CycleGAN - I'm Something of a Painter Myself

a project authored by

Shamprikta Mehreen and Oluchi Ibeneme

In fulfillment of the course

Deep-Learning

7 January, 2021

Abstract

In the following report we use a cycleGAN model to perform a image to image translation task. The report first starts by defining various relevant concepts then we define our experimental setup before summarizing out experiments. In the discussion we provide an interpretation of the results before drawing conclusions.

Contents

Chapter 1

Introduction

In this Chapter we give a detailed description of our problem and go detail about image-to-image translation problems, unpaired image-to-image translation problems along with their existing solutions and which one we chose to proceed with.

1.1 Image-to-Image Translation

One of the classes of graphics and vision problems is image-to-image translation, where the objective is to use a training set containing aligned image pairs and learn the mapping between an input image and an output image. Some examples of image-to-image translation are converting grayscale to colour images [1], satellite images to maps [2], sketches to photographs [3], and paintings to photographs [4]. However, paired training data is not accessible for many tasks. Training a model for image-to-image translation usually requires a large dataset containing paired examples. These datasets can be expensive and difficult to prepare, even impossible in some cases, such as pictures of paintings by old artists.

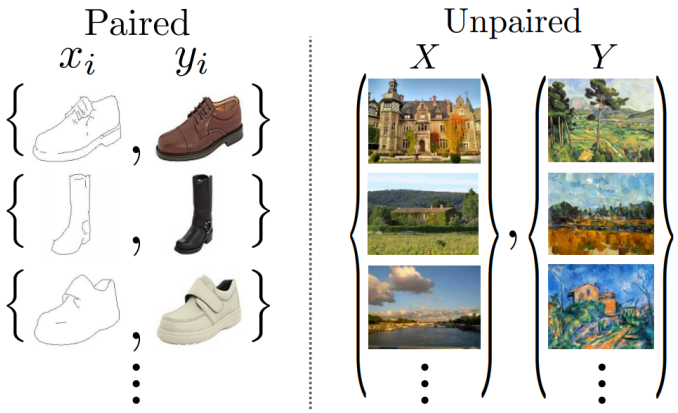


Figure 1.1: (left) **Paired** training examples $\{x_i, y_i\}_{i=1}^{N}$, where the y_i corresponds to each given x_i. (right) **Unpaired** training examples consisting a source set $\{x_i\}_{i=1}^{N} \in X$ and a target set $\{y_j\}_{j=1}^{M} \in Y$, without providing any information about which x_i matches which y_i.

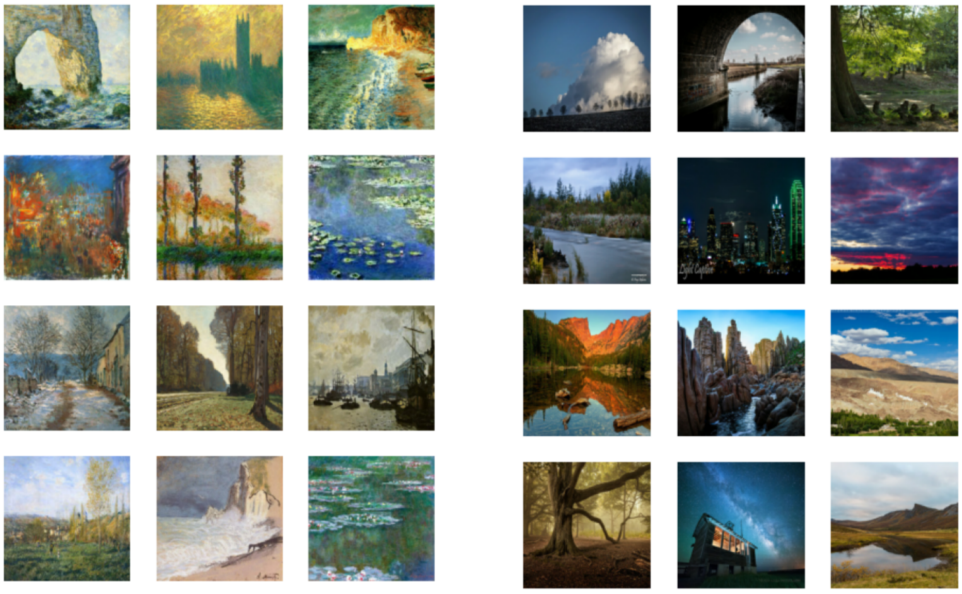
1.2 Unpaired Image-to-image Translation

For unpaired image-to-image translation, several methods have been proposed before, where the objective is to relate two different data domains, X and Y. A Bayesian framework was proposed by [5], which includes a prior based on a batch-based Markov field obtained from the source image and a likelihood term computed from multiple style images. Few years ago, CoupledGANs [6] and cross-modal scene networks [7] was proposed and both used a weight-sharing technique for learning a common representation over the domains. Later Liu et al. [8] extended the CoupledGAN framework by combining variational autoencoders [9] and Generative Adversarial Networks.[10] [11] [12] encouraged the input and output to share certain "content" features even though they may differ in "style", where they also use Adversarial Networks with some additional terms in order to enforce the output to be close to the input in a predefined metric space, such as class label space [12], image pixel space [10], and image feature space [11].

[4] introduced an approach called Cycle-Consistent Adversarial Networks or CycleGAN, which is an unsupervised method that involves the automatic training of image-to-image translation models without paired examples. In our project, we chose to use CycleGAN and it's detailed description and architecture will be discussed in Chapter 2.

