CS551G - Data Mining and Visualisation. Assessment 1 report.

Task 1: Create Synthetic Covid-19 X-Ray Images with Conditional Generative Adversarial Networks (35/50) – (max ~1500 words)

For Task 1, you are provided with the cgan.py class, which you can adapt to the problem at hand. Therefore, you do not have to implement the cGAN architecture from scratch; but you would have to play with the hyperparameters, such as learning rate, a number of epochs and batch size to optimize the outcome, i.e. generated image quality.

You are asked to do the following:

Subtask 1.1: Using the Covid-19 dataset provided (100 images) and the cgan.py class, generate 50 synthetic images. You need to create a script that preprocesses the data, trains the cGAN model, generates synthetic images and saves them to a folder for further exploration. For your reference examples of synthetic images can be seen in figure 2 (27 Marks).

We were provided with a dataset of 100 Covid-19 images and 100 normal X-Ray images. For this task, we need the Covid-19 images only. Covid-19 images given within the dataset have the min-hight of 237px and min-width of 255px. The given cGAN class model uses the “mnist” dataset to recognise the handwritten digits. I have replaced the ‘Load dataset’ and ‘Configure input’ parts in function ‘train’ with the following code:

```
        data_path = ".../Assessm1/DMV_Assess_1_Covid-19_Dataset/"
        X_train = []

        for folder in os.listdir(data_path):
            if folder == "Covid-19":
                for file in os.listdir(os.path.join(data_path, folder)):
                    try:
                        i_path = os.path.join(data_path, folder, file)
                        l_img = cv2.imread(i_path, cv2.IMREAD_GRAYSCALE)
                        l_img = cv2.resize(l_img, (28,28))
                        l_img = np.array(l_img)
                        l_img = (l_img.astype('float32') - 127.5) / 127.5
                        X_train.append(l_img)

                    except Exception as e:
                        print(e)
                        print("error in " + i_path)
                        continue

         # Configure input
```

```
        X_train = np.expand_dims(X_train, axis=3)
```

This code takes from the dataset Covid-19 images only. The dataset location is stored in the data_path variable. The code iterates each file in the 'Covid-19' folder and with the use of the cv2 library loads the images in grayscale and resize them to the 224x224px size, and adds each image into the training set. Variables "`self.img_rows`" and "`self.img_cols`" are needed to change according to the image size to 224. The number of channels stays the same as we have grayscale images. We need to remove "`self.num_classes`" variable as we are not going to do any classification at this stage. The following lines are added to collect and store loss values for both (discriminator and generator) and discriminator accuracy.

```
self.dis_loss_hist=[]
self.dis_acc_hist=[]
self.gen_loss_hist=[]
```

Optimizer stays the same 'Adam' with the learning rate of 0.0002 and momentum of 0.5. We need to remove the label value as we need only one class (Covid-19 images) to train the model.

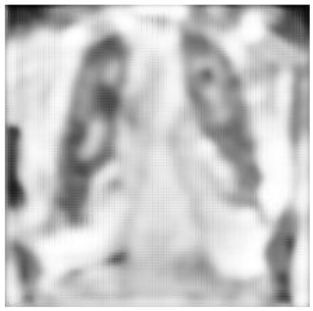
The generator. For the generator, we are building the sequential model. The generator is generating a credible image based on the training domain. The generator works by taking a random point from the latent space as input and outputting the complete generated image. I have tried a number of different models which can be found at:
https://realpython.com/generative-adversarial-networks/
https://livebook.manning.com/book/gans-in-action/chapter-8/56
https://www.kaggle.com/search?q=cgan
https://machinelearningmastery.com/how-to-develop-a-conditional-generative-adversarial-network-from-scratch/
and some more. I have tried these models to find the optimal model on time with the best performance. As the result, I have created a simple model which can be improved at the cost of computational time and resources. I am going to justify below each layer I chose for the generator and the discriminator.
The first layer of the generator is the Dense fully connected layer. This layer is used to interpret the input vector, and create a sufficient number of activations that can be reshaped into an initial low-resolution version of the final output image. I have decided to start with the 256 versions of a 28x28 image. "The Leaky ReLU is used as an activation function as it allows a small, non-zero gradient when the unit is saturated and not active" (Andrew L. Maas, Awni Y. Hannun, Andrew Y. Ng, Rectifier Nonlinearities Improve Neural Network Acoustic Models, 2013). For us, it means it is more likely to keep unchanged the fragment that is not changing from image to image. Leaky ReLU activation is used in most models I have considered to create my model. Reshape does the actual reshape into 256 versions of 28x28 image.
For the next 3 layers, I chose the convolutional transpose layer. The choice is between the UpSampling2D layer and the Conv2DTranspose layer. The UpSampling2D layer is a simple upsampling layer for 2D input. This layer does not perform any learning. So, to use this layer in a

