Coding Art with DCGANs

Pérez Mariana, ISC A01206747, Tecnológico de Monterrey Campus Querétaro

***Abstract*— This document presents the implementation and explanation of an analysis the use of Deep Convolutional Generative Adversarial Networks and its performance with different datasets of paintings.**

I. INTRODUCTION

Art has always been a very important subject for society. It can show the daily lives of people at a certain time, the feelings of the artist, and the aesthetics behind the artwork. From the Palaeolithic cave drawings in Lascaux, France to the early tools and pottery of native people, to the great Egyptian pyramids; art has always been interwind the existence of humankind. It is a manifestation of what a person believes and thinks, in a form which can be shown to others, to enrich them. It has been, for centuries, a way of manipulating the masses of human conscience itself. Technology is redefining traditional art in new ways. Works are created by people moving through laser beams or from data gathered on air pollution, and in this paper, art is created by an artificial neural network that learns from a specific artist, a specific art movement and a specific matter subject.

II. STATE OF THE ART

Generative Adversarial Networks or GANs were introduced by Ian Goodfellow in a 2014 paper. This type of architecture is set up to utilize the strength of deep discriminative models to bypass the necessity of maximum likelihood estimation and therefore avoid the main failings of traditional generative models.[1] This is done by using a generative and a discriminative model that compete with each other in order to train. The model therefore contains two neural networks, the generator which generates new data instances, and the discriminator. Generative Adversarial Networks have demonstrated their potential on various tasks such as image generation, image super resolution, 3D object generation, and video prediction. The objective is to train the generative model, which is a parametrized function, that maps noise samples to samples whose distribution is close to that of the real data. The basic scheme of the GAN training procedure is to train a discriminator which assigns higher probabilities to real data samples and lower probabilities to generated data samples, while simultaneously trying to move the generated samples towards the real data manifold using the gradient information provided by the discriminator.[2] Deep Convolution Generative Adversarial Networks also called DCGAN is one of the most popular and successful network designs for GAN. It is mainly composed of convolutional layers without max pooling or fully connected networks. It uses convolutional stride and transposed convolution for the down sampling and the up sampling.[3] Figure 1 shows the design of a generator

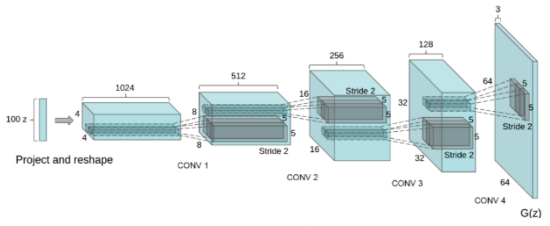


Figure 1. Graphic representation of the network design for the generator

III. DATA SET

The dataset comes from WikiArt[4], that contains a database of more than 20,000 paintings, which are separated in different categories such as style, genres, and media.

The dataset used for this project were the all the paintings from Van Gogh, all the artworks from the Expressionism art movement and all the paintings from the self-portrait's category. The difference in the datasets was to see the performance of the model using a specific artist, art movement and subject. The Van Gogh dataset contained 877 images, the expressionism dataset contained 3,560 images and the portraits dataset contained 2,141 images. All the artworks consisted of different types of techniques, there can be found some sketches, oil paintings, and watercolour paintings.

Figure 2 shows an example of the data from the Van Gogh and self-portraits dataset.

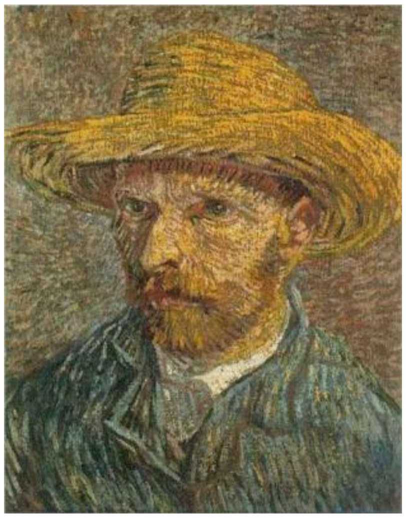
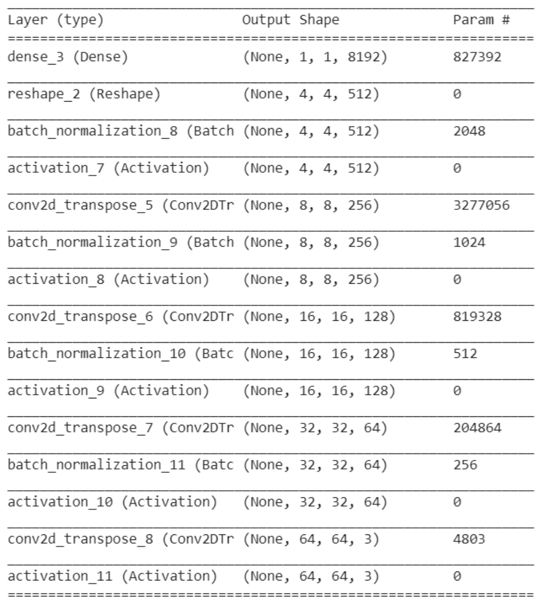