1.3 Problem Description and Data

Each Artist has a unique style, be it brush strokes, type of paint or tendencies to use certain colours. Recent improvements with Deep learning have now allowed us to be able to mimic the trademark style of artists. Here in this task, our training data is unpaired. The data cinsists of 300 Monnet style images (Figure 1.2, left) and from which a neural network was to learn to mimic the style in order to apply it to a set of 7028 photographs (Figure 1.2, right) which would have to be transformed into a Monnet style painting. Through this project we aim to create a model which is accurate enough to mimic this style to the extent that its results are practically indistinguishable from that of the original painting.



(a) Monet paintings (b) Photo samples

Figure 1.2: Some examples of Monet paintings and Photo samples

Chapter 2

CycleGAN

In this chapter, we present an overview of Generative modeling, followed by the intuition of Generative Adversarial Networks (GAN) and CycleGAN, before outlining the experiments we carried out in the subsequent chapters.

2.1 Generative Models

Generative Adversarial Networks can be classified as an approach to generative modeling using Deep Learning Methods. Generative modeling is an unsupervised task in machine learning that aims to automatically discover and learn patterns in data in order to generate or produce new examples that could have been drawn from the same distribution or dataset.

2.2 Generative Adversarial Network (GAN)

Generative Adversarial Network [13] is a class of machine learning frameworks used most popularly for the task of generating images. It was developed by Ian Goodfellow along with his colleagues in 2014. The basic idea of a GAN consists of two neural networks that compete against each other (thus adversarial). One network(Generator) tries to fool the second(Discriminator) with images it has generated and the second network attempts to distinguish the real images from the generated ones. For most use cases we consider the network to have converged when the Discriminator reaches a point where it is unable to distinguish between the real images and the fake (generated) ones.

The Generator and the Discriminator both are neural networks. The output of the generator is directly connected to the input of the Discriminator and the Generator updates its weights through Backpropagation after getting a signal from the classification of the Discriminator.

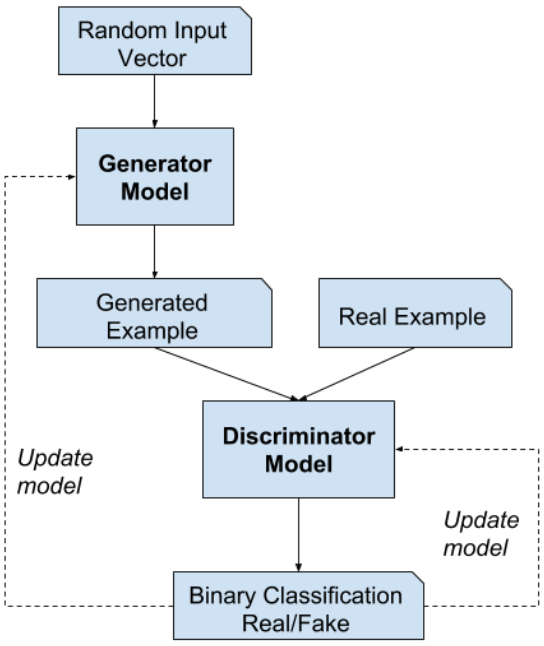


Figure 2.1: Architecture of a Generative Adversarial Network.

2.2.1 The Generator

The Generator inside a GAN learns to produce fake images by including feedback from the Discriminator. It tries to generate image as close as possible to the real image so that the Discriminator classify the generated image as real. During training the Generator includes random input. The GAN takes random noise as its input, the Generator then converts the noise to a meaningful output. The GAN can produce variety of data, sampling from different places in the target distribution by introducing noise.

Usually a neural network is trained by altering it's weights to reduce error or loss of it's output. However, when it comes to GAN, the Generator is not directly connected to the loss that, instead The Generator feeds into the Discriminator network, and the Discriminator produces the output and the generator loss penalizes the Generator for creating an image that the Discriminator network classifies as fake. Afterwards, the loss is calculated from the Discriminator classification and backpropagates through both the

Discriminator and Generator to obtain gradients, which is later used to change only the Generator weights.

2.2.2 The Discriminator

The Discriminator inside a GAN is a simple Classifier which tries to differentiate real image from the fake/generated image created by the Generator. The training data from the Discriminator comes from two different sources, one is the real image is used as positive examples by the Discriminator during training. And another is the generated image from the Generator which is used as negative examples by the Discriminator during training. During the training of the Discriminator, the Generator does not train and it's weight remains constant and it creates examples for the Discriminator to train on.

The Discriminator also connects to two loss functions and during it's training, the Discriminator only uses the Discriminator loss while ignoring ignores the Generator loss. During the training, the Discriminator loss penalizes the Discriminator for misclassifying a real image as fake(generated)incl or a fake(generated) image as real. From the Discriminator loss through the discriminator network, the Discriminator updates its weights through Backpropagation.

With the Generator improving producing better images with training, the Discriminator performance gets worse because the Discriminator cannot easily differentiate between the real image and and fate(generated) image. If the Discriminator accuracy decreases to 50%, then the Generator succeeds perfectly.

2.3 Cycle-Consistent Adversarial Networks (CycleGAN)

One of the successful approaches for unpaired image-to-image translation is CycleGAN. It is an extension of the GAN architecture which involves the simultaneous training of two Generator networks and two Discriminator networks. CycleGAN can capture the characteristics of one image domain and figure out how these characteristics could be translated into another image domain, all in the absence of any paired training examples [14]. In CycleGAN, first, the input images(X) are taken by the first Generator G from the first domain and convert them into the reconstructed images. Then this process is reversed

by converting the reconstructed images to original images using second Generator F. Then the Discriminator models are used to determine how plausible the generated images are, and update the Generator models accordingly.

