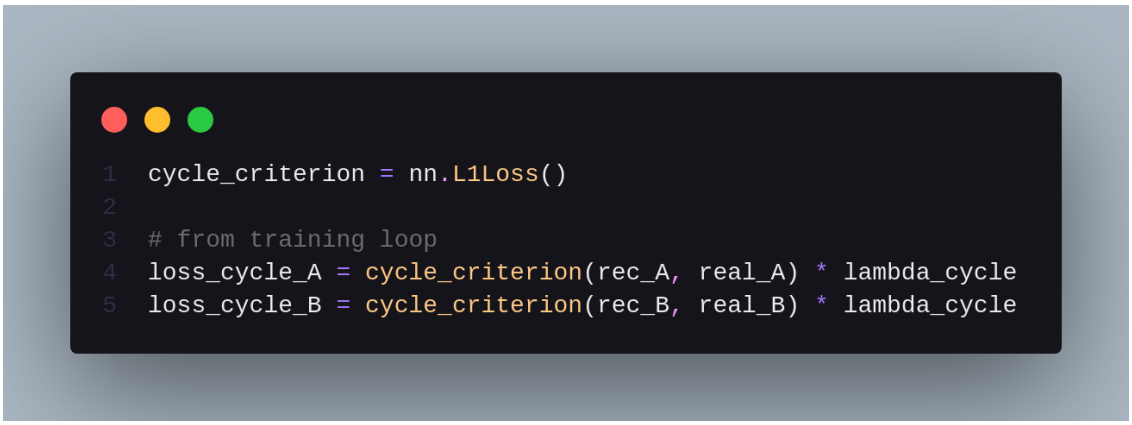


Figure 27: Adversarial Loss

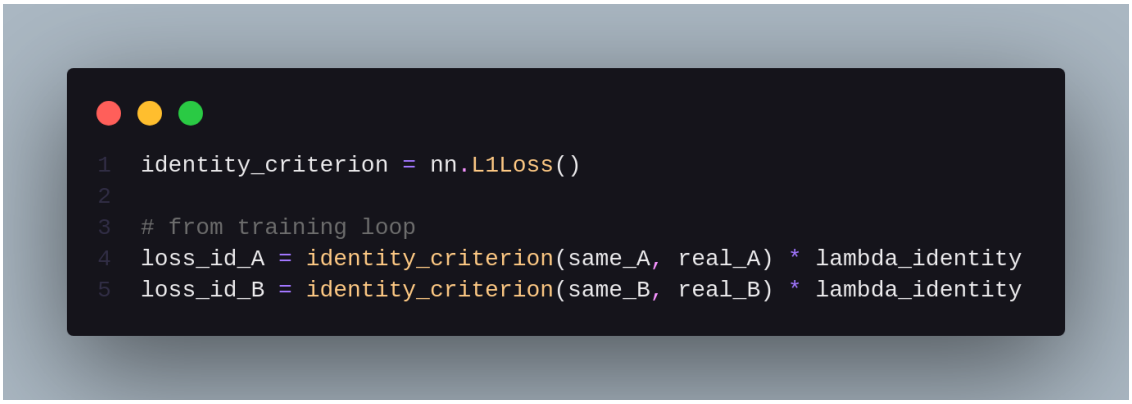
4.3.2 Cycle Consistency Loss:



```
cycle_criterion = nn.L1Loss()

# from training loop
loss_cycle_A = cycle_criterion(rec_A, real_A) * lambda_cycle
loss_cycle_B = cycle_criterion(rec_B, real_B) * lambda_cycle
```

Figure 28: Cycle Consistency Loss

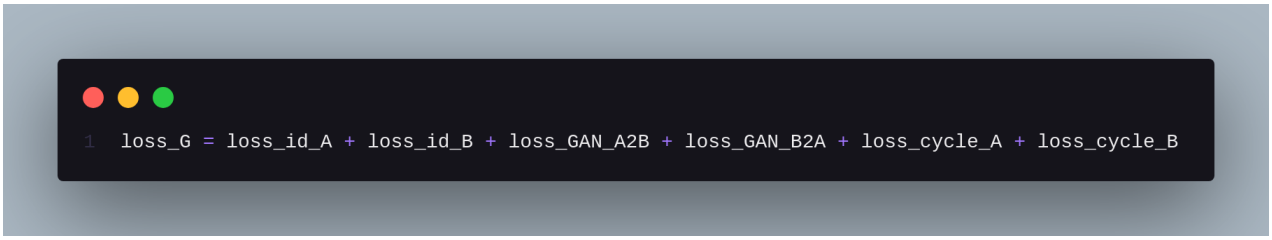
4.3.3 Identity Loss:



```
identity_criterion = nn.L1Loss()

# from training loop
loss_id_A = identity_criterion(same_A, real_A) * lambda_identity
loss_id_B = identity_criterion(same_B, real_B) * lambda_identity
```

