Using a GAN for data augmentation of chest X-Ray images for detection and classification of COVID-19 virus in patients

Group Members
Varun Govind (varung2) varung2@illinois.edu
Vyom Thakkar (vnt2) vnt2@illinois.edu

1 Introduction

In this project, we aim to create a Generative-Adversarial-Network (GAN) to augment existing chest X-Ray images of patients lungs with different preexisting conditions to aid with the detection of COVID-19, and additionally create a classifier that is able to classify the X-Rays into the patients conditions (COVID-19, Pneumonia, Normal, etc.). The final goal of this project is to be able to augment an existing X-Ray dataset with the GAN and additionally reliably detect if a patient has COVID-19 (or some other condition) or if he is healthy by looking at his/her chest X-Rays.

Currently the world is still suffering from the hold of COVID-19 and while medicine has advanced considerably with the creation and distribution of vaccines and testing kits, people are still getting infected and are getting hospitalized due to extreme symptoms. Additionally, while vaccines are being distributed and the rate of infections are reducing, it is still unclear whether COVID-19 will be eradicated or if it will remain alongside us (humans) after the fact. This project would help validate results from the paper and perhaps push new directions in helping detect the virus.

We present our work through this paper and additionally through a video presentation hosted on youtube [6]. Our code and augmented dataset are additionally hosted on google drive and github for reference [7].

2 Design Approach

For our machine learning model architecture, we follow in the footsteps of Khalifa M. [4]. The paper describes a generator architecture and discriminator architecture in which they use to synthetically boost their dataset to train their classifier. Although we follow some parts of the architecture, we ultimately take a slightly different approach to the process. In our design, we are creating a Conditional GAN. The conditional GAN will essentially take in a label prior along with some random noise so that the image it is trying to generate can specifically create what we want. In the reference paper ([4]), they only use the GAN to generate X-Ray images of patients with COVID-19, however, in our approach, we want to create a GAN that can create X-Ray images of patients with and without COVID-19. The solution is to train a GAN that takes in labels and noise as input and outputs an image with respect to the label. In other words, a Conditional GAN; it is simple in theory but difficult to train in practice, as we will show later.

For some background, GANs are typically constructed with two main parts: a generator and a discriminator. The generator's purpose is to create a 'fake' object (typically an image but can be something else in a different problem domain). The discriminator's purpose is to determine if the object/image given to it is fake or real. We construct a loss out of the discriminator on whether it was right or wrong and pass that to our generator. So the idea is to create an adversary for the generator in which the generator tries to fool. So when the discriminator gets better at distinguishing real images versus fake images, the generator has to improve to fool the discriminator. At some point, the discriminator has trouble distinguishing between real and fake and more training needs to be done for the discriminator. This process goes back and forth until we get a model that can produce something realistic (according to what training data was given). This form of unsupervised learning forces the model to learn what the distribution is of the training data that is given. In our specific case, we want our model to learn the specific patterns, features, and differences between X-Ray chest scans of patients with and without COVID-19.

For the classification task (determining whether an input X-Ray image is Covid/Normal) we will be using a CNN (convolutional neural network). For the design approach, we took three CNN models (VGG16, Resnet50 and InceptionV3) pre-trained on the Ima-

genet dataset and augmented these with custom fully connected layers for the classification task at hand. We carried out experiments to see which one of these three models performs the best on our dataset and took the corresponding architecture as our best model. The data used to train the model was split equally for the two classes and has 60:20:20 train-validation-test split. The data that we used to train our baseline model has 3616 Covid and 3616 Normal X-Ray images and each of the train, validation and test sets have a perfect class balance. After obtaining the results using the original dataset, we augmented the dataset with GAN generated images (1000 Covid and 1000 Normal) and subsequently measured the performance of the three models after data augmenting the original dataset.

2.1 Generator Architecture

For our architecture design, we follow the deep convolution layers closely but since we are creating a Conditional GAN, some of the layers at the beginning will vary. For our generator architecture, we use an input embedding layer for the labels and a dense layer for the image. We then concatenate the output of those two layers channel wise and input them into our special convolution layers. Note that our special convolution layers can be decomposed into simpler blocks referred to as **conv_layer_set**. We then have a final set of layers, which is similar to the **conv_layer_set** but it omits the 2D batch normalization and has a Tanh as the final activation function. As you can see in the figure below (Figure 1), it is a relatively straight forward architecture. For the number of filters, we use 64 filters and a square filter width of 4 pixels. Note that we use 2D convolution transpose which serves to upsample the input by convolving with weights. Typical 2D convolution will keep the same shape or reduce it.