Similar to all the adversarial networks, CycleGAN also has two parts, Generator and Discriminator. The Generator creates the samples from the desired distribution and the Discriminator tries to identify if the sample is from an actual distribution (real) or generated by the Generator (fake). The main difference between GAN and CycleGAN architecture is the CycleGAN contains two mapping functions G and F, which acts as Generators and their corresponding Discriminators Dx and Dy.

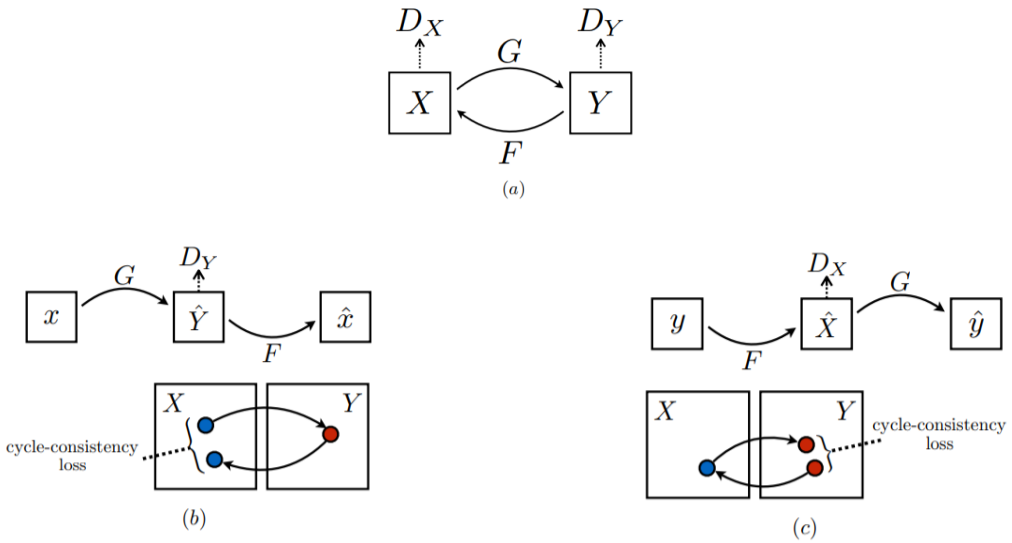


Figure 2.2: [4] (a) The CycleGAN contains two mapping functions $G : X \rightarrow Y$ and $F : Y \rightarrow X$, and associated adversarial Discriminators D_Y and D_X. D_Y encourages G to translate X into outputs indistinguishable from domain Y , and vice versa for D_X, F, and X. To further regularize the mappings, two "cycle consistency losses" are used that capture the intuition that if we translate from one domain to the other and back again we should arrive where we started: (b) forward cycle-consistency loss: $x \rightarrow G(x) \rightarrow F(G(x)) \approx x$, and (c) backward cycle-consistency loss: $y \rightarrow F(y) \rightarrow G(F(y)) \approx y$.

Generator Architecture:

Each Generator in CycleGAN has three parts: Encoder, Transformer and Decoder.

First the input image is passed into the encoder and it extracts the features from the input image using Convolutions and compressing the representation of the image while increasing the number of channels. The encoder has 3 Convolution layers, which reduces the representation by 1/4th of the actual image size i.e, an input image to the encode of

size (256, 256, 3) will be an output of size (64, 64, 256). 2.3 shows the architecture of a CycleGAN Generator.

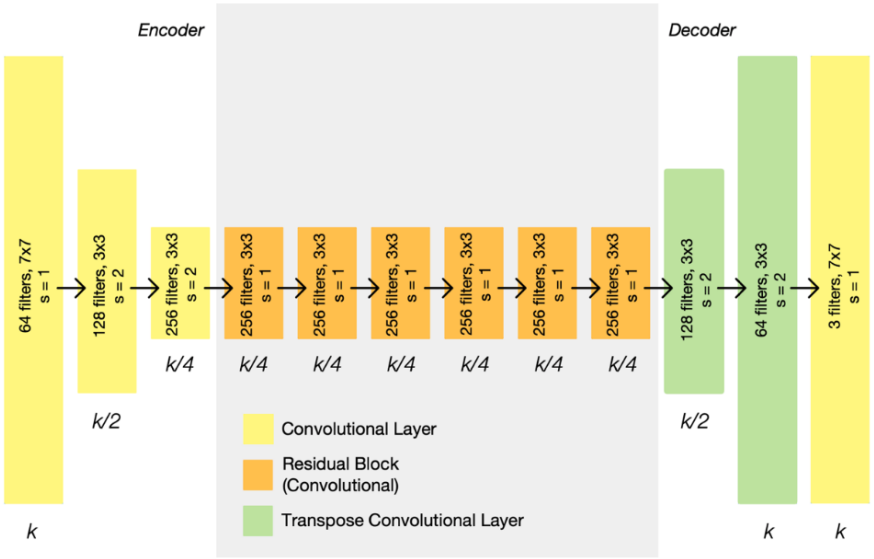


Figure 2.3: Architecture of the Generator of a CycleGAN

Afterwards, the output of the encoder, after activation function is applied, is passed into the transformer. The transformer is composed of 6 or 9 residual blocks based on the size of input. The output of the transformer is then passed into the decoder, which uses a two deconvolution block of fraction strides to expand the size of representation to original size, and an output layer to produce the final image in RGB. Each layer is followed by an instance normalization and a ReLU layer and because the architecture of the Generator is fully convolutional, they can handle arbitrarily large input once trained.