Figure 29: Identity Loss

4.3.4 Total generator Loss:



```
loss_G = loss_id_A + loss_id_B + loss_GAN_A2B + loss_GAN_B2A + loss_cycle_A + loss_cycle_B
```

Figure 30: Total generator Loss

4.3.5 Discriminator Loss:

```python
loss_D_real_A = adv_criterion(pred_real_A, torch.ones_like(pred_real_A))
loss_D_fake_A = adv_criterion(pred_fake_A, torch.zeros_like(pred_fake_A))

loss_D_A = (loss_D_real_A + loss_D_fake_A) * 0.5
loss_D_A.backward()

total_loss_D_A += loss_D_A.item()
```

Figure 31: Discriminator Loss

4.4 Training loop:

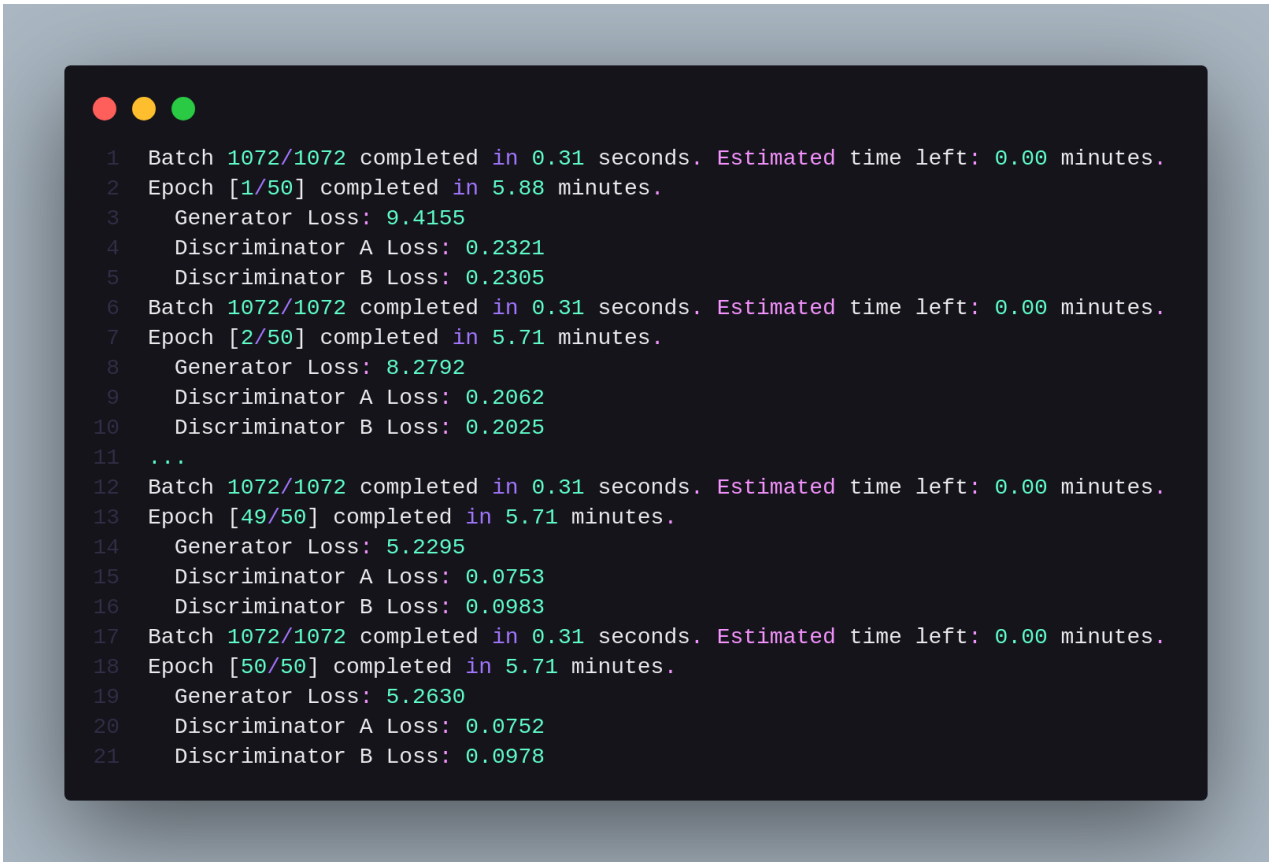
4.4.1 Training loop code:

As mentioned on the paper, they define $num_{epochs} = 200$ but due some reasons (unavailable CPU/GPU like the ones they used) we trained our model for only 50 epochs.

```python
for epoch in range(num_epochs):
    epoch_start_time = time.time() # start time of epoch
    total_loss_G = 0
    total_loss_D_A = 0
    total_loss_D_B = 0
    total_batches = len(dataloader)

    for batch_idx, batch in enumerate(dataloader):
        batch_start_time = time.time()  # start time of batch

        real_A = batch["A"].to(device)
        real_B = batch["B"].to(device)

        # train generators
        optimizer_G.zero_grad()

        # identity loss
        same_B = G_A2B(real_B)
        same_A = G_B2A(real_A)
        loss_id_A = identity_criterion(same_A, real_A) * lambda_identity
        loss_id_B = identity_criterion(same_B, real_B) * lambda_identity

        # adversarial loss
        fake_B = G_A2B(real_A)
        pred_fake_B = D_B(fake_B)
        loss_GAN_A2B = adv_criterion(pred_fake_B, torch.ones_like(pred_fake_B))

        fake_A = G_B2A(real_B)
        pred_fake_A = D_A(fake_A)
        loss_GAN_B2A = adv_criterion(pred_fake_A, torch.ones_like(pred_fake_A))

        # cycle consistency loss
        rec_A = G_B2A(fake_B)
        rec_B = G_A2B(fake_A)
        loss_cycle_A = cycle_criterion(rec_A, real_A) * lambda_cycle
        loss_cycle_B = cycle_criterion(rec_B, real_B) * lambda_cycle

        # total generator loss
        loss_G = loss_id_A + loss_id_B + loss_GAN_A2B + loss_GAN_B2A + loss_cycle_A + loss_cycle_B
        loss_G.backward()
        optimizer_G.step()

        # train discriminators
        optimizer_D_A.zero_grad()
        pred_real_A = D_A(real_A)
        pred_fake_A = D_A(fake_A.detach())
        loss_D_real_A = adv_criterion(pred_real_A, torch.ones_like(pred_real_A))
        loss_D_fake_A = adv_criterion(pred_fake_A, torch.zeros_like(pred_fake_A))
        loss_D_A = (loss_D_real_A + loss_D_fake_A) * 0.5
        loss_D_A.backward()
        optimizer_D_A.step()

        optimizer_D_B.zero_grad()
        pred_real_B = D_B(real_B)
        pred_fake_B = D_B(fake_B.detach())
        loss_D_real_B = adv_criterion(pred_real_B, torch.ones_like(pred_real_B))
        loss_D_fake_B = adv_criterion(pred_fake_B, torch.zeros_like(pred_fake_B))
        loss_D_B = (loss_D_real_B + loss_D_fake_B) * 0.5
        loss_D_B.backward()
        optimizer_D_B.step()

        # accumulate losses
        total_loss_G += loss_G.item()
        total_loss_D_A += loss_D_A.item()
        total_loss_D_B += loss_D_B.item()

        # time running per batch
        batch_time = time.time() - batch_start_time
        remaining_batches = total_batches - (batch_idx + 1)
        estimated_time_left = batch_time * remaining_batches
        print(f"Batch {batch_idx+1}/{total_batches} completed in {batch_time:.2f} seconds. Estimated time left: {estimated_time_left/60:.2f} minutes.", end='\r')

    epoch_end_time = time.time()
    epoch_duration = epoch_end_time - epoch_start_time
    G_losses.append(total_loss_G / total_batches)
    D_A_losses.append(total_loss_D_A / total_batches)
    D_B_losses.append(total_loss_D_B / total_batches)
    epoch_times.append(epoch_duration / 60)

    print(f"\nEpoch [{epoch+1}/{num_epochs}] completed in {epoch_duration/60:.2f} minutes.")
    print(f"  Generator Loss: {total_loss_G/total_batches:.4f}")
    print(f"  Discriminator A Loss: {total_loss_D_A/total_batches:.4f}")
    print(f"  Discriminator B Loss: {total_loss_D_B/total_batches:.4f}")
```

