GENERATING COVID-19 CHEST X-RAY IMAGES USING DCGAN AND WGAN-GP

Hien Nguyen - 300199540

Victoria University of Wellington

1. INTRODUCTION

Class imbalance often affects medical image classification, like X-ray detection for conditions such as COVID-19 or pneumonia. Manual detection is pricey, and expanding datasets raises privacy issues. For deep learning to succeed, substantial data is crucial to prevent overfitting. Generative Adversarial Networks (GANs) have recently emerged as a solution, generating synthetic images to balance datasets. In paper [1], the authors attempted to generate additional COVID-19 chest X-ray images using both the Deep Convolutional GAN (DCGAN) and the Wasserstein GAN with Gradient Penalty (WGAN-GP), subsequently comparing their performances. This report primarily replicates that paper, emphasizing testing various hyperparameters and delving into the underlying theories of GANs. The chosen metric for comparing the performance of the two GAN types is the Fréchet Inception Distance (FID) score. Section 2 in this report outlines the fundamental theories of GAN: 2.1 presents the original GAN and its limitations, 2.2 details DCGAN's structure and its loss function shortcomings, and 2.3 introduces WGAN-GP as an enhancement over DCGAN by using a different loss function. Then, Section 2 ends with a theory discussion of the FID score. Section 3.1 illustrates the experimental setup and execution based on the paper [1], followed by hyperparameters testing in Section 3.2. The results and GAN comparisons are drawn in Section 3.3. Conclusions are summarized in Section 4. After training on 2000 images over 500 epochs, WGAN-GP emerged superior with a more favorable FID score.

2. THEORY

2.1. Generative Adversarial Networks

As the name suggests, a GAN is a battle between two adversaries, the generator and the discriminator. Two different models work against each other but simultaneously improve themselves. Using the output from a generative model as an input for a classification model allows us to gauge the performance of the generative model with the classification one. Moreover, the classification model can be enhanced by incorporating generated samples with real ones, given that more data typically benefits machine learning model training. This iterative process where both models challenge and strengthen each other is termed adversarial learning. As illustrated in Appendix Figure 1, models A and B have divergent objectives, such as classification versus generation, but each model's output refines the other during training stages [2].

In detail, The generator uses random noise to produce fabricated samples, aiming to make them closely resemble real samples. On the other hand, the discriminator functions as a classifier, distinguishing between real and fake samples. While the generator works to deceive the discriminator, the discriminator works to identify the counterfeits. The generator refines its output as the system identifies discrepancies between real and fake samples, making the generator produce more like real samples and the discriminator more adept at spotting them. While real samples guide the discriminator's training, making it supervised, the generator operates without knowing ground truth (as seen in Figure 2) [2].

Let G and D represent the generator and discriminator networks, respectively.
V is the performance criterion of the system
x is the real sample
z is the random noise that G uses to generate fake samples
$x^{\star} = G(z)$ is the generated sample
$E_x[f]$ is the expectation over x, which means the average value of any function, f, over all samples. We have an optimization objective that is

$$min_G \, max_D \, V(D, G) = E_x[\log D(x)] + E_{x^{\star}}[\log(1 - D(x^{\star}))] \quad (1)$$

The first right-hand term of the value function (1) is the value to classify a real image correctly. The discriminator, D, aims to maximize the prediction confidence of real samples. Expectation is a mathematical term that is the sum of the weighted average of every random variable sample. The weight is the probability of data, and the variable is the log of the discriminator output. Thus, we have

$$E_X[\log D(x)] = \sum_{i=1}^{N} p(x)\log D(x) = \frac{1}{N}\sum_{i=1}^{N} \log D(x) \quad (2)$$

so, $min_D V(D) = -\frac{1}{N}\sum_{i=1}^{N} \log D(x) = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(p(y_i))$ (3)

y_i is the label, which is 1 for real images, and $p(y_i)$ is the probability of the sample being real. [3]

The second right-hand term of the value function is about fake images. $D(x^\star)$ or $D(G(z))$ is the discriminator's confidence score of how likely the image is to be real. If we use a label of 0 for fake images, we have

$$-E_{x^\star}[\log(1 - D(x^\star))] = -\frac{1}{N}\sum_{i=1}^{N}(1 - y_i)\log(1 - p(y_i)) \quad (4)$$

Thus, the discriminator loss function is a binary cross-entropy loss. [3]

$$min_D V(D) = -\frac{1}{N}\sum_{i=1}^{N} y_i \log(p(y_i)) + (1 - y_i)\log(1 - p(y_i)) \quad (5)$$