Discriminator Architecture:

In Discriminator Architecture, the PatchGAN Discriminator is used, which is fully convolutional neural networks which look at a "patch" of the input image and output the probability of the patch being "real". This is more computationally efficient and allows the discriminator to focus on more surface-level features, like texture, which is often being changed in an image translation task. In fig 2.4 till the desired output size is reached, the PatchGAN halves the representation size while it doubles the number of channels. For this case, having the PatchGAN evaluate 70x70 sized patches of the input was most effective.

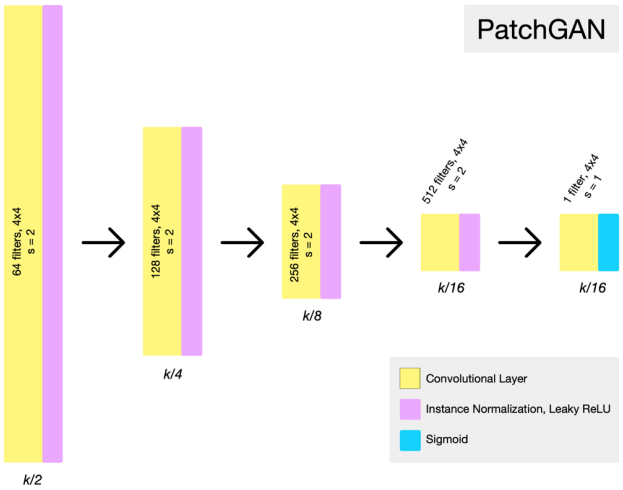


Figure 2.4: Architecture of the Discriminator of a CycleGAN

Cost Functions:

Adversarial loss

Adversarial losses [13] are applied in both mapping functions:

$$Loss_{adversarial}(G, D_Y, X, Y) = \frac{1}{m}(1 - D_Y(G(X)))^2 \tag{2.1}$$

$$Loss_{adversarial}(F, D_X, Y, X) = \frac{1}{m}(1 - D_X(F(Y)))^2 \tag{2.2}$$

Cycle Consistency Loss

With a random set of images, Adversarial networks can map the set of input images to random permutation of images in the output domain, which might make the output distribution similar to target distribution. Hence, adversarial mapping cannot guarantee the input X_i to Y_i, and for this reason [4] proposed that the process should be cycle-consistent. This loss function is used in order to measure the error rate of inverse mapping $G(X) \rightarrow F(G(X))$. The behavior caused by this loss function induce closely matching the real input (x) and F(G(X)).

$$Loss_{cycle}(G, F, X, Y) = \frac{1}{m}[(F(G(X_i)) - X_i) + (G(F(Y_i)) - Y_i)] \tag{2.3}$$

The objective is:

$$\arg\min_{G,F} \max_{D_X, D_Y} L(G, F, D_X, D_Y) \tag{2.4}$$

Chapter 3

Experimentation & Results

In this Section we provide details about the task, dataset provided and the experimental setup. After that we present the results and provide a discussion and conclusion.

The main aim of this project and the Kaggle competition associated with it was to do a style transfer between real photos and monet images. The Goal was to make the real images look as if it was painted by Claude Monet himself. For this project, we chose to use CycleGAN.

3.1 The Architecture

The detailed version of the Architecture can be found in the appendix.

3.2 Evaluation criteria

Apart from making rough estimations on the quality of the generated image based solely on our vision we also uploaded the trained models to kaggle in order to get a MiFID (Memorization-informed Fréchet Inception Distance) score, which is a modification from Fréchet Inception Distance[15] (FID).

In FID, the Inception network is used to extract features from an intermediate layer. Afterwards, the data distribution is modeled for these features by using a multivariate Gaussian distribution with mean μ and covariance $\sum$. The FID between the real images r and generated images g is computed as:

$$FID = \|\mu_r - \mu_r\|^2 + Tr\left(\sum_r + \sum_g -2(\sum_r \sum_g)^{1/2}\right) \tag{3.1}$$

Where T_r is the summation of all the diagonal elements.

MiFID

The memorization distance is basically the minimum cosine distance of all the training samples in the feature space, which are averaged across all the user generated image samples. The threshold of this distance is assigned to 1.0, if the distance exceeds a predefined epsilon.

$$d_{ij} = 1 - \cos(f_{gi}, f_{rj}) = 1 - \frac{f_{gi}.f_{rj}}{|f_{gi}|\,|f_{rj}|} \tag{3.2}$$

where f_g and f_r represent the real or generated images in feature space and f_gi and f_rj respectively represent the $i + th$ and j^{th} vectors of f_g and f_r.

$$d = \frac{1}{N}\sum_i min_j d_{ij} \tag{3.3}$$

defines the minimum distance between a certain generated image (i) across all real images (j), and averaged across all the generated images.

$$d_{thr} = \begin{cases} d, & \text{d } < \epsilon \\ 1, & otherwise \end{cases} \tag{3.4}$$

defines that the threshold of the weight only applies when the (d) is less than a certain empirically determined threshold.

