CS551G Assessment 1 - Data Mining and Visualisation

Valerija Holomjova
Student ID: 52095492
University of Aberdeen
v.holomjova.20@abdn.ac.uk

1. CREATE SYNTHETIC COVID-19 X-RAY IMAGES WITH CONDITIONAL GENERATIVE ADVERSARIAL NETWORKS

1.1 Building and training a cGAN

This subtask involved creating scripts to upload and preprocess the provided dataset, build a conditional Generative Adversarial Network (cGAN), and generate images using pre-trained generator models. The 'code.ipynb' file that was submitted with this assignment contains all the scripts that were created for the assignment (as well as their console outputs). Figure 20 in Appendix B shows all the libraries that were imported.

1.1.1 Importing and preprocessing the dataset

At the start of the task, the dataset .zip file was uploaded to Google Colab and unzipped using the 'files.upload()' and '!unzip' command. The dataset contains 200 chest x-ray images where 100 images are labelled 'Normal' and the other 100 are labelled 'Covid'. Thus, there is an equal class distribution.

The code snippet shown in Figure 21 in Appendix B illustrates how the data was preprocessed. The code iterates through each image in a directory of images and each image is resized to a size of 128 x 128, normalized between a range of -1 to 1, and expanded to a third dimension to include the channel. The images were resized as they had large dimensions, hence their dimensions had to be reduced due to GPU memory constraints. Moreover, all images had different dimensions, hence resizing was the optimal choice as it would avoid information loss over methods like cropping or padding. Each image was also assigned a class label (1 for 'Covid' and 0 for 'Normal') and the data (image and labels) was stored in a dictionary for future use. Figure 1 shows an example of images before and after preprocessing.

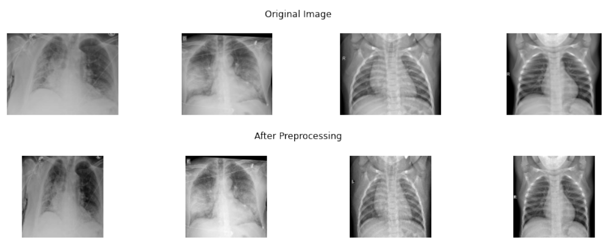


Figure 1: Example of images before (top) and after (bottom) preprocessing.

1.1.2 Building and training the cGAN model

The cGAN model was built by modifying the 'cgan.py' file that was provided for this assignment. The 'Init()' and 'sample_images()' functions in the class were only modified slightly (eg. changing the image size variables) to be compatible with the dataset. However, the 'train()' function and generator and discriminator models were modified much more as explained below. The 'Init()' function instantiates the cGAN model by creating the discriminator and generator models, then setting the discriminator model weights as not trainable (as they are trained separately) and passing the input labels and output images of the generator as input to the discriminator to determine whether the output image is real or fake. The cGAN model is compiled using the Adam optimizer and binary cross-entropy loss. The 'sample_images()' function outputs 6 sample images from the generator model every 20 epochs during training.

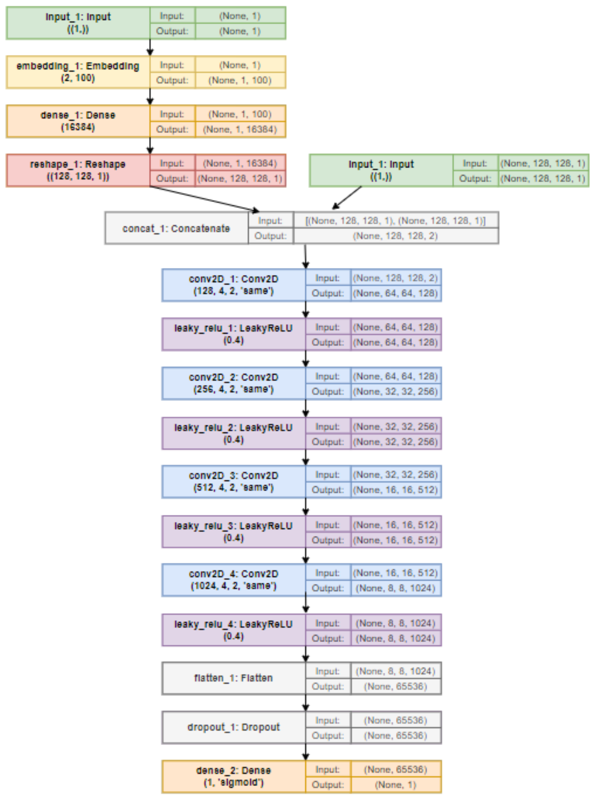

input_1: Input ((1,))
Input: (None, 1)
Output: (None, 1)
embedding_1: Embedding (2, 100)
Input: (None, 1)
Output: (None, 1, 100)
dense_1: Dense (16384)
Input: (None, 1, 100)
Output: (None, 1, 16384)
reshape_1: Reshape ((128, 128, 1))
Input: (None, 1, 16384)
Output: (None, 128, 128, 1)
input_1: Input ((1,))
Input: (None, 128, 128, 1)
Output: (None, 128, 128, 1)
concat_1: Concatenate
Input: [(None, 128, 128, 1), (None, 128, 128, 1)]
Output: (None, 128, 128, 2)
conv2D_1: Conv2D (128, 4, 2, 'same')
Input: (None, 128, 128, 2)
Output: (None, 64, 64, 128)
leaky_relu_1: LeakyReLU (0.4)
Input: (None, 64, 64, 128)
Output: (None, 64, 64, 128)
conv2D_2: Conv2D (256, 4, 2, 'same')
Input: (None, 64, 64, 128)
Output: (None, 32, 32, 256)
leaky_relu_2: LeakyReLU (0.4)
Input: (None, 32, 32, 256)
Output: (None, 32, 32, 256)
conv2D_3: Conv2D (512, 4, 2, 'same')
Input: (None, 32, 32, 256)
Output: (None, 16, 16, 512)
leaky_relu_3: LeakyReLU (0.4)
Input: (None, 16, 16, 512)
Output: (None, 16, 16, 512)
conv2D_4: Conv2D (1024, 4, 2, 'same')
Input: (None, 16, 16, 512)
Output: (None, 8, 8, 1024)
leaky_relu_4: LeakyReLU (0.4)
Input: (None, 8, 8, 1024)
Output: (None, 8, 8, 1024)
flatten_1: Flatten
Input: (None, 8, 8, 1024)
Output: (None, 65536)
dropout_1: Dropout
Input: (None, 65536)
Output: (None, 65536)
dense_2: Dense (1, 'sigmoid')
Input: (None, 65536)
Output: (None, 1)

Figure 2: Modified discriminator model in the cGAN class.