In other words, D needs to be trained with gradient ascent (the max operator in the objective).

$$\theta_d \leftarrow \theta_d + \frac{1}{m}\nabla_{\theta_d}\sum_{i=1}^{m}(\log D(x_i) + \log(1 - D(x_i^\star))) \quad (6)$$

θ_d is the parameter of D (such as convolution kernels and weights in fully connected layers), m is the mini-batch size, and i is the sample index in the mini-batch. [2]

The goal of the generator network, G, is to fool the discriminator, and let it believe that the generated samples are real. Thus, it is only involved when the model evaluates fake images, represented in the value function's second right-hand term. [3]

$$min_G V(G) = E_{x^\star}[\log(1 - D(x^\star))] \quad (7)$$

When training starts, the generator is not great at making images, so the discriminator quickly labels them as fake, scoring them a consistent 0, making $D(x^\star)$ always 0, and so is $\log(1 - 0)$. As a result, there is no error to adjust from, meaning there is no feedback to help the generator improve. The generator's learning stalls because the discriminator's prediction does not change. It is often called the "saturating gradient" issue. To tackle this, we can maximize how often it is fooled instead of minimizing how often the discriminator is tricked $1 - D(x^\star)$). [3]

$$max_G V(G) = E_{x^\star}[\log D(x^\star)] \quad (8)$$

In other words, the training of G is to maximize $D(G(z))$ or minimize $1 - D(G(z))$. Therefore, G needs to be trained with gradient descent (the min operator in the objective). [1]

$$\theta_g \leftarrow \theta_g + \frac{1}{m}\nabla_{\theta_g}\sum_{i=1}^{m}(\log(1 - D(G(z_i))) \quad (9)$$

One major weakness of GAN is 'mode collapse'. For example, if the generator gets really good at making pictures of shoes, tricking the discriminator more often with them. It then churns out even more shoe images, eventually forgetting how to make anything else. The discriminator, mostly seeing fake shoe images, might forget how to spot fake versions of other items. If the discriminator ever figures out the shoe ruse, the generator might shift to another item, like shirts, leaving shoes behind, leading to the GAN switching between a few categories without mastering any. Additionally, since both parts of the GAN constantly try to outdo each other, their inner workings might wobble and become unstable. Things might seem smooth initially but can go off track out of the blue. Furthermore, GANs react strongly to minor tweaks in their settings and are very sensitive to hyperparameters. There is also the issue of 'vanishing gradients'. As the discriminator becomes increasingly confident in its decisions, the feedback it gives to the generator can become weak or uninformative, which causes the gradient (the tool guiding the generator's learning) to shrink or vanish, halting the generator's improvement. [4]

2.2. Deep Convolutional GAN

DCGAN is known for helping the original GAN scale the networks to make them deeper to increase their capacities and stabilize the training. There are some main guidelines for building stable convolutional GANs:

- Replace any pooling layers with strided convolutions (in the discriminator) and transposed convolutions (in the generator).
- Use batch normalization in both the generator and the discriminator, except in the generator's output layer and the discriminator's input layer.
- Remove fully connected hidden layers for deeper architectures.
- Use ReLU activation in the generator for all layers except the output layer, which should use tanh.

- Use leaky ReLU activation in the discriminator for all layers. [4]

"Batch normalization" or "batch norm" greatly improves training deep neural networks. Before it, when a layer in a network adjusted its weights to get better results, the layers after it also shifted their weights, making the training target constantly move and training deep nets tricky. Batchnorm stepped in to help by ensuring that every layer's input was consistently adjusted to have a steady average and spread. It first figures out the average (μ) and how spread out the data is (σ) for every channel in a batch of data. It then adjusts the data using this formula: $x' = (x - \mu)/\sigma$ (10). After that, it tweaks the data using the formula: $y = \alpha * x' + \beta$ (11) (α and β are variables that can be learned and adjusted during training). In DCGAN, batch norm is added to both the generator and the discriminator. However, it skips the discriminator's first layer and the generator's final layer. [3]

The activations we will use in DCGAN are sigmoid, tanh, ReLU, and leaky ReLU (alpha = 0.2). As the discriminator's job is to be a binary classifier, we use sigmoid to squeeze the output between 0 (fake) and 1 (real). Meanwhile, the generator's outputs are restricted between [-1, 1] using the tanh function. For the layers in between, the generator employs ReLU, whereas the discriminator goes with leaky ReLU. Standard ReLU boosts positive inputs but does not allow negative inputs to pass through, which can hinder the generator's learning by stopping gradient updates. Leaky ReLU overcomes this by letting through a small amount of negative activations. As shown in Figure 3, for any input equal to or bigger than 0, leaky ReLU acts just like regular ReLU, giving back the same value it receives with a slope of 1. However, it scaled the output to be 0.2 times the input value for negative input values. [3]