Finally, this memorization term is used in the FID:

$$MiFID = FID.\frac{1}{d_{thr}} \tag{3.5}$$

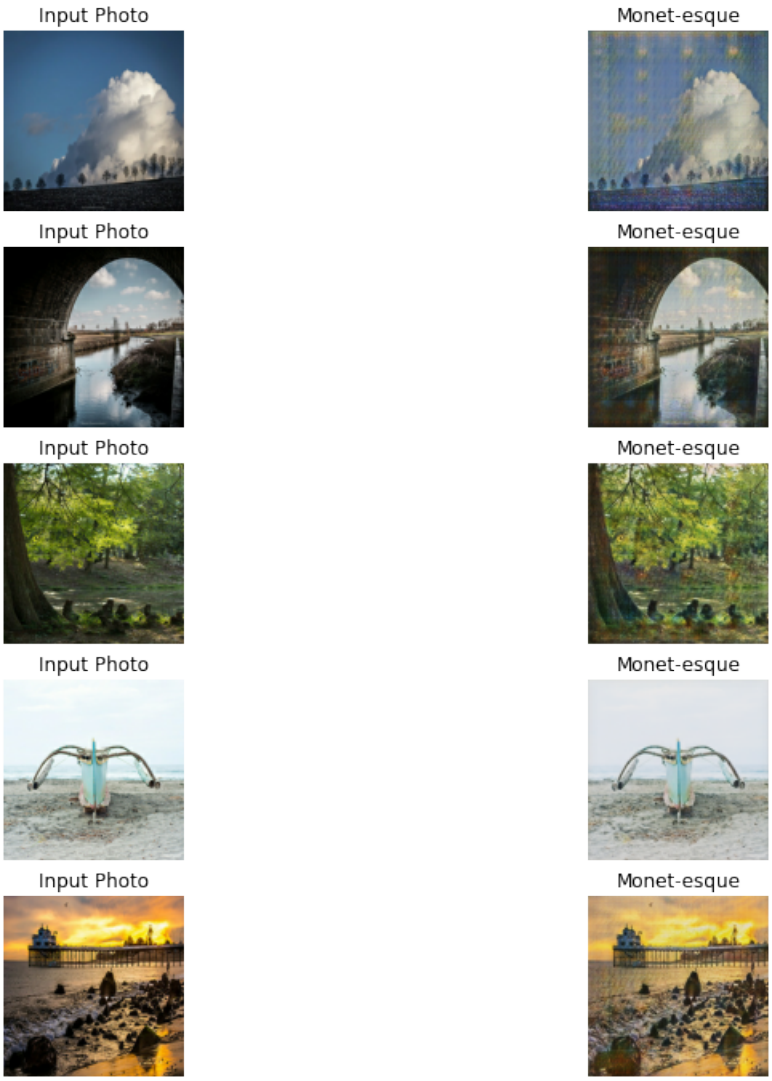
3.3 Results

The hyper-parameters for the base model are in table 3.1 as follows :

Epochs	Loss function	Optimizer	Alpha	Beta
25	Binary cross entropy	Adam (Generator and Discriminator)	2e-4	0.5

Table 3.1: Hyper-parameters for base CycleGAN

with this particular model a MiFID score of 59.7 was obtained over a run time of 3540s. the images generated by this version are seen in image



3.3.1 Effects of time

The first experiment we decided to run with the base version consisted on the effects of run time on the quality of the results. The following table 3.2 consists of the summarized results for these experiments all other factors staying the same.

Epoch	Score	Runtime
30	54	4062.00s
50	51.9	4532.00s
170	45.5	9657.00s
200	43.8	10,9665.00s

Table 3.2: comparison of MiFID scores at different epochs

Below in (Figure 3.1) are sample generated images at each of the above mentioned epochs

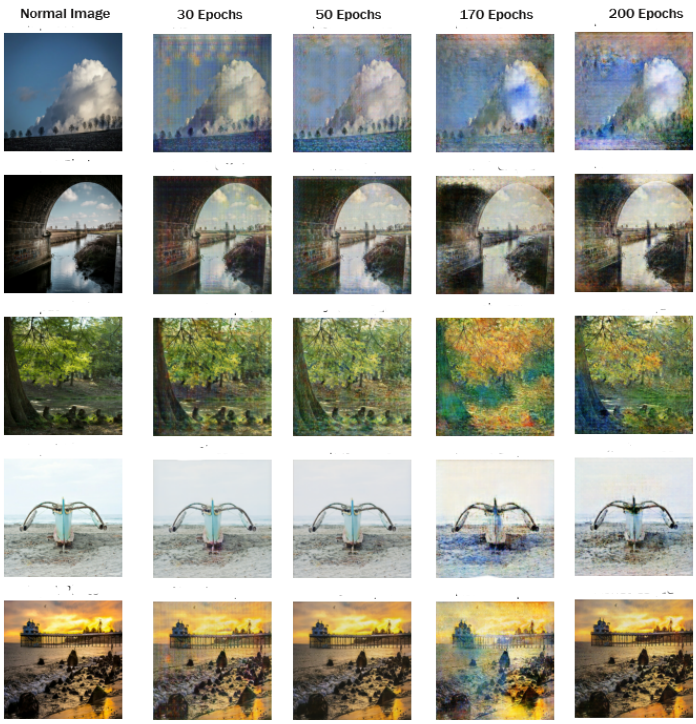


Figure 3.1: comparison of generated images at different epochs.

3.3.2 Effects of Augmentations

For the next set of experiments we explored the effects of augmenting the input Monet images in an effort to learn varying representations of the image.In addition to this, augmentation also helped increase the number of input examples.

The augmentations that were carried out included a combination of flips, crops, rotations and random jitters.The baseline used for comparison in this experiment summarised in table 3.3

Epochs	Loss function	Optimizer	alpha	Beta	score	run time
10	Binary cross entropy	Adam (Generator and Discriminator)	2e-4	0.5	75.9	2898

Table 3.3: Hyper-parameters and performance for base CycleGAN version 2

Using augments the MiFID score of the model decreased to 72.3 with a runtime of 3124 over 10 epochs. After running the model for 30 epochs a MiFID score of 63.4 was obtained.