Figure 22 in Appendix B shows the modified 'train()' function. The 'train()' function in the cGAN class was altered so that half batches (batch size/2) were used instead of the batch size when training the discriminator and the full batch size is only used for training the generator (before it only considered the full batch size for both). It was also modified to train in batches at each epoch as well as save the generator model to a file once training is complete for future use. A function called 'generate_noise()' was also added to generate random noise and labels for the generator more efficiently.

Figure 2 shows the modified discriminator model and

Figure 23 in Appendix B has the corresponding code. The dense layers in the original discriminator model were changed to convolutional layers (with strides instead of pooling layers) as this seems to be a recommended approach from other research [1, 2, 3] and tutorials[1]. The discriminator model takes as input a label (left) and an image (right). The label is passed through an embedding, dense and reshape layer to be concatenated with the image input into a two-channel 128 x 128 feature map or image. Similar to Venu [1], the discriminator model uses four convolutional layers with a kernel size of 4 and stride of 2 interlaced with Leaky ReLu layers to downsample the image to a size of 8 x 8. The final output layer then uses a sigmoid activation function to determine whether the image is real or fake.

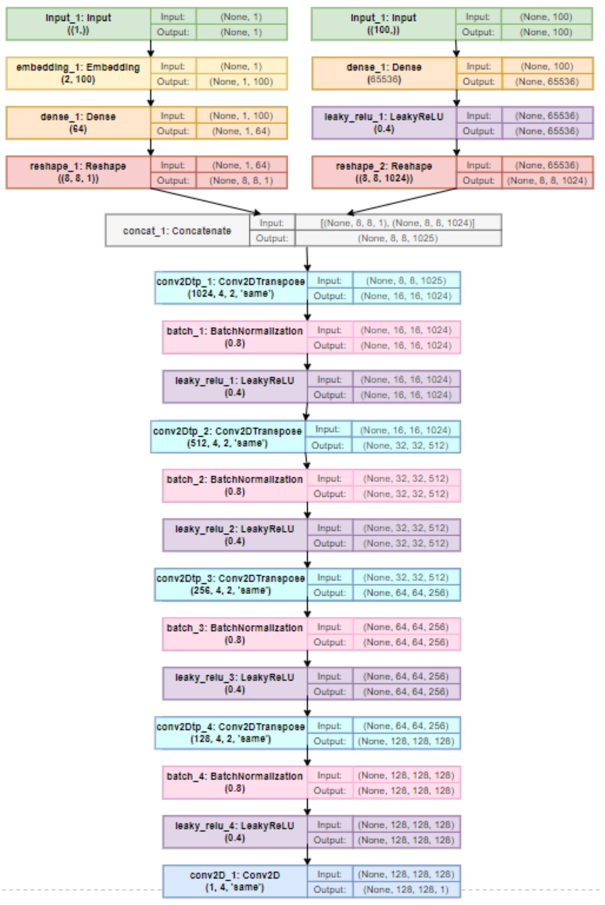

input_1: Input ((1,)) | Input: (None, 1) | Output: (None, 1)
embedding_1: Embedding (2, 100) | Input: (None, 1) | Output: (None, 1, 100)
dense_1: Dense (64) | Input: (None, 1, 100) | Output: (None, 1, 64)
reshape_1: Reshape ((8, 8, 1)) | Input: (None, 1, 64) | Output: (None, 8, 8, 1)
input_1: Input ((100,)) | Input: (None, 100) | Output: (None, 100)
dense_1: Dense (65536) | Input: (None, 100) | Output: (None, 65536)
leaky_relu_1: LeakyReLU (0.4) | Input: (None, 65536) | Output: (None, 65536)
reshape_2: Reshape ((8, 8, 1024)) | Input: (None, 65536) | Output: (None, 8, 8, 1024)
concat_1: Concatenate | Input: [(None, 8, 8, 1), (None, 8, 8, 1024)] | Output: (None, 8, 8, 1025)
conv2Dtp_1: Conv2DTranspose (1024, 4, 2, 'same') | Input: (None, 8, 8, 1025) | Output: (None, 16, 16, 1024)
batch_1: BatchNormalization (0.8) | Input: (None, 16, 16, 1024) | Output: (None, 16, 16, 1024)
leaky_relu_1: LeakyReLU (0.4) | Input: (None, 16, 16, 1024) | Output: (None, 16, 16, 1024)
conv2Dtp_2: Conv2DTranspose (512, 4, 2, 'same') | Input: (None, 16, 16, 1024) | Output: (None, 32, 32, 512)
batch_2: BatchNormalization (0.8) | Input: (None, 32, 32, 512) | Output: (None, 32, 32, 512)
leaky_relu_2: LeakyReLU (0.4) | Input: (None, 32, 32, 512) | Output: (None, 32, 32, 512)
conv2Dtp_3: Conv2DTranspose (256, 4, 2, 'same') | Input: (None, 32, 32, 512) | Output: (None, 64, 64, 256)
batch_3: BatchNormalization (0.8) | Input: (None, 64, 64, 256) | Output: (None, 64, 64, 256)
leaky_relu_3: LeakyReLU (0.4) | Input: (None, 64, 64, 256) | Output: (None, 64, 64, 256)
conv2Dtp_4: Conv2DTranspose (128, 4, 2, 'same') | Input: (None, 64, 64, 256) | Output: (None, 128, 128, 128)
batch_4: BatchNormalization (0.8) | Input: (None, 128, 128, 128) | Output: (None, 128, 128, 128)
leaky_relu_4: LeakyReLU (0.4) | Input: (None, 128, 128, 128) | Output: (None, 128, 128, 128)
conv2D_1: Conv2D (1, 4, 'same') | Input: (None, 128, 128, 128) | Output: (None, 128, 128, 1)

Figure 3: Modified generator model in the cGAN class.

Figure 3 shows the modified generator model and Figure 24 in Appendix B has the corresponding code. The generator was constructed using a similar architecture to that of Waheed et al. [3]. The generator takes as input a label (left) and noise (right). The noise is passed through a dense, Leaky ReLU layer and reshape layer to get many copies of low-resolution versions of the output image. The label is passed through the same layers as the discriminator before and then concatenated with the low-resolution images into (8 x 8 x 1025) feature maps. These images are passed through four convolutional transpose layers of kernel size 4 and stride of 2 interlaced with Batch Normalization layers and Leaky ReLU layers to upsample them to a size of (128 x 128 x 128). The final output layer uses the 'tanh' activation function to output an image of shape (128 x 128 x 1).