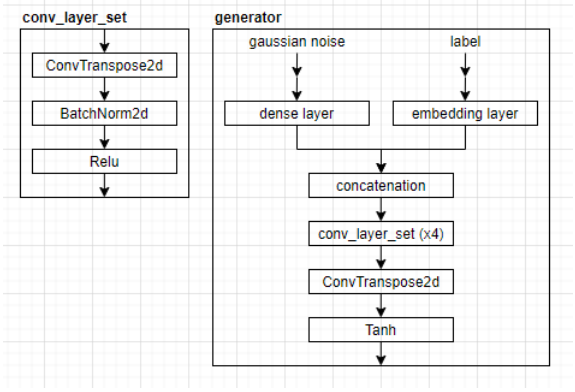

Figure 1: Generator Architecture

2.2 Discriminator Architecture

In our discriminator design, it is very similar to our generator architecture, with some subtle differences. We refer to the same special convolution layer as **conv_layer_set** but it is different than the one mentioned in the generator architecture as it contains two different layers (Figure 2). The first difference is that we don't use 2D convolution transpose and the second is the activation function after the 2D batch normalization; we use 'LeakyReLU' as an activation.

Our discriminator takes our image and concatenates it with our embedding layer which takes as input, our label. We initially pass our concatenated layer into a 2D convolution layer with a 'LeakyReLU' activation function. Notice that we omitted the batch normalization after the convolution layer. This is intentional. For our next set of layers, we use our **conv_layer_set** three times and follow it with another convolution and 'LeakyReLU' (again without batch normalization). This ends our convolution layer and leads to our fully connected layer. We perform a flattening of the layer and dropout some weights (at a probability of 50%) and follow it with a dense layer and a sigmoid activation function. The sigmoid's activation function is to serve as the final step in determining the likelihood of the input image being real or fake.

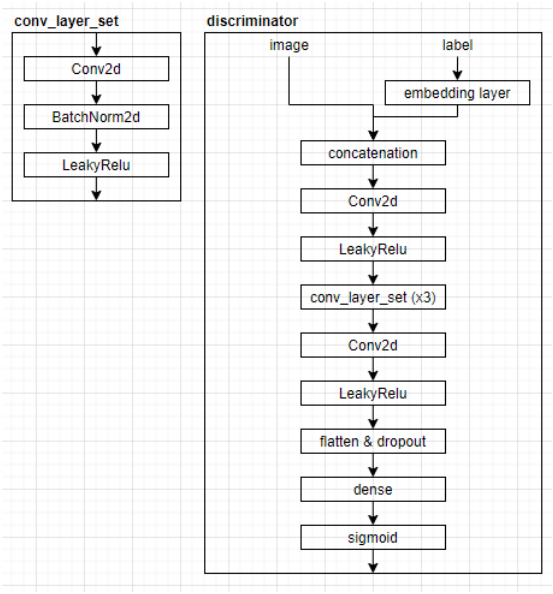

Figure 2: Discriminator Architecture

As you can see from our figure above, our discriminator is slightly more complicated than our generator but is essentially constructed of the same components. Note that we don't use the same 2D convolution as the

generator; we use the normal 2D convolution layer and not the 2D convolution transpose.

2.3 CNN Architecture

As was mentioned before, we used three pretrained (on the Imagenet dataset) models: VGG16, Resnet50 and InceptionV3 in our experiments. Our architecture of the CNN is shown below in Figure 3.