3.3.3 Repeating and Shuffling

For the next set of experiments we explored the effects of Reusing the input Monet images in an in different orders in an effort to study the effect of increased input to the network.The baseline remains the same as the one in table 3.3, The following results in table 3.4 were obtained with added shuffle and repeat .

Epoch	Score	Runtime
10	69.9	4925.00s
30	52.1	7368.00s

Table 3.4: Comparison of MiFID scores with repeat and shuffle at different epochs

when combining the augments with the shuffle and repeat and running them for 50 epochs we received the following results.

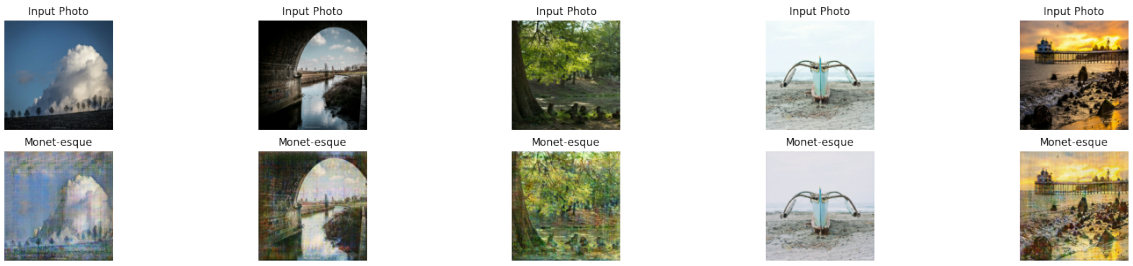


Figure 3.2: comparison of generated images at different epochs.

3.3.4 Addition of Various layers

In the following section we made various changes to the layers of our neural network in order to study how it affects the MiFID score and the quality of the image generated in general. The following table 3.5 summarises the results of the experiments carried out.

Area of change	**Layer**	**Score**	**Runtime**	**Epoch**
Base	No changes	75.83	4925.00s	10
Discriminator	1 additional conv layer with augments	75.06	7368.00s	10
Generator	3 additional conv layers without augments	241.67	2951.00s	10
Discriminator	- 1 conv layer -LeakyRelu alpha =0.2 layer -Flattern - Dense layer with sigmoid function	74.03	3214.00s	10

Table 3.5: Comparison of scores with various additional layers

3.3.5 Changing Losses

In the final Experiment carried out, we studies the effect changing the loss function would have on the performance of the model. The results are summarised in table3.6 below.

Loss Function	**Score**	**Runtime**	**Epoch**
Binary Cross Entropy	75.83	4925.00s	10
Cosine Similarity	75.78872	2991.00s	10
Huber	75.78874	2951.00s	10
KL Divergence	76.59807	2881.00s	10
Mean Squared Logarithmic	75.78874	2691.00s	10

Table 3.6: Comparison of scores Using various losses

Putting it all together, for out final model we compiled together the best results from the previous sections . We were able to achieve a score of 40.1 over 200 epocs. The images generated look as follows

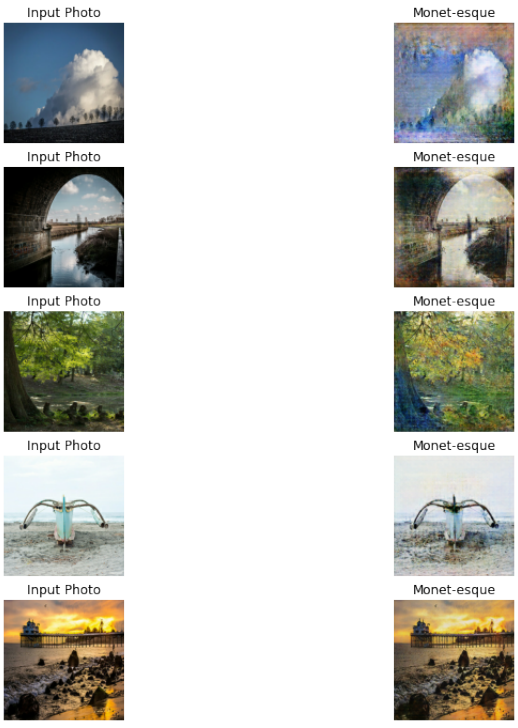


Figure 3.3: Base Architecture of the discriminator

Chapter 4

Discussion & Conclusion

4.1 Discussion

In the Experiments carried out in section[3.3.1] we did a study of the effects of run time on the MiFID score and general outcome of the general outcome of the generated image. Through the results collected in table3.2 we can see that it is evident that a longer training time does indeed have a positive impact on the MiFID score. Through obtaining a 43.8 MiFID score over 200 epochs our team was sharply boosted to well over the 60th position in rank on the leader board out of 200 competitors.

Looking at the results that this model was able to provide us in (Figure:3.3) it is evident that the generator managed to learn the parameters to make a good enough transformation to fool the human eye. Upon further inspection of the generated images, we would argue that the model performed the best at 170 epochs simply because in our opinion the second last picture of the boat looks more realistic in the final result at 200 than at 170. Never the less both images still look good.

Since we had established that time does indeed play a huge factor in the results, the experiments from here on out used a smaller number in order to conserve the run time on kaggles GPU and TPU.

In section[3.3.2] we explored augmenting the data in order to study the effect of having more examples on the dataset. Comparing the results received at 10 epochs for the base and augmented version we can conclude that augmentation does help in the generation of better images. This said the addition of more unique examples would have probably

given better results. In the following section[3.3.3], it was also evident that shuffling and repeating images does have a positive outcome on the results even more so than augmenting the data but not as much as increasing the training time .