[1]https://machinelearningmastery.com/how-to-develop-a-conditional-generative-adversarial-network-from-scratch/

Figure 4 shows the code that was used to train the final cGAN model using hyperparameters determined in subtask 1.2. A larger amount of epochs (500 instead of 300) was used, however, to try to generate better results similar to other paper [1]. Figure 25 in Appendix B shows the code that was used to generate 50 images using a pre-trained generator model. This code was adapted from the 'sample_images()' function in the 'cgan.py' file. It consists of two functions 'generate_images()' and 'plot_images()'. The 'generate_images()' function loads a pre-trained generator model then generates random noise and labels (based on the number of images needed) and passes them into the model (using the 'predict()' function) to generate a specified number of images. These images are then passed into the 'plot_images()' which are then plotted as shown in Figure 7. In the plot, the class label (Covid = 1 or Normal = 0) of each image is displayed above it. The same functions were altered and used to generate the images in Figures 5 and 6 to compare the output and performance of the final generator model to images of the actual dataset.

```
# load dictionary of preprocessed data
data = pickle.load(open("preprocessed_data.pkl",
    ↪ "rb"))

# example of training process
cgan = CGAN(data) # learning rate 0.0002
cgan.train(500, 32, 'final_model') # epochs 500
    ↪ and batch size 32
```

Figure 4: Training the final generator model.

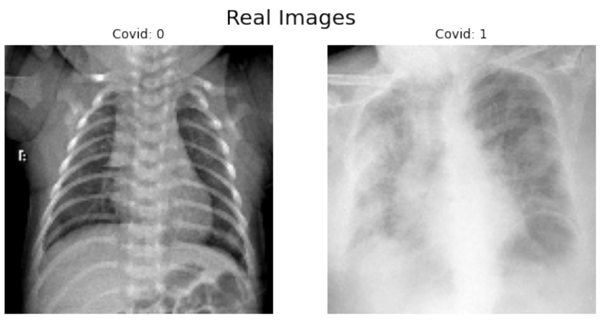


Figure 5: Sample of images from the real dataset.

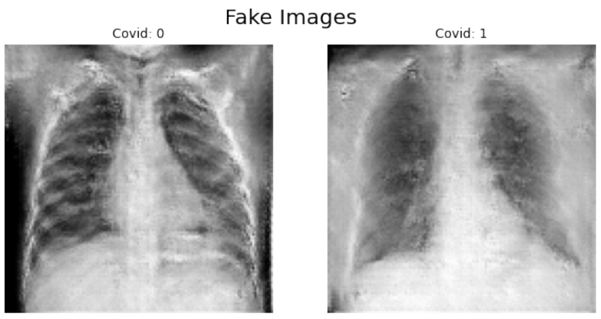


Figure 6: Sample of images generated from the final generator model.

As shown by Figures 5, 6 and 7, the final generator model seemed to generate images with similar appearance to chest x-ray images. By comparing Figures 5 and 6, one can see that the generator was able to produce an image such that details such as the spine and ribs are visible, and it is noticeable where the lungs are (the darker

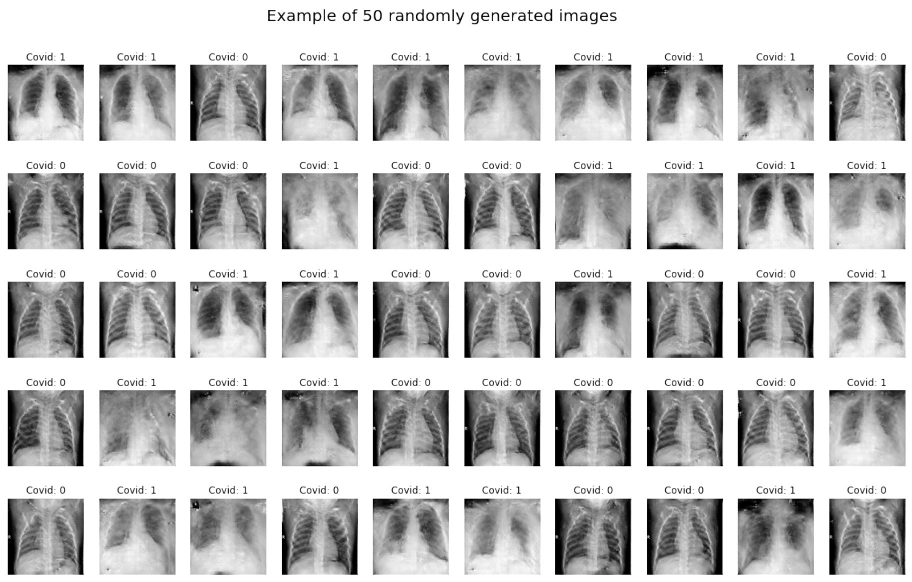


Figure 7: Generating 50 images using the final generator model.

region). By examining Figures 6 and 7, one can notice that images labelled with 'Covid' or '1' were generated much more 'foggy' or 'cloudy' compared to the ones labelled 'Normal' or '0'. This is because x-ray images with 'Covid' had similar properties as seen in Figure 5 as Covid can cause fluid in the lungs which would emit a cloudy appearance on an x-ray. On the other hand, the figures show that there is still more room for improvement to generate higher quality images which could be done by trying to improve the generator's architecture or input such as optimizing the noise input (or latent vector representation) [4].

1.2 Exploring different hyperparameters and outputting generated images

In this section, I explored different hyperparameters and their impact on the generator models performance. Figure 26 in Appendix B shows the code that was used to train different models under different hyperprameters. For this assignment, 9 different models were constructed having different **epochs** (100, 200, 300), **batch sizes** (16, 32, 64) and **learning rates** (0.002, 0.0002, 0.00002) for the Adam optimizer. Figures 17, 18, 19 in Appendix A shows tables of images (each with 1 Normal image on the left and 1 Covid image on the right) generated by the different models during their final stages of training.