Figure 2. Example of the type of images that are present in the different datasets.

IV. MODEL PROPOSAL

The model consists of two networks, the generator and the discriminator. The generator architecture consists of dense, reshape, batch normalization, 2D convolution transpose and activation layers. Figure 3 shows the complete model of the generator.



```
Layer (type)                 Output Shape              Param #
=================================================================
dense_3 (Dense)              (None, 1, 1, 8192)        827392
_________________________________________________________________
reshape_2 (Reshape)          (None, 4, 4, 512)         0
_________________________________________________________________
batch_normalization_8 (Batch (None, 4, 4, 512)         2048
_________________________________________________________________
activation_7 (Activation)    (None, 4, 4, 512)         0
_________________________________________________________________
conv2d_transpose_5 (Conv2DTr (None, 8, 8, 256)         3277056
_________________________________________________________________
batch_normalization_9 (Batch (None, 8, 8, 256)         1024
_________________________________________________________________
activation_8 (Activation)    (None, 8, 8, 256)         0
_________________________________________________________________
conv2d_transpose_6 (Conv2DTr (None, 16, 16, 128)       819328
_________________________________________________________________
batch_normalization_10 (Batc (None, 16, 16, 128)       512
_________________________________________________________________
activation_9 (Activation)    (None, 16, 16, 128)       0
_________________________________________________________________
conv2d_transpose_7 (Conv2DTr (None, 32, 32, 64)        204864
_________________________________________________________________
batch_normalization_11 (Batc (None, 32, 32, 64)        256
_________________________________________________________________
activation_10 (Activation)   (None, 32, 32, 64)        0
_________________________________________________________________
conv2d_transpose_8 (Conv2DTr (None, 64, 64, 3)         4803
_________________________________________________________________
activation_11 (Activation)   (None, 64, 64, 3)         0
=================================================================
```

Figure 3. Model of the generator

The generator uses nonlinear activation functions such as ReLU, and tanh, because nonlinearity helps to generalize or adapt with a variety of data and to differentiate between the output as in opposed to linear functions where the output of the function will not be confined between any range [5]. The tanh or hyperbolic tangent activation function is a non-linear function similar to the logistic sigmoid function. The range of the tanh function is from -1 to 1. The advantage is that the negative inputs will be mapped strongly negative and the zero inputs will be mapped near zero in the graph. Figure 4 shows the difference between the sigmoid function and the logistic sigmoid function.

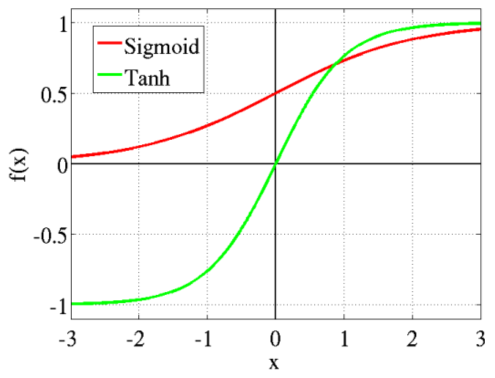


Figure 4. Tanh v/s Logistic Sigmoid

Meanwhile the ReLU or Rectified Linear Unit activation function is a half rectified from the bottom, although the name can be interpreted as if it's a linear function, it is in reality a non-linear function because all the negative values become zero immediately which decreases the ability of the model to fit or train from the data properly. That means any negative input given to the ReLU activation function turns the value into zero immediately in the graph, which in turns affects the resulting graph by not mapping the negative values appropriately. Figure 5 shows the difference between the logistic sigmoid function and the ReLU function.