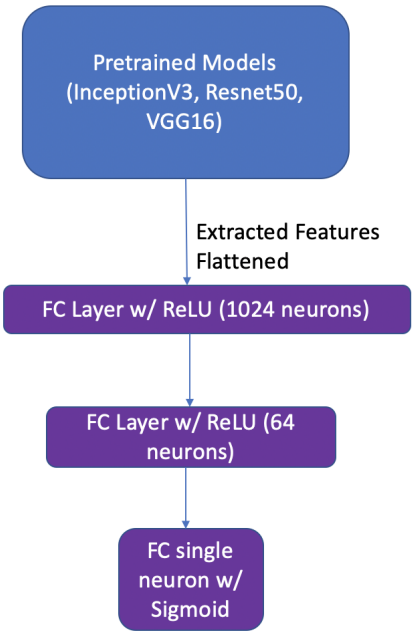

Figure 3: CNN Model Architecture

The first modification to these models is that we removed the original fully connected layers from each of these architectures and attached custom fully connected layers on top of them in order to address the classification task at hand. Next, we determined at what depth we should take the output from these architectures to feed into the custom fully connected layers. These were the levels and corresponding feature dimensions that we took the output from for each architecture:- VGG16 : block5_pool(4,4,512), Resnet50 : conv5_block3_out(5, 5, 2048) and InceptionV3 : mixed7(7, 7, 768). They were obtained experimentally after observing which feature levels gave the best performance. The output from these layers was first flattened, then fed into a 3 layer fully-connected layer, with the first layer having 1024 neurons, second layer having 64 neurons and the last layer having 1 neuron. The first two fully connected layers have the ReLU nonlinearity activation function. Furthermore, the last layer single neuron has Sigmoid nonlinearity for the classification. Moreover, we tried both approaches for model training: fine-tuning the layers of the pretrained models as well as freezing the layers of the pretrained models. Both approaches seemed to produce similar performance, hence we decided to go with the second approach (freezing the layers of the pretrained models) since this approach ran much faster.

3 Procedure

Our current procedure for training the GAN involves running (and re-running) a training loop within Google Colab. Since we are personally using laptops, we can leverage Google Colab's free GPU to train our model with some restrictions. Since Google Colab kicks of the user if the user is idle of from a usage limit, a framework is needed to reliably save the model at a checkpoint after training is done in order to preserve the latest model. To do this, a wrapper class is created which contains the model we want to train and contains some special utilities for logging and tracking the training process. It additionally contains the framework to save and load a model from its latest checkpoint in order to resume training from where it last left off. This allows us to reliably train our model without worrying about loosing progress.

3.1 GAN Training Procedure

Our training procedure for our GAN is pretty straightforward, however there were multiple runs in an effort to make our GAN converge. We split our training data set into 3616 images of patients with COVID-19 and 3616 more images of patients without; this gives us a total of 7262 training images to train on. For data transformation, we convert the images to a resolution of (300, 300) and normalize the values such that they are between -1 and 1. An interesting thing that we do to help the GAN converge is to train the discriminator one step every K training steps that we train the generator. This helps prevent the discriminator from learning to fast which causes the generator to diverge and create bad and noisy results. In our case, we pick K to be 10 training steps across all of our runs. For our Loss function, we use a Binary Cross Entropy Loss (BCE); Binary Cross Entropy is used for determining if the image is real or fake (within the discriminator) which gets propagated to the generator part to make better images to fool the adversary. For each of runs, we used the following varying parameters to test and make the GAN converge:

1. **Run 1:** In this run, we use the Stochastic Gradient Descent (SGD) optimizer (as given by Khalifa M. [4]) with a learning rate of 0.01 and a momentum

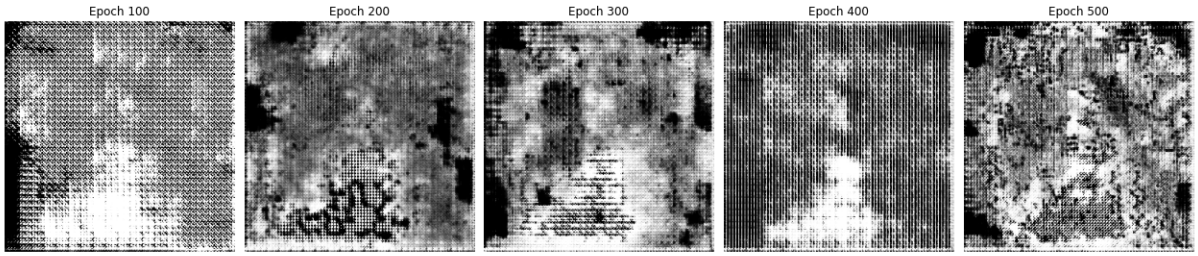

Figure 4: Run 1 of our GAN image generation shown per 100 epochs of training

of 0.9. We additionally set the mini-batch size to 4 so that the GAN could learn the differences between the images of patients that were infected with COVID-19 and the images of patients without.