The figures show that models that were trained for 300 epochs had the clearest images than those trained for 200 epochs followed by those trained for 100 epochs which is justified due to more training time. Additionally, in all figures, all models which had a learning rate of 0.002 or 0.0002 generated very unclear images compared to a learning rate of 0.0002. In fact, most of the images having a learning rate of 0.002 were barely recognizable as x-ray images except for one case in Figure 18. The figures also show that most of the models trained on batch sizes of 16 and 32 had more detailed images than those having a batch size of 64. Hence, it is apparent that the best images were generated by the models trained at 300 epochs with a learning rate of 0.0002 and batch size of 16 and 32, which are the hyperparameters I will be using to train my final model. Since other research [2] used a batch size of 32 to train their model, I used the same to train my final model which was mentioned in Subtask 1.1.

2. DETECT COVID-19 FROM CHEST X-RAY IMAGES USING PRE-TRAINED NETWORKS

2.1 Building a binary classification model using transfer learning

In this section, a binary classification model that classifies Covid-19 and Normal x-ray images was created via transfer learning using a pre-trained ResNet-50 model [5]. The code that was created for this subtask is shown in Figure 27 in Appendix B.

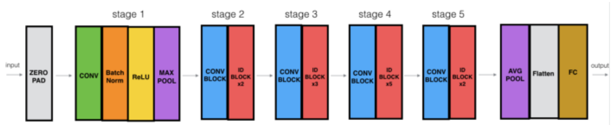


Figure 8: Architecture of the ResNet-50 model (original source from this link[2]).

The code loads the pre-processed dataset and splits into 80% training and 20% testing using the 'train_test_split()' function from Scikit-learn[3] with stratification (stratify=y) so that the data is split with equal class labels and shuffled. Next, a pre-trained ResNet-50 model is loaded from Keras[4] with 'imagenet' weights and the last fully connected layer removed (by setting include_top=False). This consists of the average pooling layer, flatten layer and final classification layer as shown in Figure 8. Next, each layer in the pre-trained ResNet-50 model is frozen by setting each layer as 'trainable=False' so that their weights are not updated during training. Then, the code takes the output of the last layer of the ResNet-50 model and passes it as input to a new set of layers which includes a 'Flatten' layer, followed by a 'Dense' layer, then a 'Dropout' layer and another final 'Dense' layer. The two 'Dense' layers are trainable and the 'Dropout' layer is used for regularization. A global pooling layer could have been used instead of a 'Flatten' layer, however, since the model already generated good results I decided not to change its architecture further. The final 'Dense' layer uses a sigmoid activation function to classify the image into a bi-

[3]https://scikit-learn.org/stable/

[4]https://keras.io/api/applications/

nary label (1 if Covid-19 is present or 0 if the x-ray is normal) which is then outputted from the model.

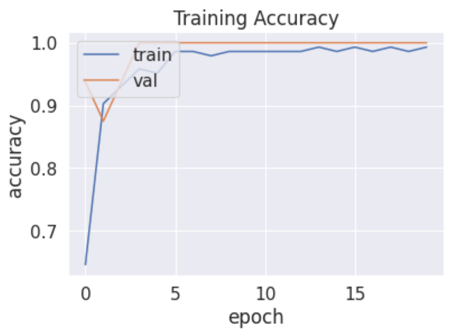


Figure 9: Training accuracy of the binary classification model in Subtask 2.1.

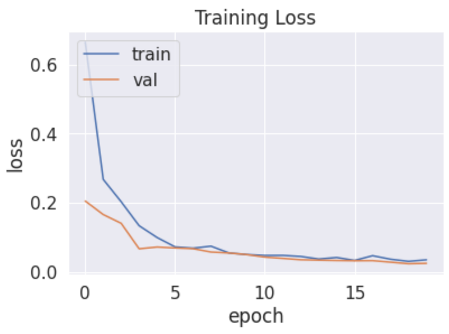
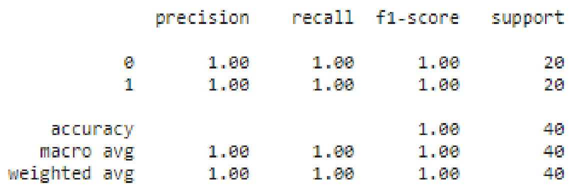


Figure 10: Training loss of the binary classification model in Subtask 2.1.

The new model is then compiled using an Adam optimizer and binary cross-entropy as the loss function (for binary classification tasks). The model is then trained using 20 epochs, a batch size of 16, a validation split of 10% and an early stopping callback implemented to cease training once the loss of the model stops decreasing. The training accuracy and training loss is also plotted as shown in Figures 9 and 10. After the model is trained, it is evaluated on the test set (using the 'predict()' function from Keras) and the classification report and confusion matrix of its performance on determining the correct labels of the test set is displayed as shown in Figure 11 and 12 using functions from Scikit-learn.



```
              precision    recall  f1-score   support

           0       1.00      1.00      1.00        20
           1       1.00      1.00      1.00        20

    accuracy                           1.00        40
   macro avg       1.00      1.00      1.00        40
weighted avg       1.00      1.00      1.00        40
```

Figure 11: Classification report of the binary classification model in Subtask 2.1.

Figures 9 and 10 shows that the model was able to train successfully, and was able to obtain a training accuracy of 0.995 and training loss of 0.032 after 15 epochs. Based on the figures, the model was able to train very quickly and started to converge at 5 epochs illustrating the benefits of transfer learning as it accelerates the training of neural networks. Figures 11 and 12 shows that the model performed very well on the test set and achieved a perfect score with an accuracy of 100% and F-measure of 100% from the classification report, as well as 0 false negatives

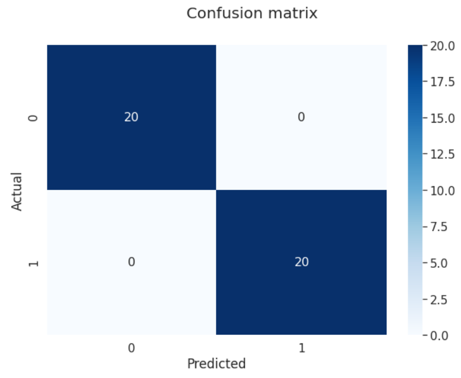


Figure 12: Confusion matrix of the binary classification model in Subtask 2.1.

and false positives. This does not necessarily mean that the model is perfect and could be as a result of over-fitting to the dataset or not having enough varying samples in the test set. Hence, more testing and evaluation should be carried out on the model on more real-word examples as well as introducing some form of explainable AI to determine whether the model is classifying the x-ray images correctly.