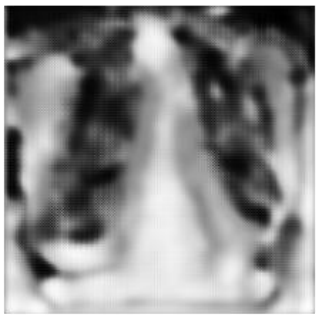
GAN, each of these layers must be followed by the Conv2D layer that allows the model to learn and interpret the doubled input and be trained to transfer it into the meaningful output. The Conv2DTranspose layer acts opposite to the Conv2D layer. This layer is like a combination of upsampling and convolutional layers into one layer. I have tried both and spotted that the model with the transposed layer produces a more accurate, but more squared image. When the model with the upsampling layer produces a more smooth, but blurry image. I have tried to combine these layers but did not spot a big difference, so decided to use the transpose layer. The transpose layer is upsampling the input with 256 filters, uses 4 kernels, strides=(2,2) means the input rows and columns are taken into the 2 strides (output 56x56px), `"same"` results in padding evenly to the left/right or up/down of the input such that output has the same height/width dimension as the input." (Keras, available at https://keras.io/api/layers/convolution_layers/convolution2d/). The next two layers upsample the image to 112x112 and 224x224 respectively with 128 filters each.

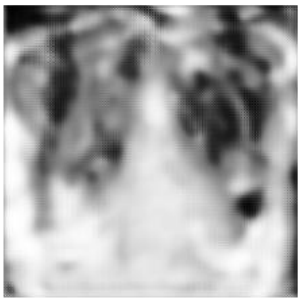
The output convolutional layer that generates the output image has 'tanh' activation function which makes the output coefficients in the range from -1 to 1, and as you remember we scaled the image and made each value of the image matrix in this range.

The discriminator has a convolutional layer as the input layer. This layer takes as an input a tensor of the same size as an image size 224x224x1. The output layer is the dense layer with a sigmoid function activation function to produce the probability that the input-label pair is real rather than fake (no change).

The full code of modified CGAN can be found in the notebook supplied and Appendix 1. The generated images examples below.

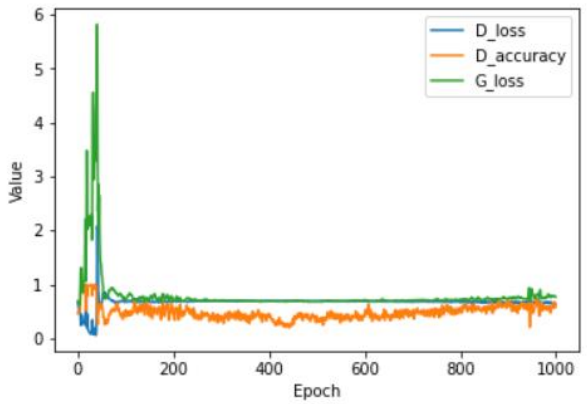






The loss function and discriminator accuracy graph. The accuracy is around 0.5 (50%) most of the time. This means that the discriminator works well in sorting images. The loss graphs have high jumps only at the beginning means both discriminator and generator are well synchronized.

```
cgan.plot_loss()
```

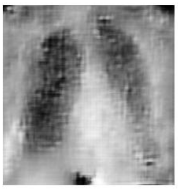


Subtask 1.2: Generate synthetic images based on three different hyperparameters, e.g. different learning rates, epochs, batch size, etc. Clearly show the generated images according to the hyperparameters chosen. For instance, you might show generated images at epoch 100 and also at epoch 200, and so on (**8 Marks**).