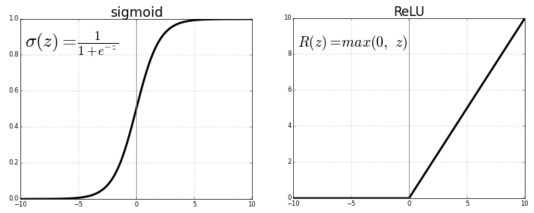
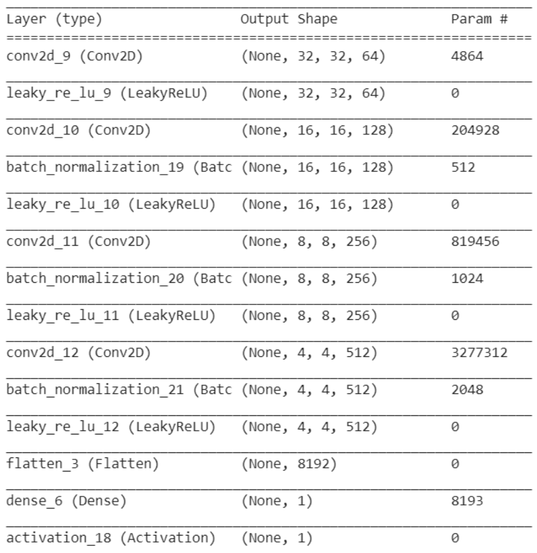


Figure 5. ReLU v/s Logistic Sigmoid.

The generator also uses the Adam optimizer, which is an extension to the stochastic gradient descent that updates the networks weights iterative based on the training data. The benefit of using Adam as opposed to the classical gradient descent is that it doesn't maintain a single learning rate for all weight updates and the learning rate does not change during training. A learning rate is maintained for each network weight and separately adapted as learning unfolds. [6] The learning rate used in this model was an alpha of 0.00015 which can be translated as the size of the steps used to reach a local minimum. The use of a small learning rate is to avoid divergence even though the learning during training will be slow.

The discriminator on the other hand consists of 2D convolutional, batch normalization, flatten, dense and activation layers. Figure 6 shows the complete setup of the discriminator.



```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_9 (Conv2D)            (None, 32, 32, 64)        4864
_________________________________________________________________
leaky_re_lu_9 (LeakyReLU)    (None, 32, 32, 64)        0
_________________________________________________________________
conv2d_10 (Conv2D)           (None, 16, 16, 128)       204928
_________________________________________________________________
batch_normalization_19 (Batc (None, 16, 16, 128)       512
_________________________________________________________________
leaky_re_lu_10 (LeakyReLU)   (None, 16, 16, 128)       0
_________________________________________________________________
conv2d_11 (Conv2D)           (None, 8, 8, 256)         819456
_________________________________________________________________
batch_normalization_20 (Batc (None, 8, 8, 256)         1024
_________________________________________________________________
leaky_re_lu_11 (LeakyReLU)   (None, 8, 8, 256)         0
_________________________________________________________________
conv2d_12 (Conv2D)           (None, 4, 4, 512)         3277312
_________________________________________________________________
batch_normalization_21 (Batc (None, 4, 4, 512)         2048
_________________________________________________________________
leaky_re_lu_12 (LeakyReLU)   (None, 4, 4, 512)         0
_________________________________________________________________
flatten_3 (Flatten)          (None, 8192)              0
_________________________________________________________________
dense_6 (Dense)              (None, 1)                 8193
_________________________________________________________________
activation_18 (Activation)   (None, 1)                 0
```

Figure 6. Model of the discriminator

The discriminator uses the LeakyReLU and sigmoid as activation functions. LeakyReLU is a variation of the ReLU function which increases the range of the ReLU function. LeakyReLU creates a "leak" that helps to solve the dying ReLU problem, in which a "dead" ReLU outputs the same value (zero) for any input. This "leak" helps the model to get a gradient that helps the model to reach a local minumum. Figure 7 shows the difference between ReLU and LeakyRelu.

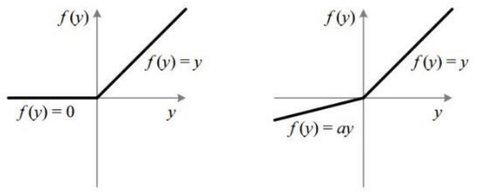


Figure 7. ReLU v/s Leaky ReLU

The discriminator also uses Adam as an optimizer with a slightly bigger learning rate, which is of 0.002. The difference between the learning rates is because from early on the discriminator can easily distinguish real and fake images, therefore the learning rate can be bigger because it is harder for the discriminator to diverge than the generator.