2.2 Building a binary classification model using transfer learning (with 50 generated Covid images)

In this subtask, a script was created to generate 50 'Covid' labelled images from the final generator model mentioned in Subtask 1.1, and using them to create a new dataset with 50 randomly sampled 'Covid' images from the original dataset, as well as the original 100 'Normal' labelled x-ray images. This new dataset was then used to train a new binary classification model (using the exact same method proposed in Subtask 2.1) and then displays the training accuracy and loss of the model (Figures 13 and 14), as well as the classification report and confusion matrix (Figures 15 and 16) of the model when evaluated on the test set as done in Subtask 2.1. I have displayed the code that was created for creating the new dataset in Figure 28 in Appendix B. I did not include the rest of the code (building, training and evaluating the model again) in the Figure as it is identical to that of Subtask 2.1.

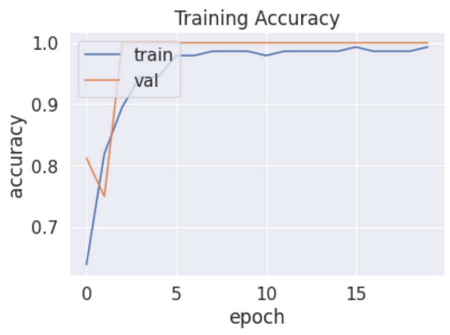


Figure 13: Training accuracy of the binary classification model in Subtask 2.2 (with modified dataset).

In the code, the pre-processed dataset is loaded and 50 random 'Covid' labelled images are selected and stored in a variable 'X_half_covid_only'. Next, the final model mentioned in Subtask 1.1, is loaded and used to generate 50 random 'Covid' x-ray images. The 50 generated images are then concatenated to the 50 randomly selected ones of earlier which are then concatenated to 100 of the original 'Normal' images of the dataset. It is ensured that labels are generated and given appropriate labels to their corresponding image.

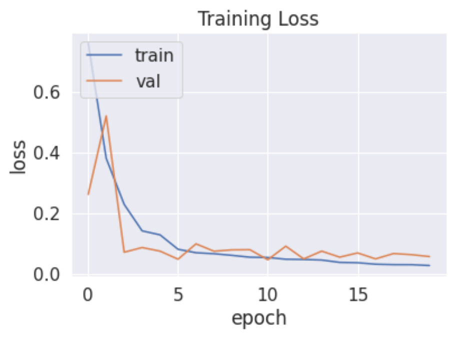
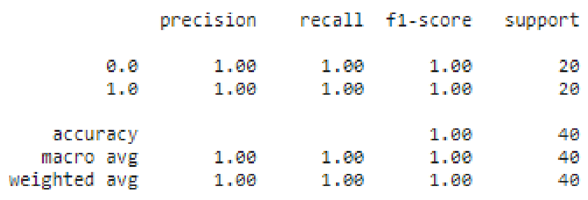


Figure 14: Training loss of the binary classification model in Subtask 2.2 (with modified dataset).



```
              precision    recall  f1-score   support

         0.0       1.00      1.00      1.00        20
         1.0       1.00      1.00      1.00        20

    accuracy                           1.00        40
   macro avg       1.00      1.00      1.00        40
weighted avg       1.00      1.00      1.00        40
```

Figure 15: Classification report of the binary classification model in Subtask 2.2 (with modified dataset).

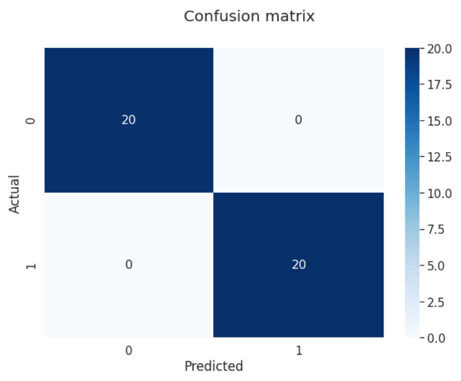


Figure 16: Confusion matrix of the binary classification model in Subtask 2.2 (with modified dataset).

Similar to Subtask 2.1, Figures 13 and 14 shows that the model was able to train successfully, and was able to obtain a training accuracy of 0.995 and training loss of 0.030 after 20 epochs and converged around an epochs of 5. Figures 11 and 12 shows that the model performed very well on the test set and achieved a perfect score with an accuracy of 100% and F-measure of 100% from the classification report, as well as 0 false negatives and false positives. This shows that the injection of fake images did not negatively impact the final results and were distinguishable as real Covid images, however, as mentioned in Subtask 2.1 this does not mean the classifier is perfect and should be evaluated further.

3. REFERENCES

[1] S. K. Venu, "Evaluation of deep convolutional generative adversarial networks for data augmentation of chest x-ray images," *arXiv preprint arXiv:2009.01181*, 2020.

[2] S. Menon, J. Galita, D. Chapman, A. Gangopadhyay, J. Mangalagiri, P. Nguyen, Y. Yesha, Y. Yesha, B. Saboury, and M. Morris, "Generating realistic covid19 x-rays with a mean teacher+ transfer learning gan," *arXiv preprint arXiv:2009.12478*, 2020.

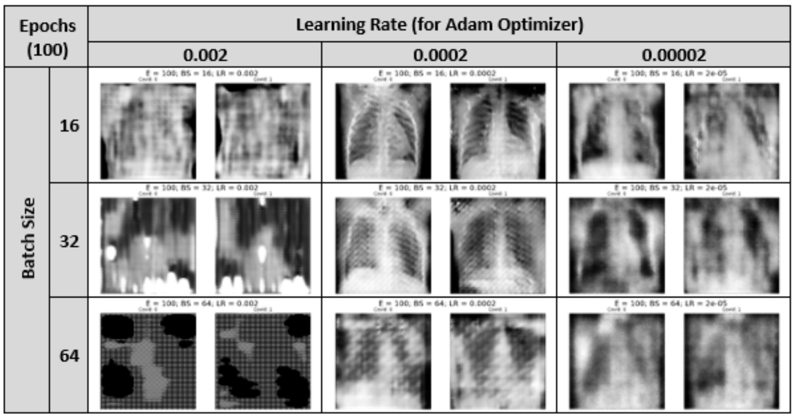
[3] A. Waheed, M. Goyal, D. Gupta, A. Khanna, F. Al-Turjman, and P. R. Pinheiro, "Covidgan: data augmentation using auxiliary classifier gan for improved covid-19 detection," *Ieee Access*, vol. 8, pp. 91916–91923, 2020.