In section[3.2.4] we observed that generally adding layers to the generator has no positive effect on the performance of the model. If anything, it decreases the performance. On the other hand we noted an improvement in performance of the model when we increased the number of layers in the discriminator. we can attribute this to the fact that additional layer for learning probably helped the network in picking up patterns in order to better distinguish the inputs. We also noted in some of our other experiments that for best results its important to add an activation layer next maintain performance of the network.

From our observations in the last set of experiments in section[3.2.5] we concluded by noting that changing the loss in the discriminator has no meaningful effect on its performance.

4.2 Conclusion

From our various experiments we conclude that increasing the run time has the most significant impact on improving the performance followed by having more data. We also noticed that it is important to have varying data in different configurations fed into the network. In addition to this, while it is hard to improve the architecture of the CycleGan generally adding convolutional layers to the discriminator may help boost the performance by a small margin. Apart from this we finally conclude that changing the type of loss has no meaningful impact on the model.

References

[1] Richard Zhang et al. "Real-time user-guided image colorization with learned deep priors". In: *arXiv preprint arXiv:1705.02999* (2017).

[2] Phillip Isola et al. "Image-to-image translation with conditional adversarial networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 1125–1134.

[3] Patsorn Sangkloy et al. "Scribbler: Controlling deep image synthesis with sketch and color". In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2017, pp. 5400–5409.

[4] Jun-Yan Zhu et al. "Unpaired image-to-image translation using cycle-consistent adversarial networks". In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 2223–2232.

[5] Rómer Rosales, Kannan Achan, and Brendan J Frey. "Unsupervised image translation." In: *iccv*. 2003, pp. 472–478.

[6] Ming-Yu Liu and Oncel Tuzel. "Coupled generative adversarial networks". In: *Advances in neural information processing systems*. 2016, pp. 469–477.

[7] Yusuf Aytar et al. "Cross-modal scene networks". In: *IEEE transactions on pattern analysis and machine intelligence* 40.10 (2017), pp. 2303–2314.

[8] Ming-Yu Liu, Thomas Breuel, and Jan Kautz. "Unsupervised image-to-image translation networks". In: *Advances in neural information processing systems*. 2017, pp. 700–708.

[9] Diederik P Kingma and Max Welling. "Auto-encoding variational bayes". In: *arXiv preprint arXiv:1312.6114* (2013).

[10] Ashish Shrivastava et al. “Learning from simulated and unsupervised images through adversarial training”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 2107–2116.

[11] Yaniv Taigman, Adam Polyak, and Lior Wolf. “Unsupervised cross-domain image generation”. In: *arXiv preprint arXiv:1611.02200* (2016).

[12] Konstantinos Bousmalis et al. “Unsupervised pixel-level domain adaptation with generative adversarial networks”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 3722–3731.

[13] Ian J. Goodfellow et al. *Generative Adversarial Networks*. 2014. arXiv: `1406.2661 [stat.ML]`.

[14] *CycleGAN*. `https://www.tensorflow.org/tutorials/generative/cyclegan`.

[15] Martin Heusel et al. *GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium*. 2018. arXiv: `1706.08500 [cs.LG]`.

Appendix A

Model Architecture

The Following is a outline of our model. Note that in the Generator all the Layers that are labelled sequential also have a Convolutional and Instance Normalization layer following it .

```
Model: "model_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_image (InputLayer)     [(None, 256, 256, 3)]     0
_________________________________________________________________
sequential_75 (Sequential)   (None, 128, 128, 64)      3072
_________________________________________________________________
sequential_76 (Sequential)   (None, 64, 64, 128)       131328
_________________________________________________________________
sequential_77 (Sequential)   (None, 32, 32, 256)       524800
_________________________________________________________________
zero_padding2d (ZeroPadding2 (None, 34, 34, 256)       0
_________________________________________________________________
conv2d_43 (Conv2D)           (None, 31, 31, 512)       2097152
_________________________________________________________________
instance_normalization_72 (I (None, 31, 31, 512)       1024
_________________________________________________________________
leaky_re_lu_43 (LeakyReLU)   (None, 31, 31, 512)       0
_________________________________________________________________
zero_padding2d_1 (ZeroPaddin (None, 33, 33, 512)       0
_________________________________________________________________
conv2d_44 (Conv2D)           (None, 30, 30, 1)         8193
=================================================================
Total params: 2,765,569
Trainable params: 2,765,569
Non-trainable params: 0
_________________________________________________________________
```

Figure A.1: Final Architecture of the discriminator

```
Model: "functional_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_image (InputLayer)     [(None, 256, 256, 3)]     0
_________________________________________________________________
sequential (Sequential)      (None, 128, 128, 64)      3072
_________________________________________________________________
sequential_1 (Sequential)    (None, 64, 64, 128)       131328
_________________________________________________________________
sequential_2 (Sequential)    (None, 32, 32, 256)       524800
_________________________________________________________________
zero_padding2d (ZeroPadding2 (None, 34, 34, 256)       0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 31, 31, 512)       2097152
_________________________________________________________________
conv2d_4 (Conv2D)            (None, 14, 14, 512)       4194304
_________________________________________________________________
instance_normalization_2 (In (None, 14, 14, 512)       1024
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 14, 14, 512)       0
_________________________________________________________________
zero_padding2d_1 (ZeroPaddin (None, 16, 16, 512)       0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 13, 13, 1)         8193
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 13, 13, 1)         0
_________________________________________________________________
flatten (Flatten)            (None, 169)               0
_________________________________________________________________
dense (Dense)                (None, 1)                 170
=================================================================
Total params: 6,960,043
Trainable params: 6,960,043
Non-trainable params: 0
_________________________________________________________________
```