2. **Run 2:** For this run, we modified a pre-processing step; after normalizing the training images, we downsample them by a factor of 2 to get an image resolution of 150x150 pixels. This supposedly should help train faster but at the cost of generating poorer quality images because some detail is lost. We additionally modify the learning rate by reducing it to 0.001. We use the same optimizer as the previous run: SGD optimizer. We also modify the number of filters per layer in our generation network such that the number of filters decreases by a factor of 2 across each layer (previously it was a constant 64 filters). The initial layer's filters starts at 256 filters and reduces by a factor of 2 every subsequent layer.

3. **Run 3:** In our last run, we keep our previous modifications. We downsample our training images by a factor of 2 to 150x150 pixels. We change our optimizer to the Adam optimizer, and set the respective learning rate to 2e-4 and momentum (beta 1) to 0.9. We also additionally add a 'CosineAnnealingWarmRestarts' learning rate scheduler and set the restart rate to 10 epochs.

3.2 CNN Training Procedure

In order to experiment with the different hyperparameters associated with each of the experiments, we primarily tried different learning rates and the kind of optimizer. For VGG16 we used the Adam optimizer with learning rate = 0.001 and 3 training epochs. For Resnet50 we used the Adam optimizer with learning rate = 0.0001 and 3 training epochs. Lastly, for InceptionV3, we used the Adam optimizer with learning rate = 0.00001 and 3 training epochs.

4 Results

4.1 GAN Results

Our results are not very good across all of our runs, however, we do observe improvements in the quality of the images. While the quality improves for each run, it is not at the level that we are looking where they would serve to augment an existing dataset (for production purposes). Although we still used our generated images for augmentation for the sake of completion. For our first run, we track the image generation every 10 or 20 training steps and log them. In the Figure above (Figure 4), we show the image generation every 100 epochs. You can see that the generator is somewhat attempting to learn the distribution of the chest and lung. You can faintly see the white figure, the dark spots of the lungs, and the stomach and liver organs at the bottom. However the results after training for so long is still not very good. One thing to note is that the images shown are for the non-covid lung scan input labels; we get very similar looking results for the covid lung scan input labels. In this run, the GAN diverges with the discriminator learning to fast and once that happens, the image fails to improve and training stops.

For our second run, we assumed that with a reduced input training resolution, the GAN would have an easier time learning. The results do show some improvement as shown in Figure 5. The overall shape of the body and lungs look familiar and the quality of the image looks much better than before however, there are still some odd artifacts that remain. We can see that the body that is generated in the images look much cleaner and smoother than what was shown in 'Run 1'. Along with the poor results, the GAN diverges with the discriminator learning too fast again. The images are obviously, still not of quality to be able to be used to augment our dataset. And similar to the previous run, we get similar looking images per epoch and per label.

Our last and final run gave better results than before, however, we still could not make quality results that would be valid to augment with. As shown in Figure 7, Adding the cosine annealing learning rate scheduler helps the generator learn better than the previous two runs did. In addition, we can see that there are