Examples from different epochs. I have generated the image for each 40 epochs step, and we can see the improvements with the epoch progress. The image becomes more detailed.

Epoch 60

Epoch 200



Epoch 500

The batch size parameter manipulation is limited for me to 64, as google colab says “Resource Exhausted” when I am trying to make it bigger. I have tried 32 and 64 values for batch size. From the examples below, it is hard to say which result is better, but analysing the whole set of images I have got hypothetically I can say that with the bigger batch size it is possible to achieve a good result with the fewer epochs training.

32

64

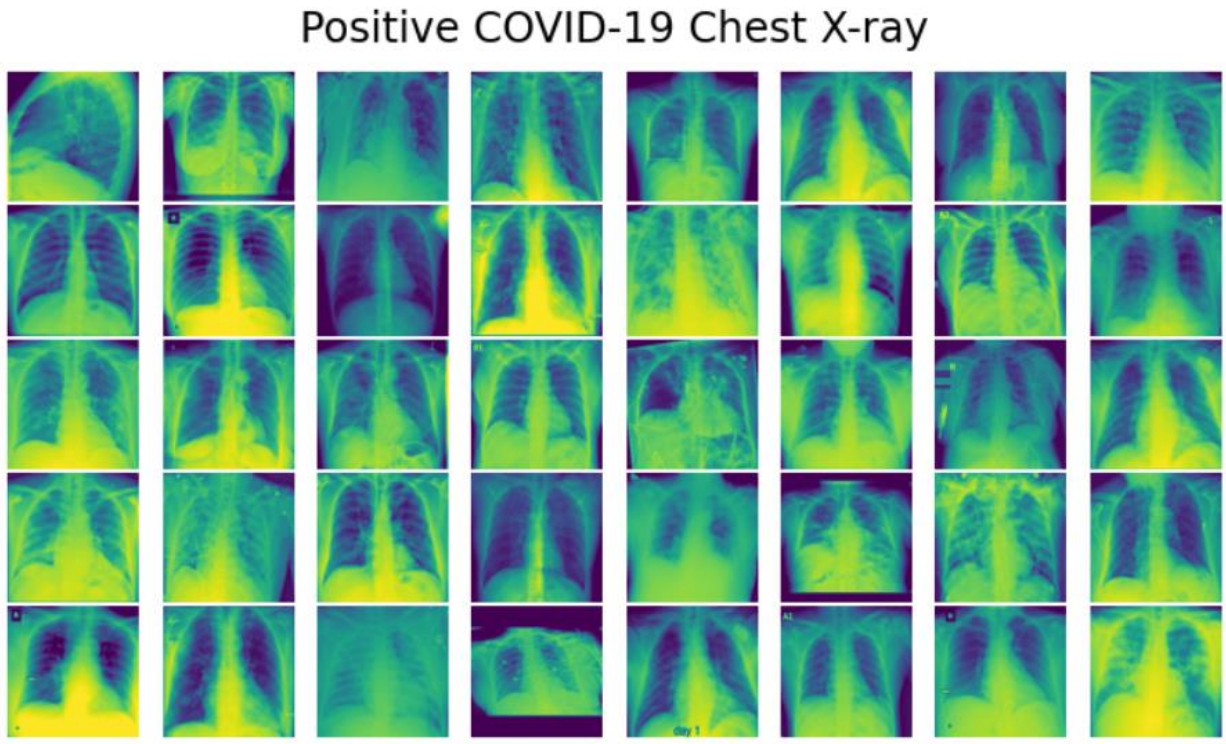
The learning rate is the parameter I found most difficult to adjust. Most papers suggest using 0.0002 value for learning rate as an optimal. I tried the learning rate of 0.02 and 0.000002 and could not see a significant difference. This may be related to the quality of images I have got.