Once the discriminator is fine-tuned, the generator starts optimizing based on the Jensen-Shannon divergence (JSD). JSD is a symmetrical version of Kullback-Leibler divergence (KLD) with an upper bound of log(2) rather than an infinite upper bound. Unfortunately, JSD is also the cause of mode collapse. As shown in Figure 4, data from two Gaussian distributions is being fitted by a single Gaussian model, using one mean and standard deviation to represent two different data sets. With KLD, the fitted Gaussian favors the larger distribution but does not entirely ignore the smaller one. However, the fit gravitates solely towards the more dominant distribution with JSD, leaving the smaller one out. As such, it highlights the mode collapse in GANs: the generator might focus only on specific high-probability outputs, and once the optimizer settles on these few patterns, it does not venture beyond them.[3]

2.3. Wasserstein GAN with Gradient Penalty

In 2016, one breakthrough happened with the introduction of Wasserstein GAN (WGAN). WGAN tackles many of the issues commonly faced by traditional GANs. There is less need to meticulously design the network or delicately balance the two competing parts. Plus, it hugely reduces the mode collapse problem. The improvement for WGAN compared to the original GAN is its revamped loss function. The problem with the old loss function was that if the two data distributions did not overlap, the JSD became non-continuous and thus not differentiable, leading to a standstill in learning due to a zero gradient. WGAN sidesteps this issue by introducing a loss function that's smooth and differentiable throughout, ensuring consistent learning. The new loss function is the Earth mover's distance or Wasserstein distance. It measures the distance or the effort needed to transform one distribution into another. It is the minimum distance for every joint distribution between real and generated images. Labels in the Wasserstein loss are slightly different: we use $y_i = 1$ and $y_i = -1$ instead of the usual 1 and 0. Also, the last layer of the discriminator drops the sigmoid activation, allowing its predictions to go beyond the [0, 1] boundary, ranging from $[-\infty, \infty]$. Therefore, in WGANs, the discriminator is often called a critic that gives a score rather than a probability estimate. The Wasserstein loss function is defined as $-\frac{1}{N}\sum_{i=1}^{N}(y_i p_i)$ (12). [5]

To train the WGAN critic D, we calculate the loss when comparing predictions for real images $p_i = D(x_i)$ to the response $y_i = 1$ and predictions for generated images $p_i = D(x_i^\star)$ to the response $y_i = -1$. Therefore, for the WGAN critic, minimizing the loss function can be written as: $min_D - (E_{x\sim P_X}[D(x)] - E_{z\sim P_Z}[D(x^\star)])$ (13). In other words, the WGAN's critic tries to maximize the difference between its predictions for authentic images and generated images. [5]

To train the WGAN generator, we calculate the loss when comparing predictions for generated images $p_i = D(x_i^\star)$ to the response $y_i = 1$. Therefore, for the WGAN generator, minimizing the loss function can be written as: $min_G - (E_{z\sim P_Z}[D((x^\star))])$ (14). In other words, the WGAN generator tries to produce images that are scored as highly as possible by the critic (i.e., the critic is fooled into thinking they are real). [5]

Unusually, the critic can now give outputs ranging from $[-\infty, \infty]$. Larger values in neural networks are also typically a red flag. As a result, to make the Wasserstein loss function effective, we can impose an extra condition on the critic. It has to be a 1-Lipschitz continuous function. The critic, D, turns an image into a prediction. This D function is termed 1-Lipschitz if it satisfies the following inequality for any two input images, x_1 and x_2: $\frac{|D(x_1) - D(x_2)|}{|x_1 - x_2|} \leq 1$ (15). $|x_1 - x_2|$ is the average pixel-wise absolute difference between two images, and $|D(x_1) - D(x_2)|$ is the absolute difference between the critic's predictions. We are defining how quickly the critic's predictions can shift between two images, meaning the absolute gradient value should not exceed one anywhere. A Lipschitz continuous 1D function depicted in Figure 5, no matter where we position a cone over the line, the line never cuts through the cone. There is a ceiling on how steeply the line can go up or down at any given spot. [5]