Figure 32: Training loop

4.4.2 Obtained results:



```
Batch 1072/1072 completed in 0.31 seconds. Estimated time left: 0.00 minutes.
Epoch [1/50] completed in 5.88 minutes.
  Generator Loss: 9.4155
  Discriminator A Loss: 0.2321
  Discriminator B Loss: 0.2305
Batch 1072/1072 completed in 0.31 seconds. Estimated time left: 0.00 minutes.
Epoch [2/50] completed in 5.71 minutes.
  Generator Loss: 8.2792
  Discriminator A Loss: 0.2062
  Discriminator B Loss: 0.2025
...
Batch 1072/1072 completed in 0.31 seconds. Estimated time left: 0.00 minutes.
Epoch [49/50] completed in 5.71 minutes.
  Generator Loss: 5.2295
  Discriminator A Loss: 0.0753
  Discriminator B Loss: 0.0983
Batch 1072/1072 completed in 0.31 seconds. Estimated time left: 0.00 minutes.
Epoch [50/50] completed in 5.71 minutes.
  Generator Loss: 5.2630
  Discriminator A Loss: 0.0752
  Discriminator B Loss: 0.0978
```

Figure 33: Obtained results

5 Results:

5.1 Visual Examples on test set:

Figure 34: Visual Examples 1

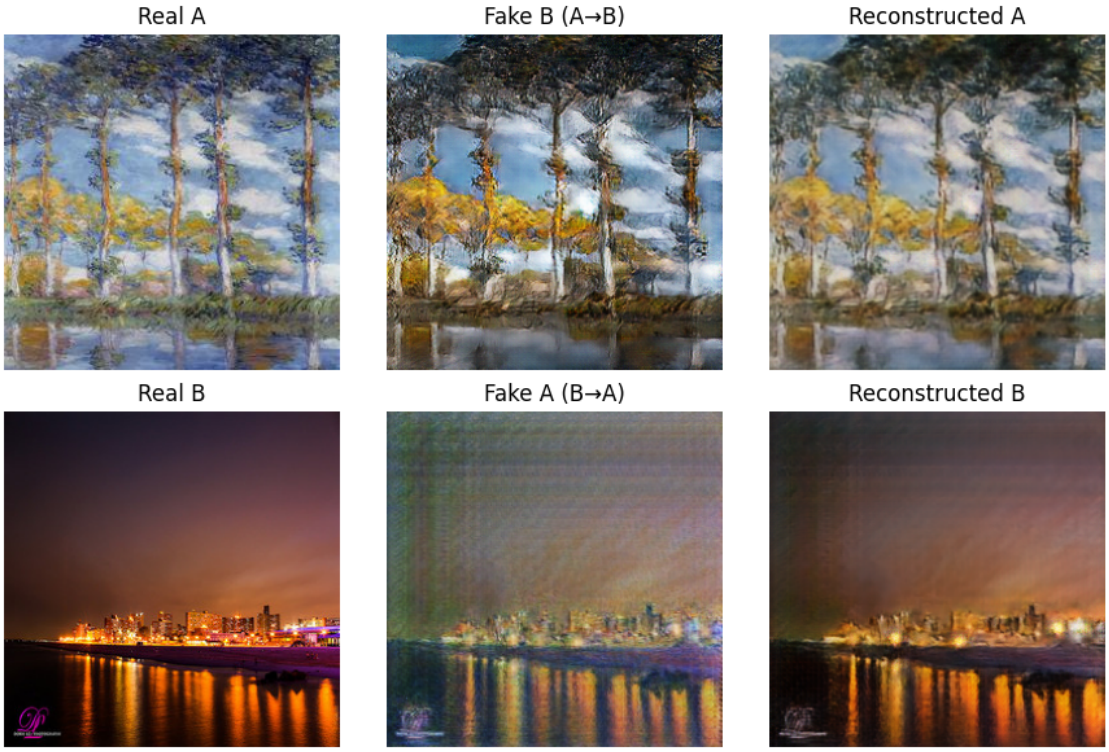


Figure 35: Visual Examples 2

Figure 36: Visual Examples 3

5.2 Training Curves:

5.2.1 Time executing:

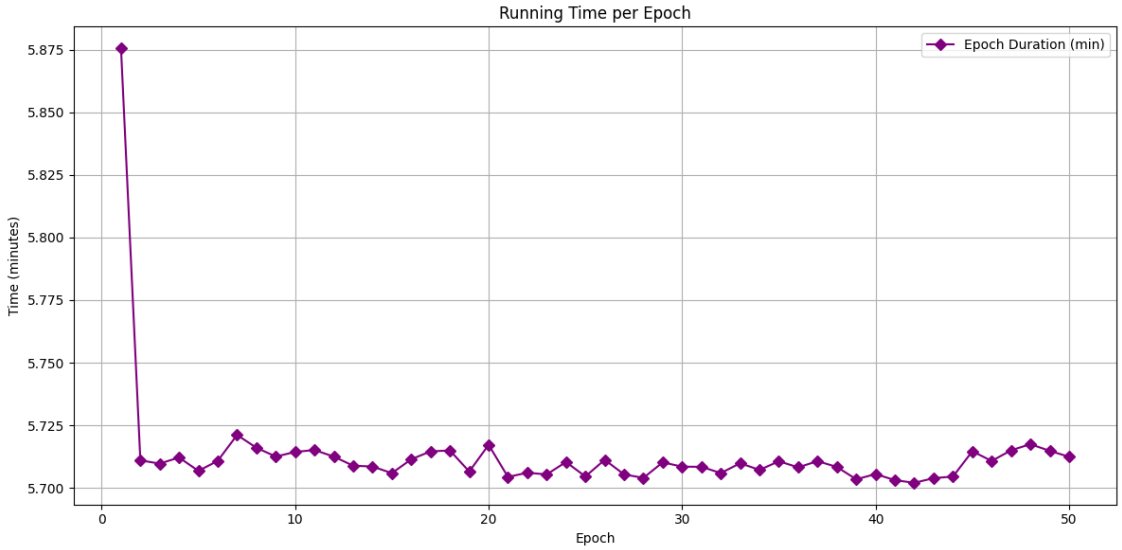


Figure 37: Running time per epoch

With total running time of 285.64min (4h 46min) and average time per epoch 5.7min.