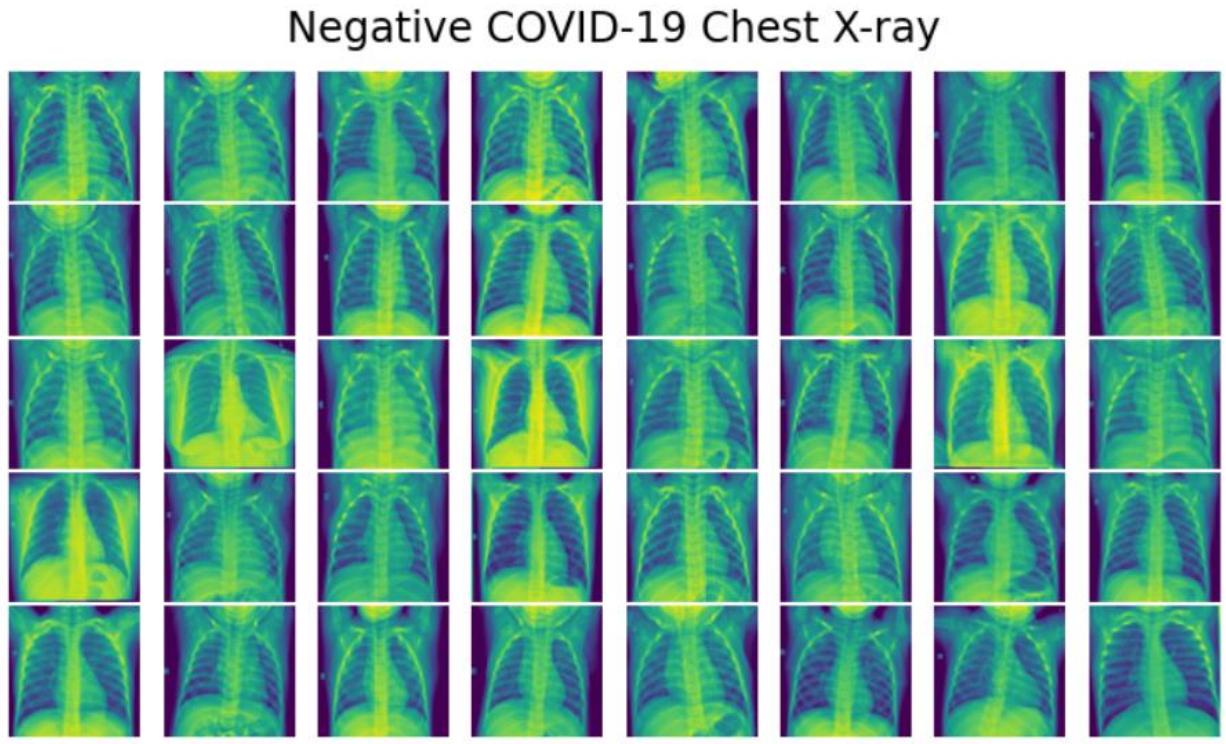
Task 2: Detect Covid-19 from Chest X-Ray Images using pre-trained networks (15/50) – (max ~ 800 words)

Using a pretrained ResNet-50 model that can be found in https://keras.io/api/applications/ and also reporting performance in terms of accuracy, do the following:

Subtask 2.1: Via a transfer learning process, use a pre-trained ResNet-50 model (with imagenet weights) to develop a binary classification model that can classify Covid-19 and Normal CXRs. To accomplish this, use the 100 Covid-19 and 100 Normal CXRs images provided to you. Follow a 80% (train) / 20% (test) process (**hint: freeze the whole pretrained model, remove the last fully connected layer and add two new trainable layers.) (10 Marks).**

For this task, I loaded data the way I described in task 1 using the cv2 library. The data sets with no-covid and covid-19 images have different palette patterns. The images loaded in the greyscale to normalise the palette patterns and with the size 224x224 which we need for the ResNet50 input.