The authors in the WGAN original paper use weight clipping to enforce the inequality, meaning clipping the critic's weights to some small values, typically between the range of [-0.01, 0.01]. However, there are better approaches than weight clipping to maintain a Lipschitz constraint. This method faces two main issues: capacity underuse and exploding/vanishing gradients. Firstly, we limit the critic's learning potential by putting caps on the weights. This restriction tends to force the network only to grasp basic functions, making it not used to its full potential. The second problem is about fine-tuning the clipping values. If these values are too high, we end up with runaway gradients, which goes against the Lipschitz requirement. If they are too low, the gradients disappear, making it harder for the network to adjust and learn. The weight clipping tends to push gradients to extreme ends (Figure 6). [3]

One better way to enforce the Lipschitz constraint and improve the capacity of the WGAN to learn complex features is using the Wasserstein GAN with Gradient Penalty. Lipschitz constraint can be enforced directly by including a gradient penalty term in the loss function for the critic that penalizes the model if the gradient norm deviates from 1. The gradient penalty loss measures the squared difference between the norm of the gradient of the predictions to the input images and 1. The model will naturally be inclined to choose weights that minimize this gradient penalty, thereby encouraging the model to conform to the Lipschitz constraint. [5]

Wasserstein's distance with gradient penalty =
$E_{z\sim P_z}[D(x^{\star})] - E_{x\sim P_X}[D(x)] + \lambda E_{\hat{x}\sim P_{\hat{x}}}[(||\nabla_{\hat{x}} D(\hat{x})||_2 - 1)]^2$ (16). [1]

Calculating this gradient everywhere during the training process is intractable, so instead, the WGAN-GP evaluates the gradient at only a handful of points. To ensure a balanced mix, we use a set of interpolated images ($\hat{x}$) that lie at randomly chosen points along lines, connecting the batch of real images to the batch of fake images pairwise [5]. The ratio of the images, or alpha, is drawn from a uniform distribution of [0,1]. $\nabla_{\hat{x}} D(\hat{x})$ is the gradient of the critic's output with respect to the interpolation, and $||\nabla_{\hat{x}} D(\hat{x})||_2$ is the L2-norm calculation. The lambda in the equation is the ratio of the gradient penalty to other critic losses (usually set at 10). [3]

2.4. Fréchet Inception Distance (FID) score

The FID score measures how good a GAN's generated images are by comparing the Fréchet distance between these generated images and the original training images. This comparison utilizes the Fréchet distance between two multidimensional Gaussian distributions, sourced from the 2048-dimensional outputs of the InceptionV3 model's pooling layer for real and generated images. A lower FID score indicates better GAN performance. [1]

3. RESULT

3.1. Experimental setup

There are three GAN models trained on two datasets. The original dataset contains 3616 COVID-19 chest X-ray images size 299 x 299. Before passing the data to the models, images are pre-processed firstly using Histogram Equalization (CLAHE), then resized to either size 128x128 pixels (dataset 1) or size 64x64 pixels (dataset 2). Dataset 1 is used to train WGAN-GP only, while dataset 2 is for both WGAN-GP and DCGAN. The architectures for both models trained on dataset 2 are very similar (Figure 7) except for the loss function, where the DCGAN uses a binary cross-entropy loss function, and the WGAN-GP uses the Wasserstein's distance with gradient penalty.

In the DCGAN's generator structure, the starting point is a noise vector of 100 x 1 in size. The vector then gets upscaled through five convolution-transpose layers. After each of these upsampling layers, there is a BatchNorm2d layer, followed by a ReLU layer, except for the final set. The ending layer, or the output layer, upsamples and then applies a Tanh function, giving a 64x64x3 sized output with values ranging from [-1, 1]. The discriminator begins with an input layer that takes in a 3x64x64 vector produced by the generator. It is then passed through five convolution layers. Post every convolution layer, there is a LeakyReLU layer with a slope of 0.2, except for the last one. The final layer uses a convolution process and a sigmoid function, determining the likelihood of the image being genuine or counterfeit.

There are a few tweaks in the WGAN-GP's architecture compared to DCGAN's. Instead of using a batch normalization layer in the discriminator, the WGAN-GP employs an InstanceNorm layer. This adjustment ensures that the critic's penalty operates correctly. Additionally, the discriminator's final layer does not have a sigmoid function because it is not trying to determine the likelihood of an image being real or fake. Instead, it assigns a score to each image.

The WGAN-GP model trained on dataset 1 is mostly the same as the WGAN-GP mentioned above but has the spatial dimensions of the feature maps twice as large by the end of the model to adapt for 128 x 128 images. One way to achieve this is to add one of the upsampling layers from the generator and one of the downsampling layers from the critic (Figure 7). The WGAN-GP model trained on dataset 1 is chosen as the base model to study the effects of different hyperparameters.