Both the generator and the discriminator kernels are of 3x3. The kernels are used to sharpen, blur and accomplish edge detection of an image, all of this with the intention to extract the features of the image, this means that the kernel's main goal is to determine the most important portions of the input image. The kernel used in the model is initialized with the the glorot uniform initializer, also called Xavier uniform initializer that draws samples from a uniform distribution within a certain limit. The size of the kernel can affect negatively the model because a bigger kernel can make the model lose important details. Figure 8 shows an example of the activation maps done by the discriminator using a kernel of 3x3 after a convolutional layer and a layer with ReLU as an activation function.

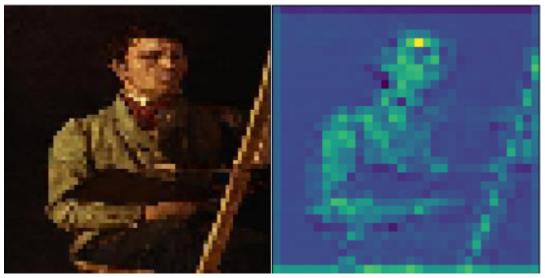


Figure 8. Comparison between original image before and after passing a convolution with ReLU as an activation function.

Dimensionality reduction is used as a form of preprocessing the images that will be used as an input. This is done because according to the curse of dimensionality [7], the number of features is directly proportional to the data needed to train the model. This means that the more features that need to be processed, then the more data is needed to train the model. Dimensionality reduction of the image was performed because of the limited amount of paintings in the datasets, and the limited computational power available. In order to reduce the dimensionality of the image the image is compressed into a vector which works as a compressed representation of the original image and is then passed to the model. This allows the model to go from processing on average 162,240 parameters per picture to 4,864 parameters per picture. Figure 9 shows an example of the resulting image before and after dimensionality reduction.

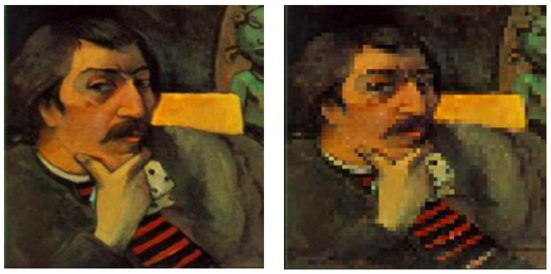


Figure 9. Comparison between the original image and the image after reducing dimensionalities.

With the resulting images the model selects a batch that consist of 64 images, this means that only 64 images will be selected at random to be used as training samples. The model normalizes all the given images to values between -1 and 1. Then a number of noise vectors equal to the batch size are generated. The noise vectors are used by the generator to create a batch of generated fake images.

V. TEST AND VALIDATION

The first test that was done using the Van Gogh dataset which consisted of 877 images. The resulting images had some issues like looking very pixelated and having shadow acne. Figure 10 shows the generated images after 175 epochs.

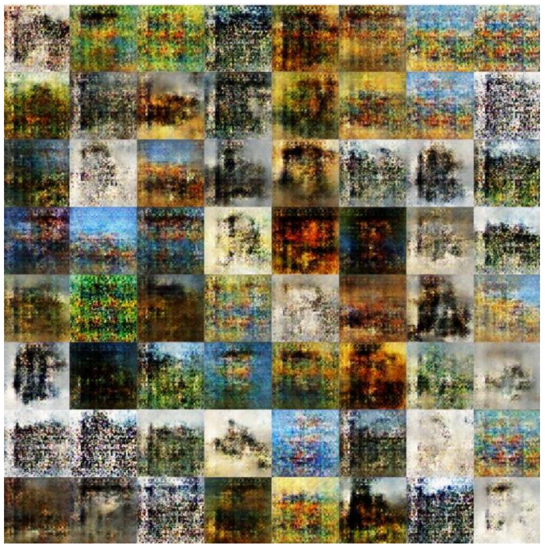


Figure 10. Resulting images from the Van Gogh Dataset

The graph that measured the loss of the discriminator (green) and the generator (blue) confirms that the generator was not able to improve its results, this due to vanishing gradients in generator as consequence of the discriminator becoming optimal [8]. This means that the discriminator reached its optimal point before the generator could reach it, consequently the generator was not able to learn from the discriminator because the discriminator was not learning any more. Figure 10 shows the corresponding graph.