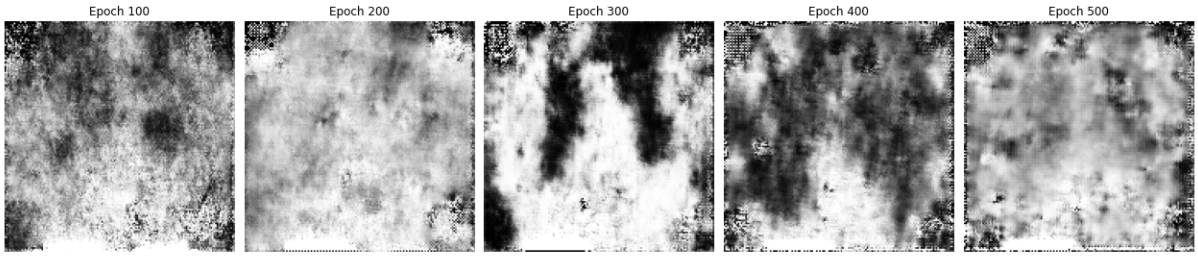

Figure 5: Run 2 of our GAN image generation shown per 100 epochs of training

nearly no blocky artifacts on the images. We suspect that the Adam optimizer helped a lot with this because the Adam optimizer is able to determine a smoother path to convergence within the loss landscape than the Stochastic Gradient Descent (SGD) optimizer. This run also ended in failure however, it was not because of the same reason as the previous two runs. Instead, the generator failed to continue learning and was resulting in a smooth uniform grey image after approximately 600 epochs of training. It continued to output the same style of images for the rest of training. There were no distinguishing features in the smooth, uniform grey image other than the border of the images which were only a couple of pixels thick and very noisy.

In all of our runs, we failed to make the GAN converge. The first two runs of our training procedure gave poor results with blocky artifacts which we assume to be attributed to the optimizer and learning rate. This was shown to be alleviated by changing the optimizer to the Adam optimizer and modifying the learning rate to be much smaller (2e-4). Despite these changes, we still could not get satisfactory results. In Figure 6, our loss plots are shown for run 2. The loss plots for run 1 are very similar with the discriminator loss approaching zero very quickly causing the GAN to diverge. Notice the fact that the initial training (at epoch 0 - green line) is very unstable for both the generator and the discriminator, showing big spikes and jumps before reducing and converging to a loss rate of around 0.75. This is relatively normal for GANs given that they have not been started on pre-trained weights. Since all of the weights are initially zero, the GAN outputs very noisy results without any structure which takes some time to find some structure in the dataset.

One may wonder why we didn't reduce the rate of training of the discriminator if our discriminator was training to fast (i.e. converging before our generator). Recall, rate of training refers to the number of training steps we take for the generator before we train a single step for the discriminator. The reason is because we did try smaller and larger discriminator training steps. In the case of smaller training steps, our discriminator obviously converged much faster giving us poor results and for large training steps, our generator would not learn any particular features. We approached the number of training steps to be 10 for our discriminator because it seemed the most stable out of the options that we have tried. For some perspective, we tried these steps for our discriminator: 5, 7, 9, 10, 12, 15, 20.

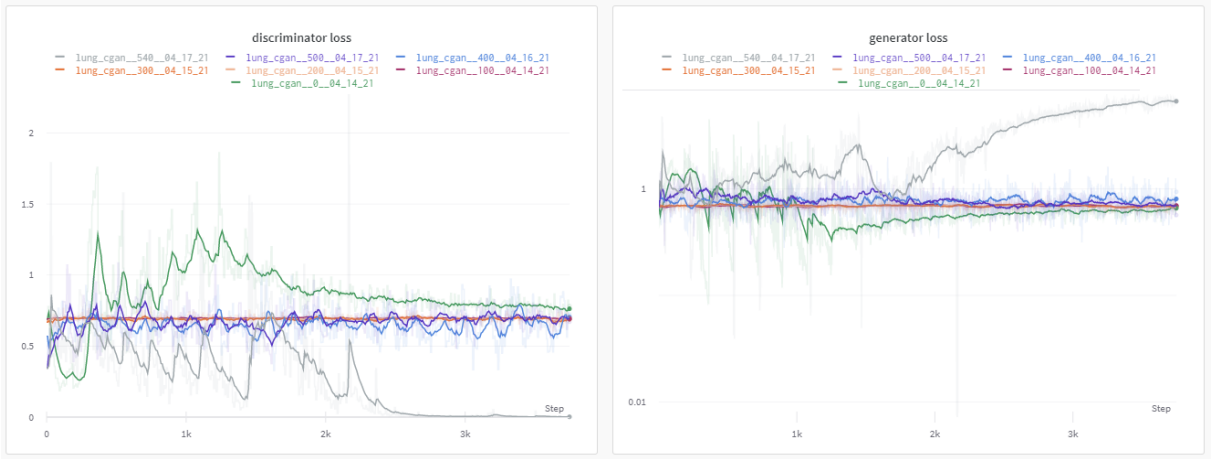

Figure 6: Smoothed loss plots of Run 2 showing the discriminator loss on the left and the generator loss on the right. The legend describes the different epochs which is denoted by **lung_cgan__<epoch>__<date>**. The plot shows that at epoch 540 (grey line), the discriminator learns faster than the generator where the discriminator loss converges to zero while the generator loss increases (inversely proportional to the discriminator loss).