3.2. Hyperparameters testing

Normalizing images or adjusting pixel values from [0, 255] to [-1, 1] is recommended. While the CLAHE method from the cv2 package aids in this, it is also valuable to try Pytorch's normalization function to determine which produces a better FID score. At a glance, images not processed with the Pytorch normalization function appear more enhanced than the originals. In contrast, the normalized ones seem to lose some details (as seen in Figure 8). The FID scores reinforce this observation: unnormalized images score higher than their normalized counterparts (960 vs. 1336). Hence, The non-normalization technique is adopted in subsequent steps.

Specific base hyperparameters will remain constant during model training, and only one of them at a time is adjusted to observe its impact on the FID score. The foundational hyperparameters, which result in an FID score of 960, include the dataset size of 1000 images, the noise factor of 0, 50 epochs, and the image size of 128x128.

The subsequent analysis involves comparing dataset sizes (as shown in Appendix Table 1). Generally, larger datasets improve training outcomes. As anticipated, models trained with 2000 and 3000 images have FID scores of 949 and 941, respectively, better than the score of 960 achieved with a 1000-image dataset. However, given the resource-intensive nature of training and the marginal improvement in FID scores between the 2000 and 3000 image datasets, it is preferable to proceed with 2000 images for further training and generation before introducing the data to the classification model.

Considering that GANs produce images from Gaussian noise, it is worth investigating the sensitivity of WGAN-GP's generator to varying noise factors. The torch.randn function creates a tensor populated with random numbers from a standard normal distribution (mean 0 and variance 1). By adding the noise factor * torch.randn, we infuse additional randomness to the base noise, although within a constrained range. Table 2 confirms that the original noise factor of 0 remains optimal. Moreover, the WGAN-GP consistently yields comparable image quality across different noise factors, as no significant difference exists between the FID scores.

Up to this point, the WGAN-GP has only been trained for 50 epochs, resulting in somewhat blurry and incomplete images. Extended training is warranted. The epoch count itself is a tunable hyperparameter. As per Table 3, the FID scores show significant improvement up to the 500th epoch. However, by the 700th epoch, the WGAN-GP ceased to enhance, with the FID score even rising slightly.

A notable observation is that the WGAN-GP trained on 64x64 images outperforms its counterpart trained on 128x128 images (Table 4), which suggests that image size might be important in enhancing GAN performance.

3.3. Models comparison

FID scores are used to assess the quality of images produced by DCGAN and WGAN-GP (the corresponding images are presented in Figure 9). For a fair comparison, both GAN models were trained using the same dataset of 2000 COVID-19 chest X-ray images, sized at 64x64, with consistent hyperparameters: a batch size of 128, noise factor set to 0, and 500 epochs. Table 5's results indicate that WGAN-GP outperforms DCGAN, as expected. Moreover, Table 6 reveals that the DenseNet161 COVID-19 image classifier, when utilizing augmented data from WGAN-GP, performs slightly better than DCGAN in all accuracy metrics. In particular, the Recall score stands out, which is particularly vital given the preliminary disease screening context. Moreover, the confusion matrix in Figure 10 highlights that the model employing WGAN-GP-generated images misclassified three actual disease cases as normal compared to DCGAN's six.

4. CONCLUSION

GANs address the class imbalance by generating high-quality, realistic COVID-19 chest X-ray images, even when trained on limited datasets. WGAN-GP outperforms DCGAN (measured by the FID scores), corroborating theoretical predictions. Factors like image size, normalization, processing method, and number of epochs significantly influence GAN's performance. However, WGAN-GP demonstrates resilience to varying noise factor levels, simplifying tuning by reducing the number of parameters to adjust. In conclusion, while classifiers utilizing datasets enhanced by DCGAN and WGAN-GP produce commendable outcomes, WGAN-GP may be preferable for medical datasets, especially given its superior Recall score.

5. STATEMENTS

Google Colab:
https://colab.research.google.com/drive/1oq08BeHJ3WphL0E5kZ12zK0b-28jIl51?usp=sharing
torch version 2.0.1+cu118
PIL version 9.4.0
numpy version 1.23.5
torchvision version 0.15.2+cu118
random, math, os, Ipython, Matplotlib, tqdm, scipy, mlxtend sklearn

Reference code for DCGAN:
https://pytorch.org/tutorials/beginner/dcgan_faces_tutorial.html

Reference code for WGAN-GP:
https://github.com/eriklindernoren/PyTorch-GAN/blob/master/implementations/wgan_gp/wgan_gp.py

Reference code for classifier model DenseNet161:
https://www.kaggle.com/code/arunrk7/covid-19-detection-pytorch-tutorial