The ResNet50 takes as an input (224,224,3) data shape and we have (224,224) as they are greyscaled. So, to make images appear as an RGB we need to repeat the grayscaled image on each RGB dimension. This can be done with numpy repeat.

```
covid_images = np.repeat(covid_images[..., np.newaxis], 3, -1)
```

```
normal_images = np.repeat(normal_images[..., np.newaxis], 3, -1)
print(covid_images.shape)
```

Out: `(100, 224, 224, 3)`

On output, we have 100 covid images and 100 normal images in format (224,224,3). Next, we are splitting the training set with 80% of data and the testing set 20% of data and join the covid and normal image sets. The labels have to be in the categorical format, so we are transforming the format of the labels into the categorical.

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelBinarizer
from tensorflow.keras.utils import to_categorical

# split into training and testing
covid_x_train, covid_x_test, covid_y_train, covid_y_test = train_test_split(
    covid_images, covid_labels, test_size=0.2)
noncovid_x_train, noncovid_x_test, noncovid_y_train, noncovid_y_test =
train_test_split(
    normal_images, normal_labels, test_size=0.2)


X_train = np.concatenate((noncovid_x_train, covid_x_train), axis=0)
X_test = np.concatenate((noncovid_x_test, covid_x_test), axis=0)
y_train = np.concatenate((noncovid_y_train, covid_y_train), axis=0)
y_test = np.concatenate((noncovid_y_test, covid_y_test), axis=0)

# make labels into categories - either 0 or 1
y_train = LabelBinarizer().fit_transform(y_train)
y_train = to_categorical(y_train)

y_test = LabelBinarizer().fit_transform(y_test)
y_test = to_categorical(y_test)
```

We need to remove top layers of ResNet and add layers to the end to generate the output.

```
from tensorflow.keras.preprocessing.image import ImageDataGenerator

res = ResNet50(weights="imagenet", include_top=False,
    input_tensor=Input(shape=(224, 224, 3)))

outputs = res.output
outputs = Flatten(name="flatten")(outputs)
outputs = Dropout(0.5)(outputs)
outputs = Dense(2, activation="softmax")(outputs)

model = Model(inputs=res.input, outputs=outputs)
```

```
for layer in res.layers:
    layer.trainable = False

model.compile(
        loss='categorical_crossentropy',
        optimizer='adam',
        metrics=['accuracy']
)

train_aug = ImageDataGenerator(
    rotation_range=20,
    width_shift_range=0.2,
    height_shift_range=0.2,
    horizontal_flip=True
)
```

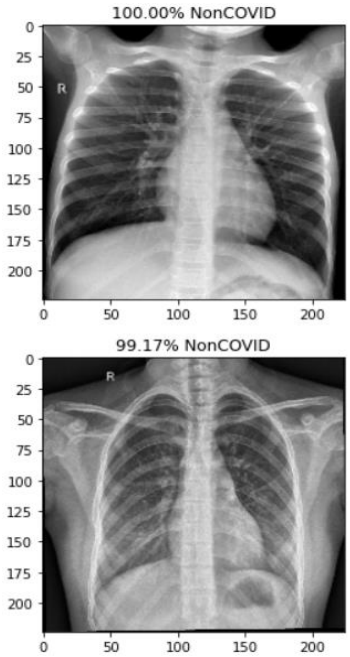
Now we are ready to train our model.

```
history = model.fit(train_aug.flow(X_train, y_train, batch_size=32),
                    validation_data=(X_test, y_test),
                    validation_steps=len(X_test) / 32,
                    steps_per_epoch=len(X_train) / 32,
                    epochs=500)
```

After the training finished we can test our model.