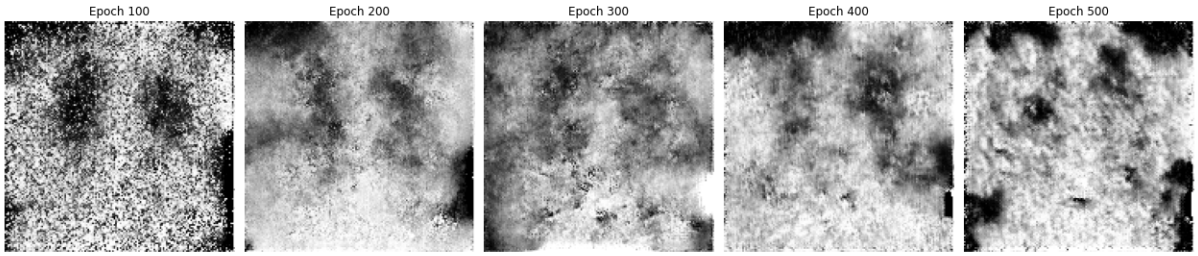

Figure 7: Run 3 of our GAN image generation shown per 100 epochs of training. You can see much smoother features which are much better looking than the previous runs which has significant blocky artifacts on the generated images

4.2 CNN Results

Below, is a table which contains the performance of the three models on the original dataset (Table 8). The table following contains the performance of the three models on the original dataset augmented with the GAN generated images (Table 9). From our results, we see that there is actually a slight improvement in all accuracy metrics for all of the models tested (except for the test accuracy for Resnet50). We can conclude that the GAN did learn some useful information which allowed the classifier to improve its accuracy since 2/3 of the models had their test accuracy improved.

Model	Training Accuracy	Validation Accuracy	Test Accuracy
VGG16	0.9617	0.9516	**0.9537**
Resnet50	0.7333	0.7621	0.7697
InceptionV3	0.9742	0.9371	0.9391

Figure 8: Baseline Performance Comparison

Model	Training Accuracy	Validation Accuracy	Test Accuracy
VGG16	0.9683	0.9648	**0.9675**
Resnet50	0.7943	0.7752	0.7644
InceptionV3	0.9797	0.9518	0.9529

Figure 9: Performance Comparison after Data Augmentation

5 Dataset and Resources

For our data we are using several existing datasets which can be found in the respective links in the **References** section from GitHub and Kaggle [1, 3, 2, 5, 4]. We used the Pytorch python libraries as a machine learning framework for our GAN architecture and Tensorflow-Keras for our classification architecture. No external code was used to perform these experiments and all the tools to measure and validate were built using pytorch's existing tools and numpy's library tools. Some code from tensorflow library was used for the pretrained models at the classification step.

An example of our dataset with some labels can be shown in the figure above (Figure 10). This specific set of images comes from the *COVID-19 Radiography Database* ([3]). The set consists of 3616 COVID-19 images, 10k of Normal images, and many more images (which are not relevant at the moment - they consist of viral pneumonia and lung opacity scans). Our data is of dimension (299, 299, 1) ordered by height, width, and channel. Our image is shown with a color map but they are essentially grayscale images (i.e, single channel). One interesting thing to note is that a short look at the images can give a quick conclusion that if the patient's lungs are cloudy, then the patient has some abnormal lung condition and if the patient's lungs are clear, they are normal. However, it should be stated (even if obvious) that cloudy lungs does not equate that the patients have COVID-19 but rather that it could be a different underlying condition. You may additionally notice that the scales vary between each image which presents another difficulty in training; it simply means that the GAN must learn generate scale-invariant features which is a difficult task in itself.