APPENDIX

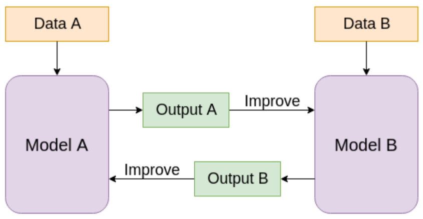


Figure 1: A typical adversarial learning system [2]

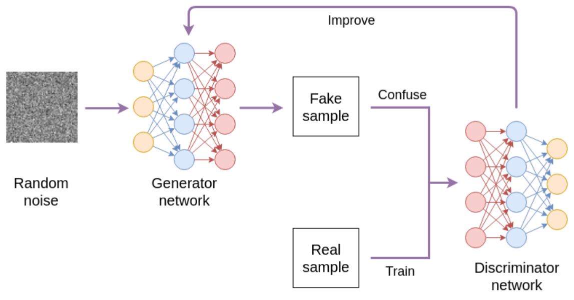


Figure 2: Basic process of a GAN [2]

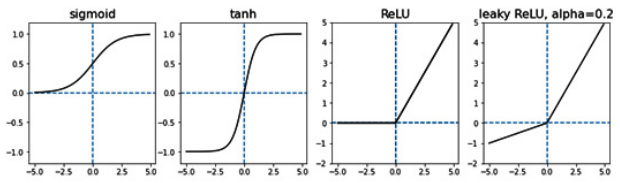


Figure 3: Activations functions [3]

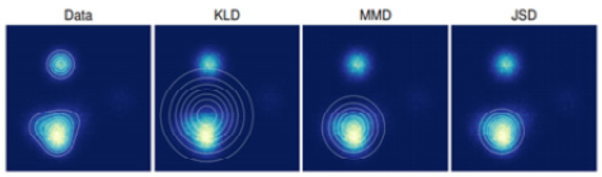


Figure 4: KLD, MMD, and JSD objective functions [3]

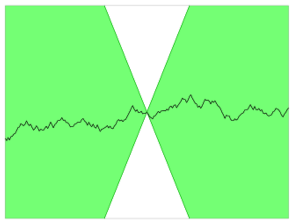


Figure 5: A Lipschitz continuous function [5]

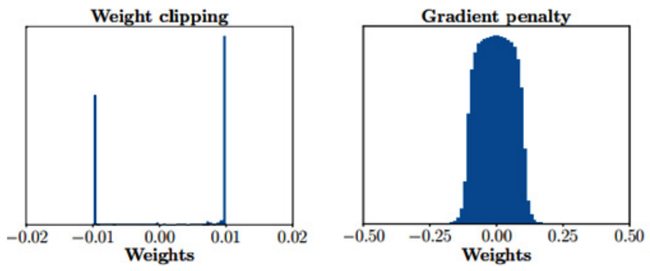
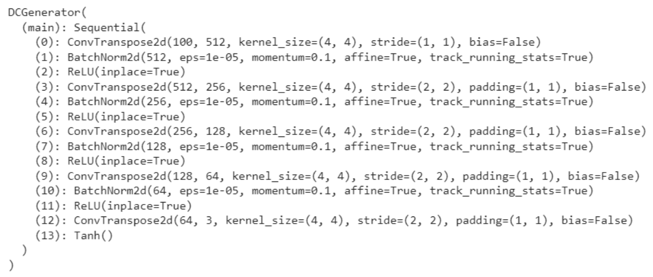


Figure 6: Left: Weight clipping pushes weights toward two values. Right: Gradients produced by gradient penalty. [3]



```
DCGenerator(
  (main): Sequential(
    (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU(inplace=True)
    (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (13): Tanh()
  )
)
```

DCGAN Generator for 64 x 64 images

```
Generator64(
  (gen): Sequential(
    (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU(inplace=True)
    (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (13): Tanh()
  )
)
```

WGAN-GP Generator for 64 x 64 images

```
Generator(
  (gen): Sequential(
    (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1))
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU(inplace=True)
    (12): ConvTranspose2d(64, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (13): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (14): ReLU(inplace=True)
    (15): ConvTranspose2d(32, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (16): Tanh()
  )
)
```

WGAN-GP Generator for 128 x 128 images

```
DCDiscriminator(
  (main): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): LeakyReLU(negative_slope=0.2, inplace=True)
    (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (3): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): LeakyReLU(negative_slope=0.2, inplace=True)
    (5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (6): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (7): LeakyReLU(negative_slope=0.2, inplace=True)
    (8): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (9): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): LeakyReLU(negative_slope=0.2, inplace=True)
    (11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (12): Sigmoid()
  )
)
```