```
y_pred = model.predict(X_test, batch_size=batch_size)
```

Prediction visualization is shown below.



The Classification Report.

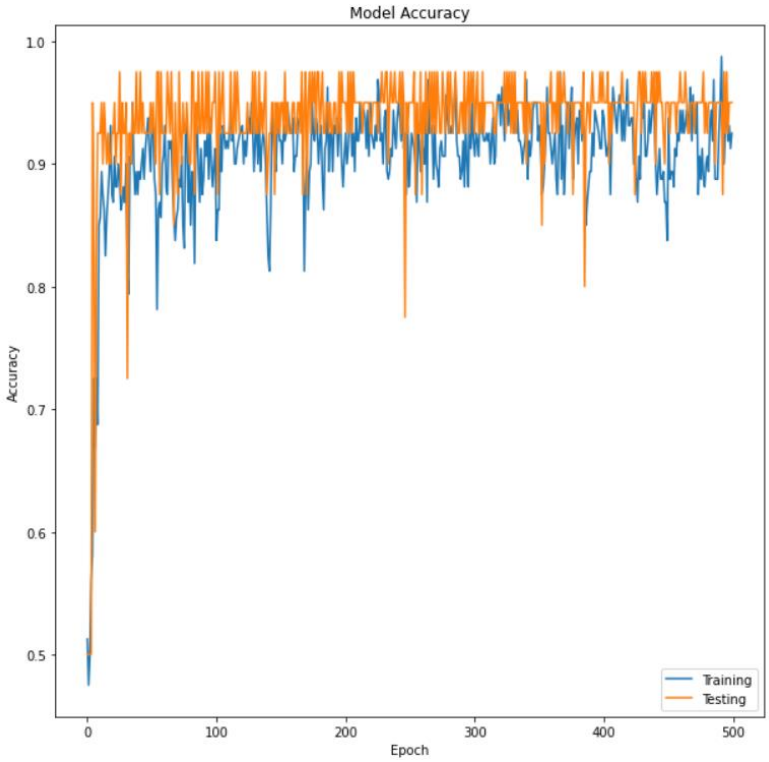
```
from sklearn.metrics import classification_report
print(classification_report(y_test_bin, y_pred_bin))
```

```
              precision    recall  f1-score   support

           0       0.95      0.95      0.95        20
           1       0.95      0.95      0.95        20

    accuracy                           0.95        40
   macro avg       0.95      0.95      0.95        40
weighted avg       0.95      0.95      0.95        40
```

Accuracy and Loss.



Subtask 2.2: Repeat subtask 2.1, but now randomly select 50 of the real Covid-19 images and add another 50 images generated as part of Task 1. The normal images remain the same 100 ones used in subtask 2.1 **(5 Marks)**.

We need 50 images from Covid-19 folder to load 50 images I added a counter to stop at 50. The generated images loaded with the code below.

```
elif folder == "images50":
```

```
    for file in os.listdir(os.path.join(data_path, folder)):
      i_path = os.path.join(data_path, folder,  file)
      try:
        l_img = cv2.imread(i_path,cv2.IMREAD_GRAYSCALE)
        l_img = cv2.resize(l_img, (224,224))
        l_img = np.array(l_img)
        l_img = (l_img.astype('float32')) / 255
        #img /= 255
        covid_images.append(l_img)
        covid_labels.append([1])

      except Exception as e:
        print(e)
        print("error in " + i_path)
        continue
```

The result is two sets, one with covid images (50 generated and 50 real), and one with normal (no covid) images.

```
print(normal_images.shape)
print(covid_images.shape)
```

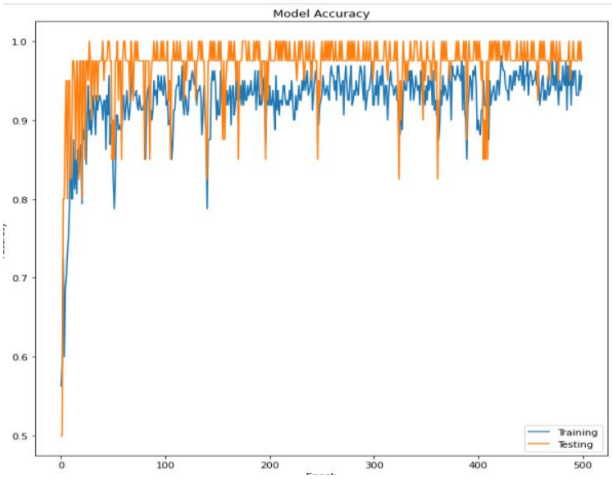
```
(100, 224, 224)
(100, 224, 224)
```

All steps the same as in task 2.1. The classification report.