[4] S. Morvan, C. Treal, and J. Sorette, "Methods applicable on cgan for improving performance related to image translation applications," 2019.

[5] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

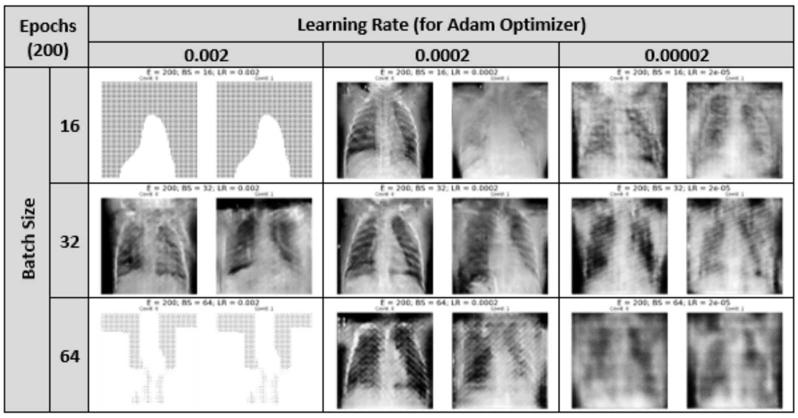
APPENDIX

A. FIGURES



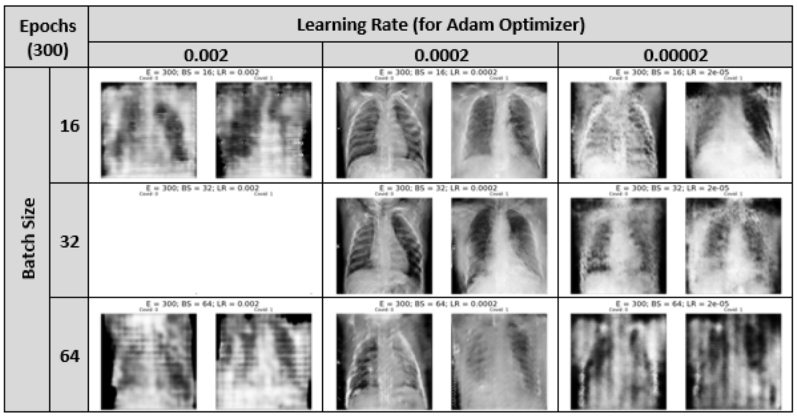
Epochs (100)
Learning Rate (for Adam Optimizer)
0.002
0.0002
0.00002
Batch Size
16
32
64
E = 100; BS = 16; LR = 0.002
E = 100; BS = 16; LR = 0.0002
E = 100; BS = 16; LR = 2e-05
E = 100; BS = 32; LR = 0.002
E = 100; BS = 32; LR = 0.0002
E = 100; BS = 32; LR = 2e-05
E = 100; BS = 64; LR = 0.002
E = 100; BS = 64; LR = 0.0002
E = 100; BS = 64; LR = 2e-05
Covid 0
Covid 1

Figure 17: Images generated by models that were trained for 100 epochs.



Epochs (200)
Learning Rate (for Adam Optimizer)
0.002
0.0002
0.00002
Batch Size
16
32
64
E = 200; BS = 16; LR = 0.002
E = 200; BS = 16; LR = 0.0002
E = 200; BS = 16; LR = 2e-05
E = 200; BS = 32; LR = 0.002
E = 200; BS = 32; LR = 0.0002
E = 200; BS = 32; LR = 2e-05
E = 200; BS = 64; LR = 0.002
E = 200; BS = 64; LR = 0.0002
E = 200; BS = 64; LR = 2e-05
Covid 0
Covid 1

Figure 18: Images generated by models that were trained for 200 epochs.



Epochs (300)
Learning Rate (for Adam Optimizer)
0.002
0.0002
0.00002
Batch Size
16
32
64
E = 300; BS = 16; LR = 0.002
E = 300; BS = 16; LR = 0.0002
E = 300; BS = 16; LR = 2e-05
E = 300; BS = 32; LR = 0.002
E = 300; BS = 32; LR = 0.0002
E = 300; BS = 32; LR = 2e-05
E = 300; BS = 64; LR = 0.002
E = 300; BS = 64; LR = 0.0002
E = 300; BS = 64; LR = 2e-05
Covid 0
Covid 1

Figure 19: Images generated by models that were trained for 300 epochs.

B. CODE SNIPPETS

```
from __future__ import print_function, division
from keras.datasets import mnist
from keras.layers import Input, Dense, Reshape, Flatten, Dropout, multiply, Concatenate
from keras.layers import BatchNormalization, Activation, Embedding, ZeroPadding2D, Conv2DTranspose
from keras.layers.advanced_activations import LeakyReLU
from keras.layers.convolutional import UpSampling2D, Conv2D
from keras.models import Sequential, Model
from keras.optimizers import Adam
from keras.models import load_model
from keras.applications.resnet50 import ResNet50
from keras.callbacks import EarlyStopping
import pickle
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from numpy import asarray
import cv2
import os
from google.colab import files
import pickle
import random
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, confusion_matrix, plot_confusion_matrix
```

Figure 20: Installing the necessary libraries in Google Colab.

```
xray_covid_path = 'DMV_Assess_1_Covid-19_Dataset/Covid-19'
xray_normal_path = 'DMV_Assess_1_Covid-19_Dataset/Normal'

# preprocesses all images in a given directory and returns the preprocessed images
def preprocess_images(path):
  prep_imgs = [] # preprocessed images
  width, height = 128, 128

  for filename in os.listdir(path):
      img = cv2.imread(path + '/' + filename, 0) # read grayscale image
      # resize image to size (128 x 128)
      img = cv2.resize(img, (width, height), interpolation=cv2.INTER_NEAREST)
      # normalize image between range -1 - 1
      img = (img.astype(np.float32) - 127.5) / 127.5
      # add a channel - from shape (128 x 128) to (128 x 128 x 1)
      img = img.reshape(width, height, 1)

      # add image to list of preprocessed images
      prep_imgs.append(img)
  return np.array(prep_imgs)

# get preprocessed images and assign them their correct class labels
xray_covid_imgs = preprocess_images(xray_covid_path)
xray_covid_lbls = np.full(shape=len(xray_covid_imgs), fill_value=1, dtype=np.int)
xray_normal_imgs = preprocess_images(xray_normal_path)
xray_normal_lbls = np.full(shape=len(xray_normal_imgs), fill_value=0, dtype=np.int)

# saving data to a dictionary
data = {
    'X_train': np.concatenate((xray_covid_imgs, xray_normal_imgs)),
    'y_train': np.concatenate((xray_covid_lbls, xray_normal_lbls))
}
# save to .pkl file
pickle.dump(data, open("preprocessed_data.pkl", "wb"))
```