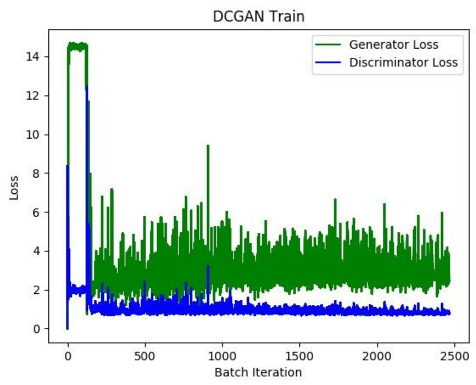


Figure 10. Graphic representation of the generator and discriminator loss over 175 epochs of 44 batch iterations.

The graph shows that the generator after the batch iteration number 1,500 was not able to learn more, so its loss was always almost the same, between 4 and 2. To try to solve this problem, data augmentation was implemented. The problem was that the only transformation that did not affect the model negatively was the horizontal flip, meaning that the augmentation done to the dataset was very small. Figure 11 shows the resulting images after 175 epochs using horizontal flip to augment the images in the dataset.

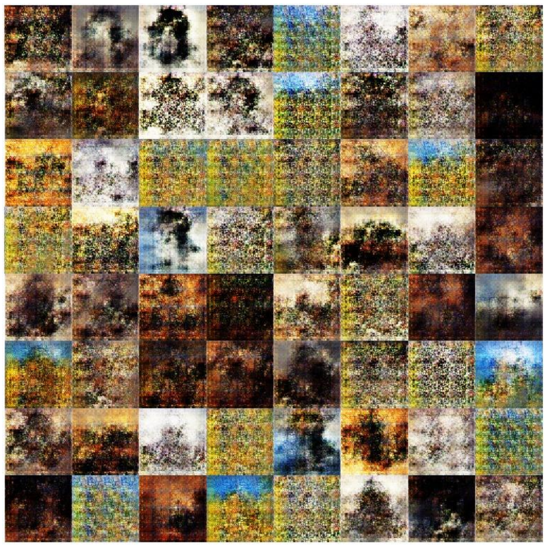


Figure 12. Resulting images from the Van Gogh dataset using horizontal flip after 175 epochs

The resulting images doesn't show a big improvement in the model, but Figure 12, which shows the graph of this iteration shows a slightly improvement where in some iterations it was able to have a loss smaller than 2.

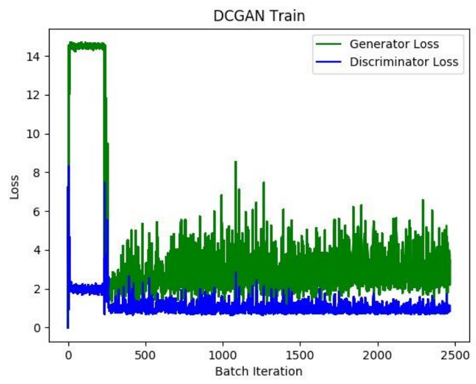


Figure 12. Graphic representation of the generator and the discriminator loss over 175 epochs of 44 batch iterations using data augmentation.

This showed that one way to improve the model was with more data, therefore the next test was done using 3,560 images of the art movement called Expressionism. Figure 13 shows the resulting images after 175 epochs. The resulting images look very good in comparison with the Van Gogh ones.

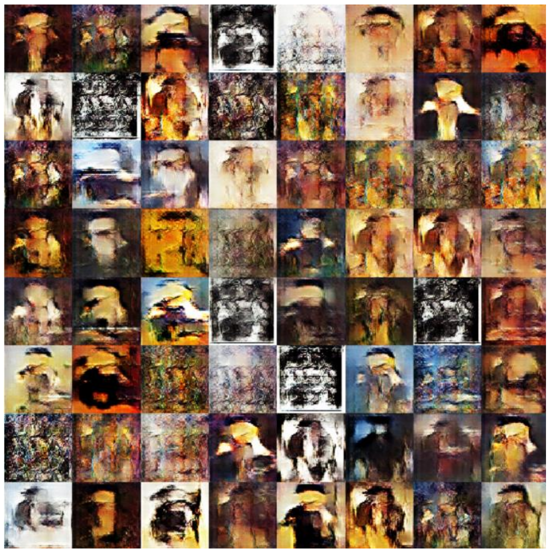


Figure 13. Resulting images based on the expressionism movement.

Figure 14 shows a comparison between original images and the resulting images.