```
[19] from sklearn.metrics import classification_report
     print(classification_report(y_test_bin, y_pred_bin))
```

	precision	recall	f1-score	support
0	1.00	0.95	0.97	20
1	0.95	1.00	0.98	20
accuracy			0.97	40
macro avg	0.98	0.97	0.97	40
weighted avg	0.98	0.97	0.97	40

We can see that the bad quality of generated images even improved the accuracy.



We have a binary classification model, but the images generated in task 1 are different from the real ones and they could be recognized as the third class. We trained the model with these images included into the training set as the covid images and this made the system to recognise such images as abnormal (covid) images.

Appendix 1.

```
class CGAN():
    def __init__(self):
        # Input shape
        #rows and cols are changed from 28 to 224
        self.img_rows = 224
        self.img_cols = 224
        self.channels = 1
        self.img_shape = (self.img_rows, self.img_cols, self.channels)
        #only 2 classes positive and negative
        #self.num_classes = 2
        self.latent_dim = 100

        self.dis_loss_hist=[]
        self.dis_acc_hist=[]
        self.gen_loss_hist=[]

        optimizer = Adam(0.0002, 0.5)

        # Build and compile the discriminator
        self.discriminator = self.build_discriminator()
        self.discriminator.compile(loss=['binary_crossentropy'],
            optimizer=optimizer,
            metrics=['accuracy'])

        # Build the generator
        self.generator = self.build_generator()

        # The generator takes noise and the target label as input
        # and generates the corresponding digit of that label
        noise = Input(shape=(self.latent_dim,))
        #label = Input(shape=(1,))
        img = self.generator(noise)

        # For the combined model we will only train the generator
        self.discriminator.trainable = False

        # The discriminator takes generated image as input and determines
```

```
validity
        # and the label of that image
        valid = self.discriminator(img)

        # The combined model  (stacked generator and discriminator)
        # Trains generator to fool discriminator
        self.combined = Model(noise, valid)
        self.combined.compile(loss=['binary_crossentropy'],
            optimizer=optimizer)

    def build_generator(self):

        model = Sequential()
        # foundation for 56x56 image
        n_nodes = 256 * 28 * 28
        model.add(Dense(n_nodes, input_dim=self.latent_dim))
        model.add(LeakyReLU(alpha=0.2))
        model.add(Reshape((28, 28, 256)))
        # upsample to 112x112
        model.add(Conv2DTranspose(256, (4,4), strides=(2,2), padding='same'))
        model.add(LeakyReLU(alpha=0.2))
        # upsample to 112x112
        model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
        model.add(LeakyReLU(alpha=0.2))
        # upsamplde to 224x224
        model.add(Conv2DTranspose(128, (4,4), strides=(2,2), padding='same'))
        model.add(LeakyReLU(alpha=0.2))
        model.add(Conv2D(1, (7,7), activation='tanh', padding='same'))

        model.summary()

        noise = Input(shape=(self.latent_dim,))
#         label = Input(shape=(1, ), dtype='int32')
#         label_embedding = Flatten()(Embedding(self.num_classes,
self.latent_dim)(label))

#         model_input = multiply([noise, label_embedding])
        img = model(noise)

        return Model(noise, img)


    def build_discriminator(self):

        model = Sequential()
```

```
        model.add(Conv2D(64, (3,3), padding='same',
input_shape=self.img_shape))
        model.add(LeakyReLU(alpha=0.2))
        model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same'))
        model.add(LeakyReLU(alpha=0.2))
        model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same'))
        model.add(LeakyReLU(alpha=0.2))
        model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same'))
        model.add(LeakyReLU(alpha=0.2))
        model.add(Flatten())
        model.add(Dropout(0.4))
        model.add(Dense(1, activation='sigmoid'))

        model.summary()

        img = Input(shape=self.img_shape)


        validity = model(img)#model(model_input)

        return Model(img, validity)


    def train(self, epochs, batch_size, sample_interval):