Figure 21: Importing and pre-processing the dataset in Google Colab.

```
    # generate random noise (points in latent space) and integer class labels
    def generate_noise(self, n):
        # generate random points/noise and reshape into proper size
        noise = np.random.randn(self.latent_dim * n)
        noise = noise.reshape(n, self.latent_dim)
        # generate random labels
        labels = np.random.randint(0, self.num_classes, n)
        return [noise, labels]

    def train(self, epochs=100, batch_size=128, model_name='cgan_generator'):
        batch_per_ep = int(len(self.X_train)/batch_size) # batches per epoch
        half_batch = int(batch_size/2) # half batch size for training discriminator

        valid = np.ones((half_batch, 1))
        fake = np.zeros((half_batch, 1))
        fake_full = np.ones((batch_size, 1)) # for fake samples

        for e in range(epochs):
            for b in range(batch_per_ep):
                # ---------------------
                # Train Discriminator
                # ---------------------
                # select random half batch of 'real' images
                idx = np.random.randint(0, self.X_train.shape[0], half_batch) # choose random
                    ↪ instances
                real_imgs, labels_real = self.X_train[idx], self.y_train[idx] # select images and
                    ↪ labels
                # select half batch of 'fake' images
                noise_fake, labels_fake = self.generate_noise(half_batch)
                # generate a half batch of new images
                gen_imgs = self.generator.predict([noise_fake, labels_fake])

                # update the discriminator model weights
                d_loss_real = self.discriminator.train_on_batch([real_imgs, labels_real], valid)
                d_loss_fake = self.discriminator.train_on_batch([gen_imgs, labels_fake], fake)
                d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

                # ---------------------
                # Train Generator
                # ---------------------
                # get noise and random labels for generator
                noise, labels = self.generate_noise(batch_size)
                g_loss = self.combined.train_on_batch([noise, labels], fake_full)
            # summarize progress
            print("Epoch␣%d/%d␣Batch␣%d/%d␣[D␣loss:␣%.5f,␣acc:␣%.2f%%]␣[G␣loss:␣%.5f]" % (e+1, epochs
                ↪ , b+1, batch_per_ep, d_loss[0], 100*d_loss[1], g_loss))
            # save generated smaples at every x epochs
            if e % 20 == 0:
                self.sample_images(e, model_name)
        # save the generator model for future use/evaluation
        self.generator.save('models/'+model_name+'.h5')
```

Figure 22: Training the cGAN model in the cGAN class.

```
def build_discriminator(self):
    # parameters
    strides = 2 # stride to downsample (best practice)
    alpha = 0.2 # leaky relu slope/alpha (best practice)
    dropout = 0.4
    kernel_size = 4

    # conditional (discriminator)
    input_image = Input(shape=self.img_shape)
    input_label = Input(shape=(1,), dtype='int32')
    label_embedding = Embedding(self.num_classes, 100)(input_label)
    label_embedding = Dense(self.img_shape[0]*self.img_shape[1])(label_embedding)
    label_embedding = Reshape(self.img_shape)(label_embedding)
    model_input = Concatenate()([input_image, label_embedding])

    # discriminator
    model = Sequential()
    # downsample to 64x64
    model.add(Conv2D(128, kernel_size, strides=strides, padding='same'))
    model.add(LeakyReLU(alpha=alpha))
    # downsample to 32x32
    model.add(Conv2D(256, kernel_size, strides=strides, padding='same'))
    model.add(LeakyReLU(alpha=alpha))
    # downsample to 16x16
    model.add(Conv2D(512, kernel_size, strides=strides, padding='same'))
    model.add(LeakyReLU(alpha=alpha))
    # downsample to 8x8
    model.add(Conv2D(1024, kernel_size, strides=strides, padding='same'))
    model.add(LeakyReLU(alpha=alpha))
    # output layer
    model.add(Flatten())
    model.add(Dropout(dropout))
    model.add(Dense(1, activation='sigmoid'))

    classification = model(model_input)
    model.summary()
    discriminator = Model([input_image, input_label], classification)
    discriminator.compile(loss=['binary_crossentropy'], optimizer=self.optimizer, metrics=['
        ↪ accuracy'])
    return discriminator
```

Figure 23: Building the discriminator in the cGAN class.

```
    def build_generator(self):
        # parameters
        strides = 2 # stride to downsample (best practice)
        alpha = 0.2 # leaky relu slope/alpha (best practice)
        dropout = 0.4
        kernel_size = 4
        momentum = 0.8

        # conditional (generator)
        input_label = Input(shape=(1,), dtype='int32')
        label_embedding = Embedding(self.num_classes, self.latent_dim)(input_label)
        label_embedding = Dense(8*8)(label_embedding)
        label_embedding = Reshape((8, 8, 1))(label_embedding)

        input_noise = Input(shape=(self.latent_dim,))
        noise = LeakyReLU(alpha=alpha)(Dense(1024 * 8 * 8)(input_noise))
        noise = Reshape((8, 8, 1024))(noise)

        model_input = Concatenate()([noise, label_embedding])

        # generator
        model = Sequential()
        # upsample to 16x16
        model.add(Conv2DTranspose(1024, kernel_size, strides=strides, padding='same'))
        model.add(BatchNormalization(momentum=momentum))
        model.add(LeakyReLU(alpha=alpha))
        # upsample to 32x32
        model.add(Conv2DTranspose(512, kernel_size, strides=strides, padding='same'))
        model.add(BatchNormalization(momentum=momentum))
        model.add(LeakyReLU(alpha=alpha))
        # upsample to 64x64
        model.add(Conv2DTranspose(256, kernel_size, strides=strides, padding='same'))
        model.add(BatchNormalization(momentum=momentum))
        model.add(LeakyReLU(alpha=alpha))
        # # upsample to 128x128
        model.add(Conv2DTranspose(128, kernel_size, strides=strides, padding='same'))
        model.add(BatchNormalization(momentum=momentum))
        model.add(LeakyReLU(alpha=alpha))
        # output layer
        model.add(Conv2D(1, kernel_size, padding='same', activation='tanh'))

        output_img = model(model_input)
        model.summary()
        return Model([input_noise, input_label], output_img)
```