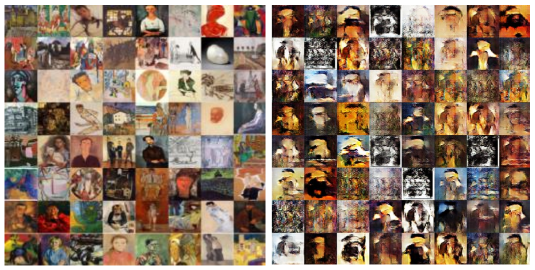


Figure 14. Comparison between original v/s generated

The resulting images can show a pattern in the colors used where it tends to be ochres and browns as the ones used by expressionist artists. Figure 15 shows the graph of this test.

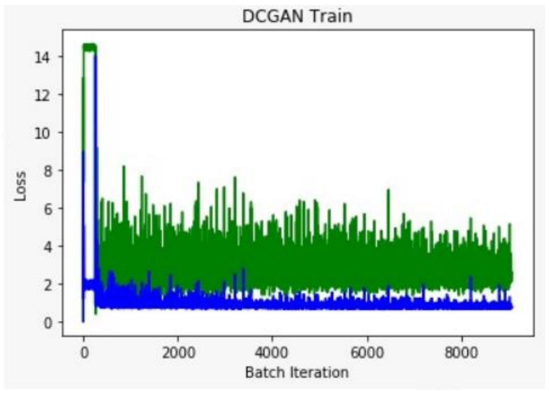


Figure 15. Graphic representation of the generator and the discriminator loss over 175 epochs of 55 batch iterations using the expressionism dataset.

It is possible to see an improvement in the convergence of the generator where the generator loss decreases up to 2 in almost all iterations, this explains the better results in the resulting images. The next test was using the same expressionism dataset but applying a regularizer to the model that allowed penalties on the layer parameters during the optimization. The implemented regularizer was a kernel regularizer which regularizes the function applied to the kernel weights matrix. The penalty added was a penalty of 1x10-7 in the 2D convolutional layers. Figure 16 shows the resulting images.

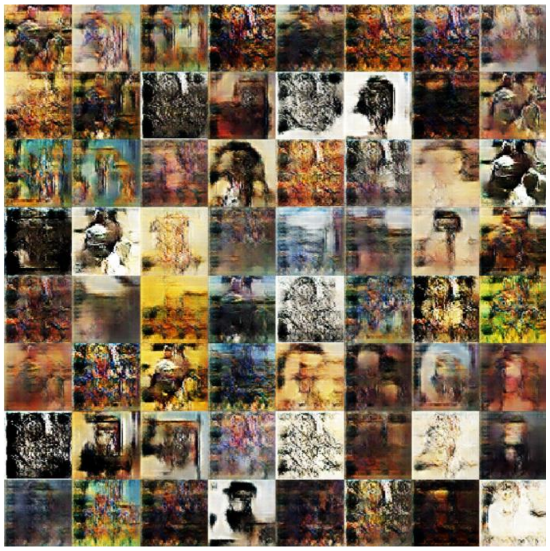


Figure 16. Resulting images based on the expressionism movement after applying regularization to the model

Applying regularization to the model helped to diverse the resulting images and improving the color performance based on the original artworks. Figure 17 shows the graph of the generator and discriminator loss.

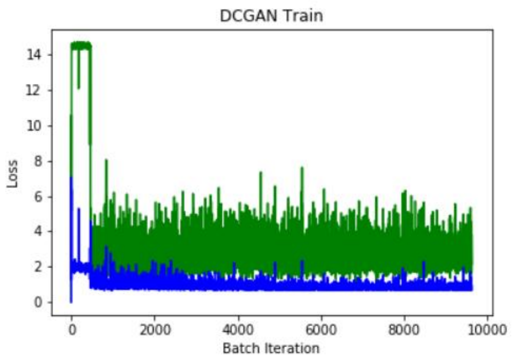


Figure 17. Graphic representation of the generator and discriminator loss over 175 epochs of 55 batch iterations using the expressionism dataset and applying regularization.

The graph shows that the error in the generator decreased and was able to have more iterations where the total loss was smaller than 2, as in opposed to the previous graph where the generator could not reach constantly a loss smaller than 2. Finally, the last test was done using 2,142 images of self-portraits. The images contained self-portraits from Van Gogh, Picasso, Frida Kahlo and many different artists. Figure 18 shows the resulting images of using the portraits dataset.