Figure A.2: Final Architecture of the discriminator

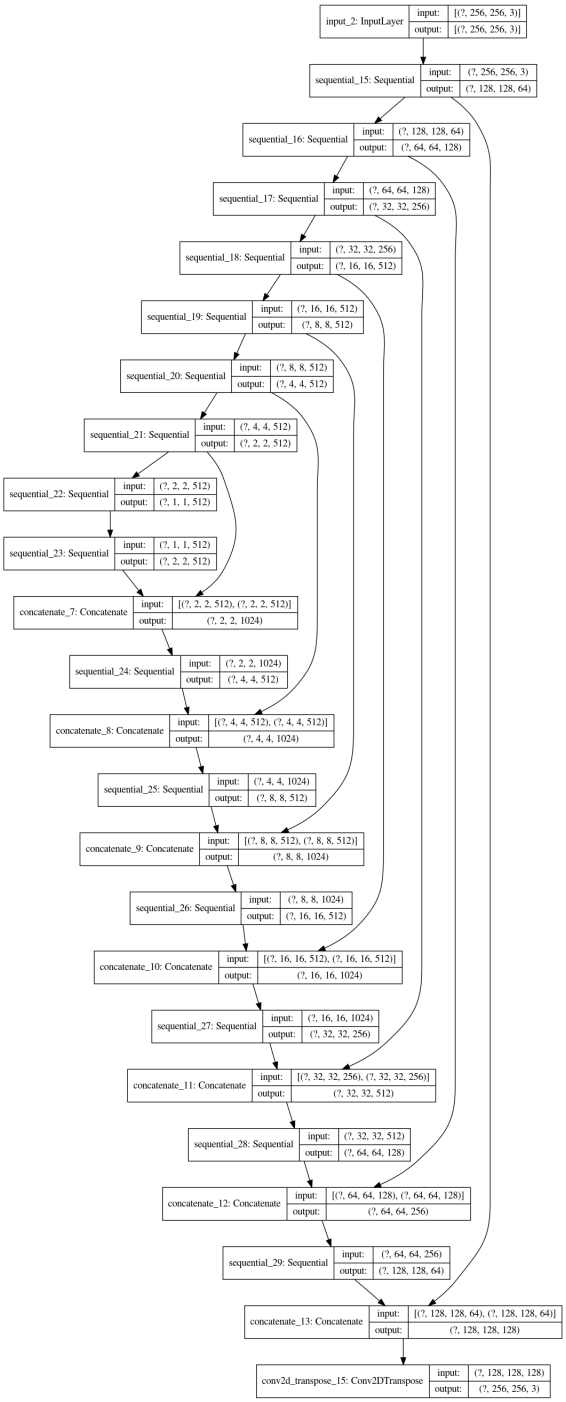


Figure A.3: Final Architecture of the discriminator

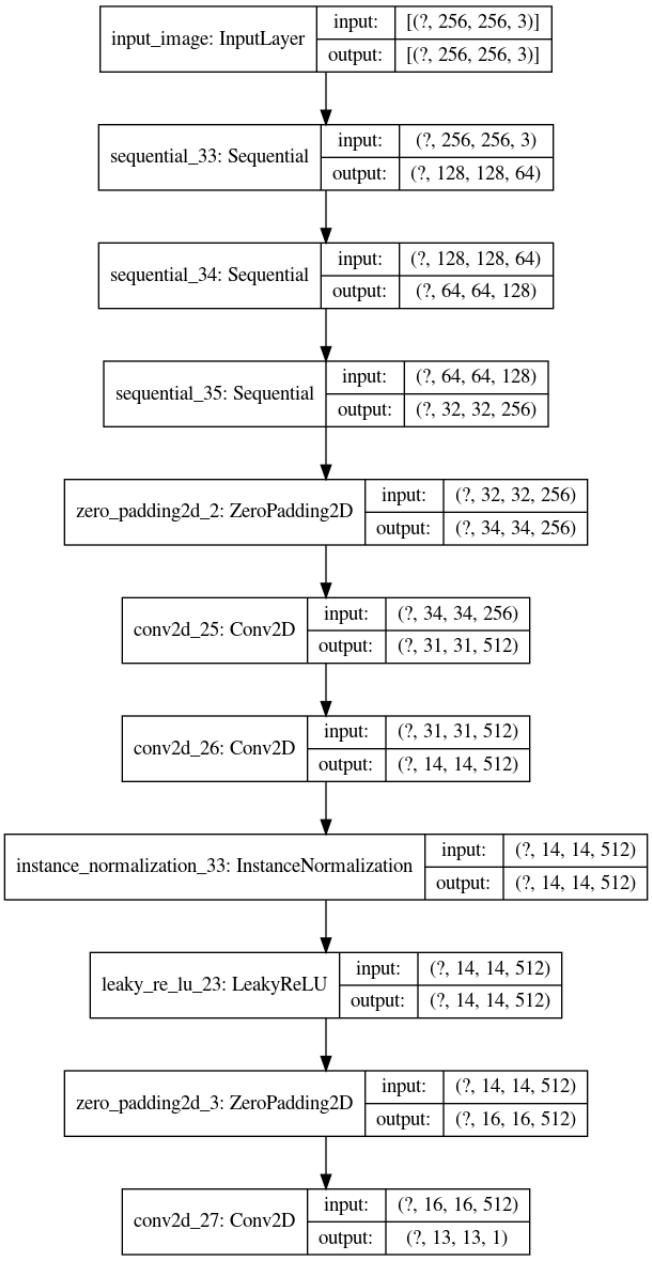


Figure A.4: Base Architecture of the discriminator