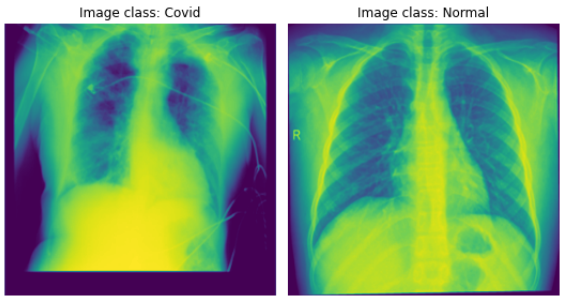

Figure 10: Sample of two patients' chest x-rays with different conditions

6 Conclusion

Overall, we found that training GANs are extremely difficult with a variety of hyperparameters that much be chosen very carefully in order for the generator and its adversary to learn from each other. In our results, we found that there was a lot of variance between our

training results. We think that we may have not had enough data to train the GAN because our task involved making a generator to learn the difference and distribution between images of patients with COVID-19 and without. We may have been conservative by trying to get the GAN to learn those features with just 7232 images. Another reason that we believe the GAN had difficulty is that there is a high amount of variance in the lung scan dataset; take for example Figure 10. The images are not aligned very well and the scale of the lungs are different. We think that the GAN was having trouble learning with these dataset issues. We additionally found that some of the lung scans were flipped sideways which may have further impeded our training. Moreover, although the GAN did not produce visually good quality images, it did seem that it was able to encode some meaningful data because we observed improved performance on two out of three models after augmenting them with the GAN generated data. An open ended question is if the improvement in performance that we observed was worth the effort that was put into generating the 2000 new images (1000 for each class) from the GAN. The relatively minor performance improvements might not justify the efforts of training a GAN to generate the additional images.

As mentioned previously, our body of work is presented through a video-presentation provided on youtube [6]. And our code and augmented dataset are additionally hosted on google drive and github [7].

7 Member Roles and Strategy

Our code is being shared by Google Colab and by GitHub. We meet on weekends to discuss project status and goals/deliverables for the next week, any issues that may have arisen, and the total project plan. The data is being shared through google drive. Both of us worked closely for data cleaning, model architecture, and training. However, the responsibilities were broadly split into the following items:

1. **Varun Govind:** (Weight:0.5) Data cleaning, Data retrieval, Model architecture for GAN, Training GAN.
2. **Vyom Thakkar:** (Weight:0.5) Model architecture for Classifier, Training Classifier.
3. **Both Members:** Performance/Result Analysis, Report Write-up and Evaluations.

For the purposes of this final report, nothing has changed although the member roles for training specific model architectures have been swapped (the changes have been appended above). Additionally, while one part of the model did not work, we still proceeded to train a classifier with the original dataset and with the augmented dataset to test how the augmentation affects the classifier accuracy.

References

[1] J. P. Cohen, P. Morrison, L. Dao, K. Roth, T. Q. Duong, M. Ghassemi, Covid-19 image data collection: Prospective predictions are the future, arXiv 2006.11988.
URL https://github.com/ieee8023/covid-chestxray-dataset

[2] Kaggle, Chest x-rays datasets compilation (2021).
URL https://www.kaggle.com/c/vinbigdata-chest-xray-abnormalities-detection/discussion/208035

[3] Kaggle, Covid-19 radiography dataset (2021).
URL https://www.kaggle.com/tawsifurrahman/covid19-radiography-database

[4] N. E. M. Khalifa, M. H. N. Taha, A. E. Hassanien, S. Elghamrawy, Detection of coronavirus (covid-19) associated pneumonia based on generative adversarial networks and a fine-tuned deep transfer learning model using chest x-ray dataset (2020).

[5] M. Loey, F. Smarandache, N. E. M. Khalifa, Within the lack of chest covid-19 x-ray dataset: A novel detection model based on gan and deep transfer learning, Symmetry 12 (4).
URL https://www.mdpi.com/2073-8994/12/4/651

[6] V. Thakkar, V. Govind, Ece 549 spring 2021 final project (covid x-ray data augmentation using gan and cnn classification) (2021).
URL https://www.youtube.com/watch?v=2Eys-8-ooqM

[7] V. Thakkar, V. Govind, Github code repository - ece 549 final project (covid x-ray data augmentation using gans) (2021).
URL https://github.com/varung2/ECE_549_final_project