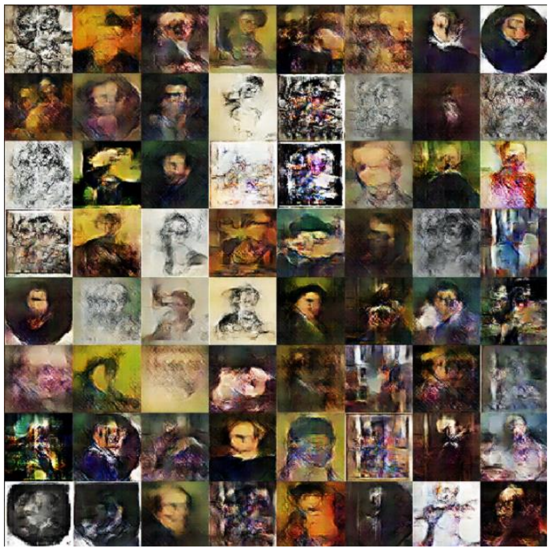


Figure 18. Resulting images from using the portraits dataset.

The resulting images show a very defined pattern in which the subject matter, meaning the person is always in the center of the picture. It also learned that some self-portraits where done in a circular frame, which explains the images that are inside a circle. Although the model was not able to learn how to draw a good face the overall image is quite good. During the training, accuracy was tried to be implemented to see the performance of the discriminator. The first complication presented was that in order to get a better performance of the overall network some noise was added to the labels meaning that the labels that the discriminator tried to predict were continuous numbers between 0 and 1, this creates the loss as seen in the previous graphs, but when trying to get the accuracy using the accuracy function implemented in keras, the labels needed to be changed from continues to discrete values, as the discriminator model was using the binary cross-entropy function that only outputs binary discrete values, if the labels were not changed it produced an accuracy of 0% all the time, but when the labels were changed, the performance of the network was negatively affected as shown in figure 19, figure 20, and figure 21 , where it shows that the discriminator reached its optimal point early on provoking that the generator could not learn anymore.

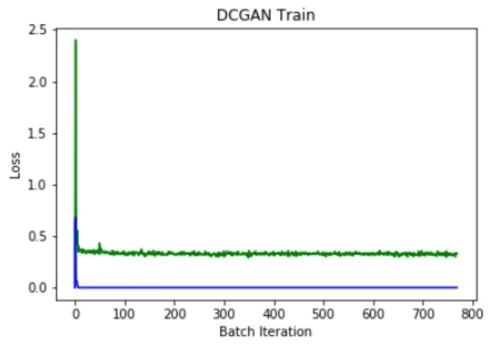


Figure 19. Graphic representation of loss of the DCGAN using discrete values

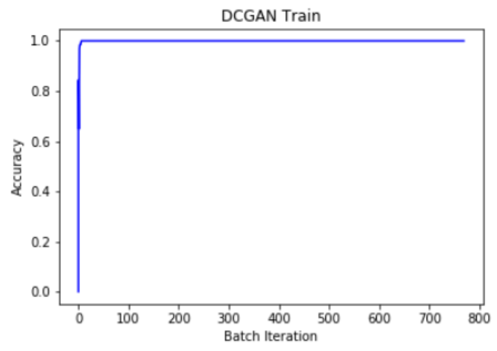


Figure 20. Graphic representation of the accuracy of the discriminator

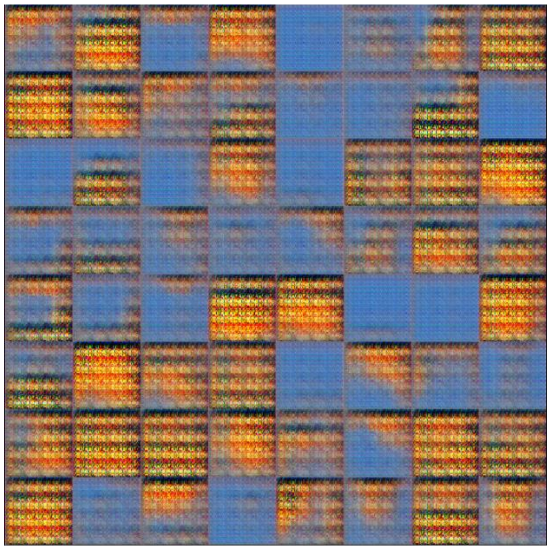


Figure 21. Resulting images from changing the continuous labels to discrete labels.

Therefore, the accuracy was implemented manually with the help of the predict function of the model before the training of the batch. Figure 22 was produced using the Expressionism dataset without regularization.