5.2.2 Discriminators loss:

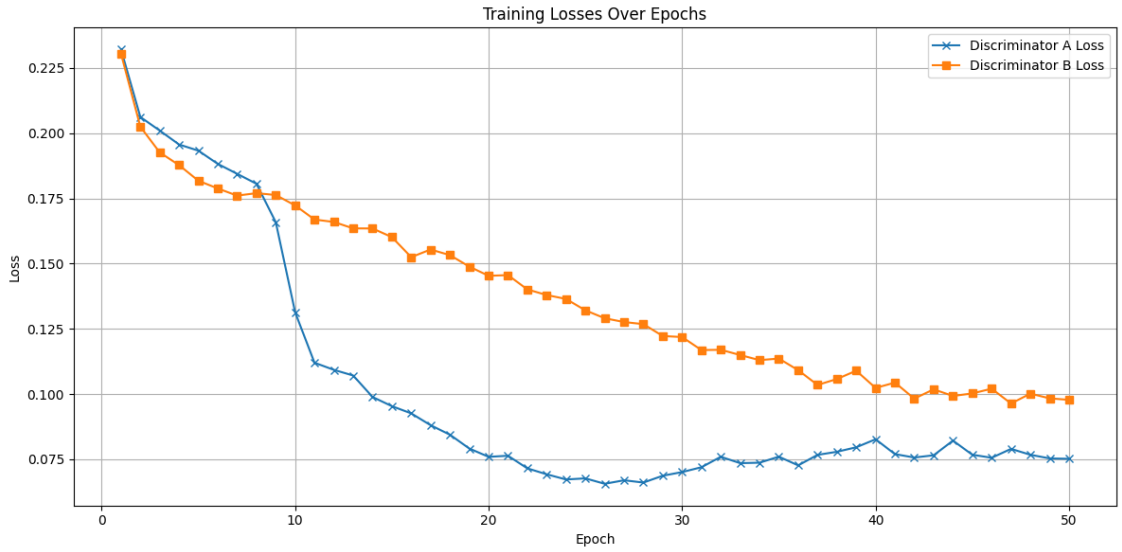


Figure 38: Discriminators loss

5.2.3 Discriminators loss compared to Generators loss:

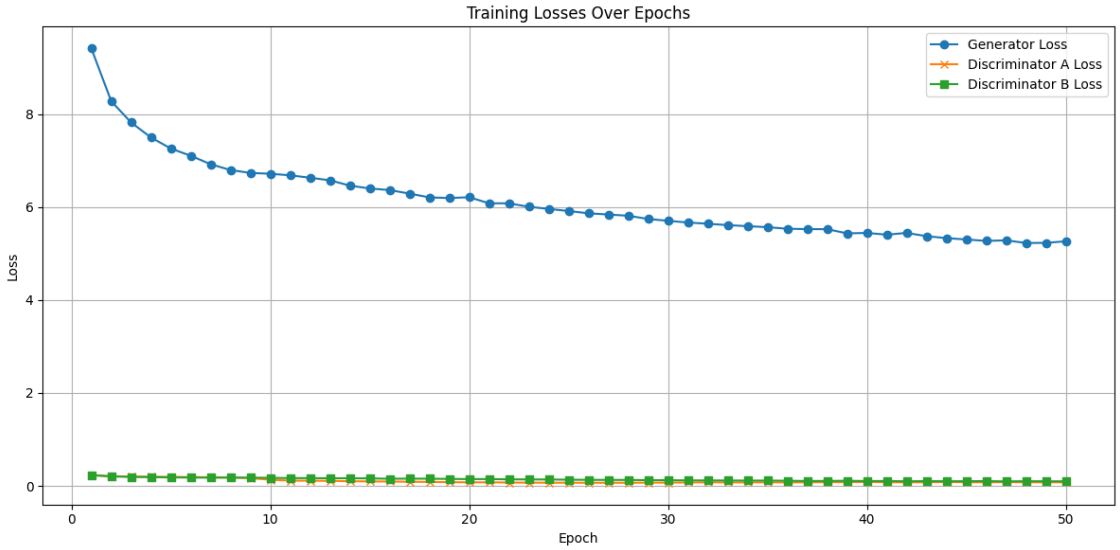


Figure 39: Discriminators loss compared to Generators loss

5.3 Quantitative Evaluation:

5.3.1 Cycle Consistency Loss:

Which measures how close the reconstructed image is to the original input with a range of $[0, \infty[$. We get 0.2965 which shows high fidelity.

5.3.2 LPIPS:

Learned Perceptual Image Patch Similarity compares two images using deep features from a pre-trained network (e.g., AlexNet in our case or VGG), measuring perceptual similarity more aligned with human vision with a range $[0, 1]$ where:

- $[0.0, 0.2]$ refers to good reconstructions
- $[0.2, 0.4]$ is the mediocre
- $[0.4, 1]$ refers to poor reconstructions

We get 0.3765.

In the paper, for evaluating the *CycleGAN* model they used:

- **Qualitative Evaluation:** visually inspecting the generated images and that's what we did.
- **Quantitative Evaluation:** they used:
  - Loss functions: we used this.
  - Inception Score (IS) which measures both the quality and diversity of the generated images. The idea is that high-quality images should be confidently classified by a pre-trained Inception model, and diverse images should cover a wide variety of categories. We didn't use it because

it was designed for unconditional image generation (GANs generating from noise) so we didn't think it's gonna give us effective result.

- Fréchet Inception Distance (FID) which compares the distribution of generated images to real images by using the Inception network's feature representations. A lower FID score indicates better performance, as it shows that the generated images are closer in distribution to real images. We didn't use it because it's requires a large set of test images for reliable results.