Figure 24: Building the generator model in the cGAN class.

```
def generate_images(model_name, r, c, title, latent_dim=100, num_classes=2):
    # load model
    model = load_model('models/'+model_name+'.h5')
    # generate random noise + labels
    noise = np.random.normal(0, 1, (r*c, 100))
    labels = np.random.randint(0, num_classes, r*c)
    # generate images
    imgs = model.predict([noise, labels])
    imgs = 0.5 * imgs + 0.5 # rescale images 0 - 1
    plot_images(imgs, labels, r, c, title) # visualize images

# create plot of generated images
def plot_images(imgs, labels, r, c, title):
    fig, axs = plt.subplots(r, c)
    fig.set_figheight(12)
    fig.set_figwidth(20)
    fig.suptitle(title, size=20, y=0.95)
    cnt = 0
    for i in range(r):
      for j in range(c):
        axs[i,j].imshow(imgs[cnt,:,:,0], cmap='gray')
        axs[i,j].set_title("Covid: %d" % labels[cnt])
        axs[i,j].axis('off')
        cnt += 1
    plt.show()
    plt.close()

# generate 50 images using pretrained model
generate_images('final_model', 5, 10, 'Example of 50 randomly generated images',)
```

Figure 25: Generating 50 images using a pretrained cGAN model.

```
data = pickle.load(open("preprocessed_data.pkl", "rb"))

# hyperparameters to explore
params = {
    'epochs': [100, 200, 300],
    'batch_size': [16, 32, 64],
    'learning_rate': [0.002, 0.0002, 0.00002],
}

"""### 1.2.1 - Training (Part 1)"""

# build different models with different hyperparameters
for e in params['epochs']:
  for bs in params['batch_size']:
    for lr in params['learning_rate']:
      name = 'e_'+str(e)+'_bs_'+str(bs)+'_lr_'+str(lr) #create a model name
      os.mkdir('images/'+name)
      cgan = CGAN(data, lr)
      cgan.train(e, bs, name)
```

Figure 26: Building different cGAN models and exploring hyperparameters.

```
def plot_training_metrics(history):
  # show training history and accuracy
  plt.plot(history.history['accuracy'])
  plt.plot(history.history['val_accuracy'])
  plt.title('Training␣Accuracy')
  plt.ylabel('accuracy')
  plt.xlabel('epoch')
  plt.legend(['train', 'val'], loc='upper␣left')
  plt.show()

  plt.plot(history.history['loss'])
  plt.plot(history.history['val_loss'])
  plt.title('Training␣Loss')
  plt.ylabel('loss')
  plt.xlabel('epoch')
  plt.legend(['train', 'val'], loc='upper␣left')
  plt.show()

def plot_cf_and_classification(y_test, y_pred):
  # display confusion matrix and classification report
  df_cm = pd.DataFrame(confusion_matrix(y_test, y_pred), index = ['0','1'], columns = ['0','1'])
  df_cm.index.name = 'Actual'
  df_cm.columns.name = 'Predicted'
  plt.figure(figsize = (10,7))
  plt.suptitle("Confusion␣matrix")
  sns.set(font_scale=1.4) #for label size
  sns.heatmap(df_cm, cmap="Blues", annot=True, annot_kws={"size": 16}) # font size

  target_names = ['0', '1']
  print(classification_report(y_test, y_pred))

# load 100 COVID images and normal images (preprocessed)
data = pickle.load(open("preprocessed_data.pkl", "rb"))
X = data['X_train']
y = data['y_train']
X = np.repeat(X, 3, -1) # convert to 3-channel
# split data into 80% train and 20% test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.20, stratify=y, shuffle=True)

# load ResNet50 model with imagenet weight and remove last fully connected layer (include_top = False)
model = ResNet50(weights = 'imagenet', include_top=False, input_tensor=Input(shape=(128, 128, 3)))
model.summary()
# freeze whole pretrained model
for layer in model.layers:
  layer.trainable = False
last_layer = model.layers[-1].output # get last layer from ResNet
# add two new trainable layers
new_layer = Flatten()(last_layer)
new_layer = Dense(1024, activation='relu')(new_layer)
new_layer = Dropout(0.2)(new_layer)
new_layer = Dense(1, activation='sigmoid')(new_layer)
# build and compile model
new_model = Model(model.input, new_layer)
new_model.compile(optimizer=Adam(lr=0.0002), loss='binary_crossentropy', metrics=['accuracy'])

# train new model
epochs = 20
batch_size = 16
callback = EarlyStopping(monitor='loss', patience=3)
history = new_model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size, callbacks=[callback],
    ↪ verbose=1, validation_split=0.1)
plot_training_metrics(history) # show training and validation scores

# evaluate new model
y_pred = new_model.predict(X_test)
y_pred = [1 * (x[0]>=0.5) for x in y_pred]
plot_cf_and_classification(y_test, y_pred)
```

Figure 27: Building a binary classification model using transfer learning on the ResNet-50 model.

```
# load 100 COVID images and normal images (preprocessed)
data = pickle.load(open("preprocessed_data.pkl", "rb"))
X = data['X_train']
y = data['y_train']

# ---------- select 50 random COVID images from X ----------
X_normal_only = [x for i, x in enumerate(X) if y[i] == 0]
X_covid_only = [x for i, x in enumerate(X) if y[i] == 1]
X_half_covid_only = random.sample(X_covid_only, 50) # randomly select 50 COVID images

# ---------- generate 50 random COVID images ---------------
model_name = 'final_model'
model = load_model('models/'+model_name+'.h5')
# generate random noise + labels
noise = np.random.normal(0, 1, (50, 100))
labels = np.ones((50, 1)) # only covid labels
# generate 50 random COVID images
gen_imgs = model.predict([noise, labels])

# combine 50 + 50 COVID images and add to 100 normal images and set appropriate labels
X_covid_only = np.concatenate((X_half_covid_only, gen_imgs))
X = np.concatenate((X_normal_only, X_covid_only))
y = np.concatenate((np.zeros(100), np.ones(100)))
```

Figure 28: Forming a new dataset by selecting 50 random COVID images and combining it with 50 generated Covid images as well as the original 100 normal images.