DCGAN Discriminator for 64 x 64 images

```
Critic64(
  (crit): Sequential(
    (0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (1): InstanceNorm2d(64, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)
    (2): LeakyReLU(negative_slope=0.2, inplace=True)
    (3): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (4): InstanceNorm2d(128, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)
    (5): LeakyReLU(negative_slope=0.2, inplace=True)
    (6): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (7): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)
    (8): LeakyReLU(negative_slope=0.2, inplace=True)
    (9): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (10): InstanceNorm2d(512, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)
    (11): LeakyReLU(negative_slope=0.2, inplace=True)
    (12): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
  )
)
```

WGAN-GP Critic for 64 x 64 images

```
Critic(
  (crit): Sequential(
    (0): Conv2d(3, 16, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (1): InstanceNorm2d(16, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)
    (2): LeakyReLU(negative_slope=0.2)
    (3): Conv2d(16, 32, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (4): InstanceNorm2d(32, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)
    (5): LeakyReLU(negative_slope=0.2)
    (6): Conv2d(32, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (7): InstanceNorm2d(64, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)
    (8): LeakyReLU(negative_slope=0.2)
    (9): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (10): InstanceNorm2d(128, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)
    (11): LeakyReLU(negative_slope=0.2)
    (12): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))
    (13): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)
    (14): LeakyReLU(negative_slope=0.2)
    (15): Conv2d(256, 1, kernel_size=(4, 4), stride=(1, 1))
  )
)
```

WGAN-GP Critic for 128 x 128 images

Figure 7: All GAN's architectures

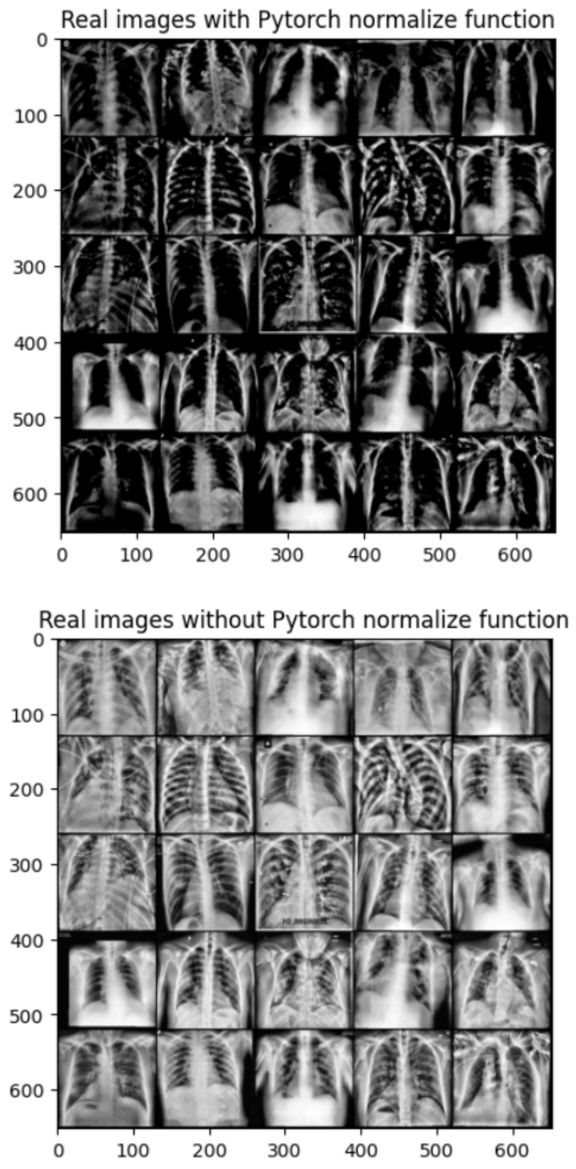
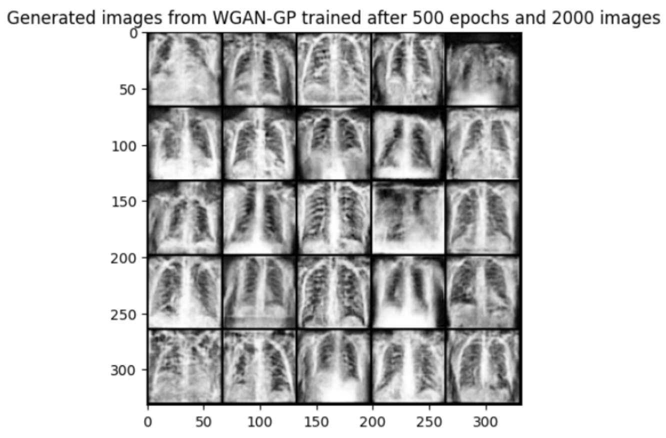