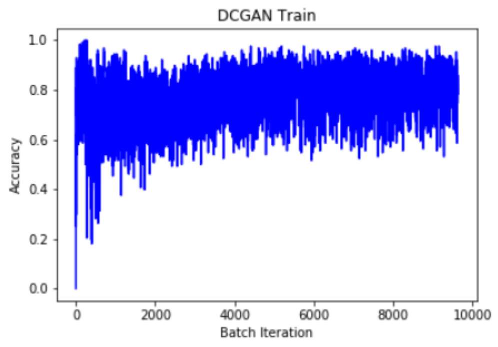


Figure 22. Graphic representation of the accuracy of the discriminator after the manual implementation

The graph shows the accuracy of the discriminator during training. It is very visible that at first the discriminator was very successful in classifying between real and fake images but overtime the discriminator's accuracy started to decrease because the generator was producing better images that were fooling the discriminator. Figure 23 shows the difference between the accuracy of the discriminator and the loss of the generator and discriminator over time.

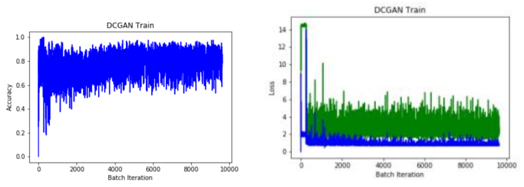


Figure 23. Discriminator accuracy v/s discriminator (blue) and generator (green) loss

Using the above comparison, it is possible to see that even though the discriminator had some iterations where it wasn't optimal, after the iteration number 4,000 the discriminator stabilized provoking the loss of learning in the generator.

VI. CONCLUSIONS

It is easier to recognize a Van Gogh's painting than to draw one. Generative models are considered much harder comparing with the discriminative models. Training a GAN is very hard because there has to be a balance between the discriminator and the generator, when the model collapses is often explained as an imbalance between the discriminator and the generator as what happened in the Van Gogh test, the obvious solution is to balance their training to avoid overfitting, however some researchers believe that avoiding overfitting in a GAN is not feasible or a desirable goal since a good discriminator gives good feedback. Some of the attention is therefore shifted for cost functions with nonvanishing gradients instead. The limited datasets did not help to achieve better results.

A way to improve this model could be using Wasserstein Generative Adversarial Network with a gradient penalty, use label information to train a better generator, for example using auxiliary GANs or conditional GANs, or explore different types of architecture.

VII. REFERENCES

[1] Skymind. (n.d.). A Beginner's Guide to Generative Adversarial Networks (GANs). Retrieved from https://skymind.ai/wiki/generative-adversarial-network-gan
[2] Che T, Jacob A, Li Y, Bengio Y, Li W. (2017). Mode Regularized Generative Adversarial Networks. Retrieved from https://openreview.net/pdf?id=HJKkY35le
[3] Hui J. (2018). GAN – DCGAN (Deep Convolutional Generative Adversarial Networks). Retrieved from https://medium.com/@jonathan_hui/gan-dcgan-deep-convolutional-generative-adversarial-networks-df855c438f
[4] WikiArt. (n.d.). Visual Art Encyclopedia. Retrieved from https://www.wikiart.org/
[5] Sharma S. (2017). Activation Functions in Neural Networks. Retrieved from https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6
[6] Brownlee J. (2017). Gentle Introduction to the Adam Optimization Algorithm for Deep Learning. Retrieved from https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/
[7] Shetty B. (2019). Curse of Dimensionality. Retrieved from https://towardsdatascience.com/curse-of-dimensionality-2092410f3d27
[8] Hui J. (2018). GAN – Why it is so hard to train Generative Adversarial Networks. Retrieved from https://medium.com/@jonathan_hui/gan-why-it-is-so-hard-to-train-generative-advisory-networks-819a86b3750b
Perez M. (2018). Coding Art with DCGAN code. https://colab.research.google.com/drive/1Zshhw1mSX2gp60lL7c_vmn_L2rhsfTcU

The implemented code used as a base the following codes:

DataSnaek. (2018). Simple Deep Convolutional Generative Adversarial Network built in Keras. Retrieved from https://github.com/DataSnaek/DCGAN-Keras/blob/master/DCGAN.py

Shibuya N. (2017). Face Generation with DCGAN. Retrieved from https://github.com/naokishibuya/deep-learning/blob/master/python/dcgan_celeba.ipynb

Gsurma. (2019). Image Generator Simpsons. Retrieved from https://github.com/gsurma/image_generator

Pavitrakumar78. (2017). Anime Face DCGAN. Retrieved from https://github.com/pavitrakumar78/Anime-Face-GAN-Keras