6 Discussion and Conclusion:

In this project, we implemented and evaluated a *CycleGAN* based approach for unpaired image-to-image translation. The model was trained to learn a bidirectional mapping between two visual domains (from/to monet painting and real photos) without requiring paired training data. Instead of relying on traditional evaluation metrics like Inception Score (IS) or Fréchet Inception Distance (FID), we focused on loss-based evaluation specifically the cycle consistency loss where we obtained low value (0.29) for the training set. The results demonstrated that our model successfully learned the mapping between the two domains. The generated images retained the content structure of the input while applying style elements from the target domain, which is the expected behavior of *CycleGAN* while the loss curves during training indicated stable convergence without signs of mode collapse or exploding gradients, suggesting that the generators/discriminators dynamics were well balanced.

In general, The visual results for *CyclaGAN* are much better than the results for a normal GAN where artistic strokes are clearly visible in those images produced by *CycleGAN*, the plots of losses show a lower values, maybe different value for the hyperparameters may produce better results but we didn't have sufficient time to try it out.

Github repo where you can find the code:
`https://github.com/amaliahm/neural-style-transfer-with-GAN.git`

List of Equations

7 References:

- The original paper by Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A. Efros, *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks* on `https://arxiv.org/abs/1703.10593`
- Monet2Photo (Monet Paintings and Natural Photos Dataset), available at: `https://www.kaggle.com/datasets/balraj98/monet2photo`
- Medium articles on `https://medium.com/`