Real images with Pytorch normalize function
0
100
200
300
400
500
600
Real images without Pytorch normalize function

Figure 8: Real images after pre-processing


Generated images from WGAN-GP trained after 500 epochs and 2000 images
0
50
100
150
200
250
300

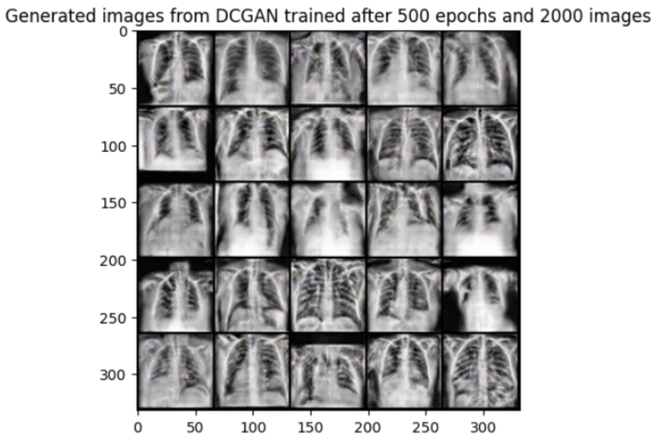

Generated images from DCGAN trained after 500 epochs and 2000 images
0
50
100
150
200
250
300

Figure 9: Generated images (64x64) by WGAN-GP and DCGAN

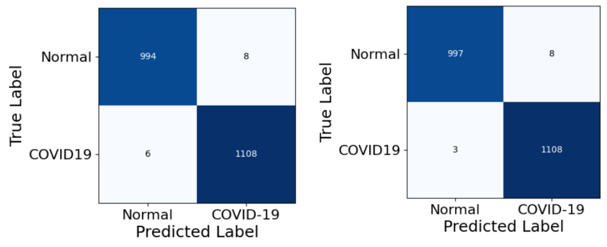


Figure 10: Confusion matrix of models using DCGAN (left) and WGAN-GP (right)

Dataset Size (Number of Actual COVID-19 Positive Images)	FID Score of the Chest X-ray Images Generated by WGAN-GP
1000	960
2000	949
3000	941

Table 1: FID Score with different dataset sizes

Noise Factor (The level of extra noise in the Generator's input)	FID Score of the Chest X-ray Images Generated by WGAN-GP
0	960
0.01	1031
0.1	1242
0.5	1181
1	1121

Table 2: FID Score with different noise factors

Number of epochs	FID Score of the Chest X-ray Images Generated by WGAN-GP
50 epochs	960
100 epochs	1062
300 epochs	640
500 epochs	612
700 epochs	619

Table 3: FID Score with different number of epochs

Image Size (Resolution of an image measured in pixels)	FID Score of the Chest X-ray Images Generated by WGAN-GP
128 x 128	960
64 x 64	696

Table 4: FID Score with different image sizes

GAN Type	FID Score
WGAN-GP	110
DCGAN	155

Table 5: FID scores of DCGAN and WGAN-GP

Dataset	Accuracy	Recall	Precision	F1 Score
COVID-19 positive + images generated from WGAN-GP	0.9948	0.9973	0.9928	0.9951
COVID-19 positive + images generated from DCGAN	0.9934	0.9946	0.9928	0.9937

Table 6: Model accuracy metrics

REFERENCES

[1] Mai Feng Ng, & Carol Anne Hargreaves. (2023). *Generative Adversarial Networks for the Synthesis of Chest X-ray Images*. Engineering Proceedings, 31(84), 84–. https://doi.org/10.3390/ASEC2022-13954

[2] Hany, J., & Walters, G. (2019). *Hands-on generative adversarial networks with Pytorch 1.x : implement next-generation neural networks to build powerful GAN models using Python (1st edition)*. Packt.

[3] Cheong, S. Y. (2020). *Hands-On Image Generation with TensorFlow: A Practical Guide to Generating Images and Videos Using Deep Learning (1st ed.)*. Packt Publishing, Limited.

[4] Aurélien Géron. (2022). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow, 3rd Edition*. O'Reilly Media, Inc.

[5] Foster, D. (2019). *Generative deep learning : teaching machines to paint, write, compose, and play (First edition.)*. O'Reilly.