        # Load the dataset
#         (X_train, y_train), (_, _) = mnist.load_data()
        data_path = "/content/drive/My Drive/Colab
Notebooks/DMV_Assess_1_Covid-19_Dataset/"
        X_train = []
        #y_train = []

        for folder in os.listdir(data_path):
            if folder == "Covid-19" or folder == "Covid-19":
                for file in os.listdir(os.path.join(data_path, folder)):
                    try:
                        i_path = os.path.join(data_path, folder,  file)
                        l_img = cv2.imread(i_path, cv2.IMREAD_GRAYSCALE)
                        l_img = cv2.resize(l_img, (224,224))
                        l_img = np.array(l_img)
                        l_img = (l_img.astype('float32') - 127.5) / 127.5
                        #img /= 255
                        X_train.append(l_img)
```

```
                    except Exception as e:
                        print(e)
                        print("error in " + i_path)
                        continue

#         # Configure input
#         X_train = (X_train.astype(np.float32) - 127.5) / 127.5
        X_train = np.expand_dims(X_train, axis=3)

        # Adversarial ground truths
        valid = np.ones((batch_size, 1))
        fake = np.zeros((batch_size, 1))

        for epoch in range(epochs):

            # ---------------------
            #  Train Discriminator
            # ---------------------

            # Select a random half batch of images
            idx = np.random.randint(0, X_train.shape[0], batch_size)
            #imgs, labels = X_train[idx], y_train[idx]
            imgs = X_train[idx]

            # Sample noise as generator input
            noise = np.random.normal(0, 1, (batch_size, self.latent_dim))

            # Generate a half batch of new images
            gen_imgs = self.generator.predict(noise)

            # Train the discriminator
            d_loss_real = self.discriminator.train_on_batch(imgs, valid)
            d_loss_fake = self.discriminator.train_on_batch(gen_imgs, fake)
            d_loss = 0.5 * np.add(d_loss_real, d_loss_fake)

            #store discriminator history
            self.dis_loss_hist.append(d_loss[0])
            self.dis_acc_hist.append(d_loss[1])

            # ---------------------
            #  Train Generator
            # ---------------------

            # Condition on labels
            #sampled_labels = np.random.randint(0, 2, batch_size).reshape(-1,
```

```
1)

            # Train the generator
            g_loss = self.combined.train_on_batch(noise, valid)

            self.gen_loss_hist.append(g_loss)

            # Plot the progress
            print ("%d [D loss: %f, acc.: %.2f%%] [G loss: %f]" % (epoch,
d_loss[0], 100*d_loss[1], g_loss))

            # If at save interval => save generated image samples
            if epoch % sample_interval == 0:
                self.sample_images(epoch)

    def sample_images(self, epoch):

        noise = np.random.normal(0, 1, (10, self.latent_dim))
        sampled_labels = np.arange(0, 2).reshape(-1, 1)
        gen_imgs = self.generator.predict(noise)

        # Rescale images 0 - 1
        gen_imgs = 0.5 * gen_imgs + 0.5

        #Save generated image
        # turn off axis
        plt.axis('off')
        plt.imshow(gen_imgs[0,:,:,0], cmap='gray')
        plt.savefig("/content/drive/My Drive/Colab
Notebooks/DMV_Assess_1_Covid-19_Dataset/images/%d.png" % epoch)
        plt.close()

    def plot_loss(self):
      plt.plot(self.dis_loss_hist)
      plt.plot(self.dis_acc_hist)
      plt.plot(self.gen_loss_hist)
      plt.legend(["D_loss", "D_accuracy", "G_loss"])

      plt.xlabel("Epoch")
      plt.ylabel("Value")

      plt.show()
```