# Application of GAN in Camera Surveillance

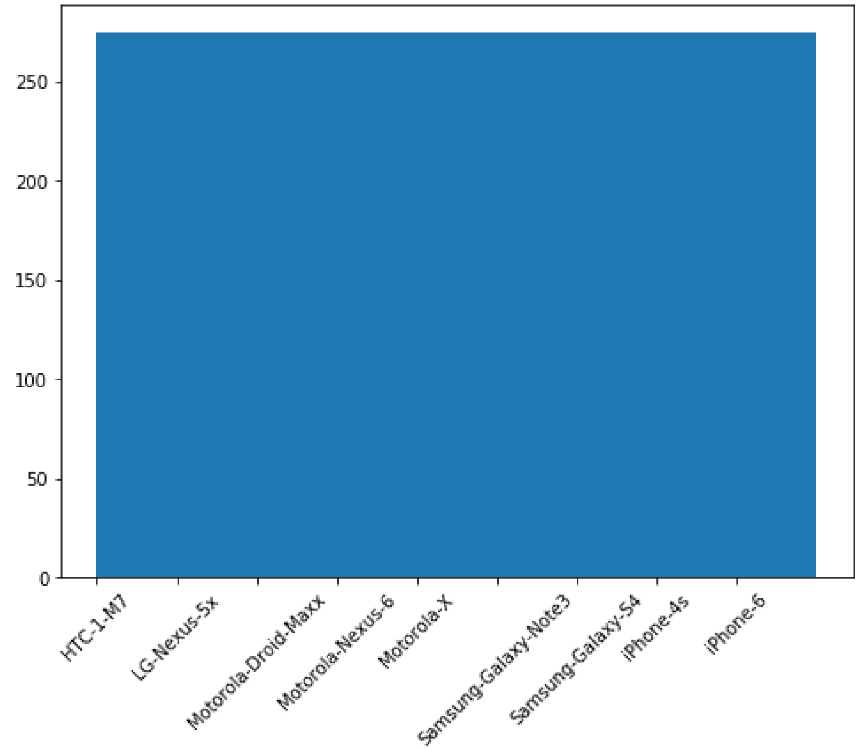
## Introduction:

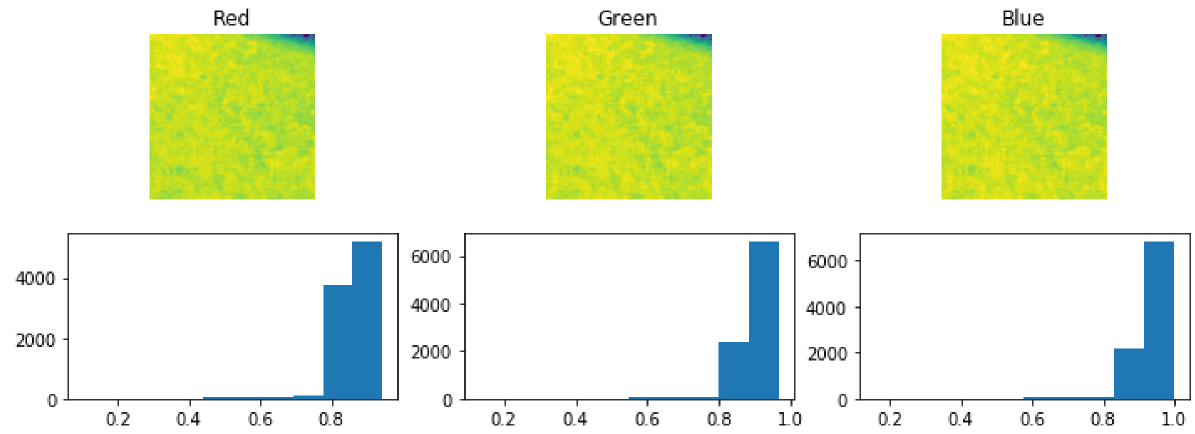
GANs are even deeper and were brought in to the real world by Ian Goodfellow and his team in 2014. GANs consist of two neural networks – the generator and the discriminator – the two of which try to improve in order to outdo each other. The generator essentially generates the high fakes while the discriminator is simply designed to assess the high fakes against actual high data. This adversarial process goes on until the generator produces data that cannot at all be distinguished from real data.

## GANs in Camera Surveillance:

For camera surveillance, GANs have attracted a lot of attention because they can augment and synthesize high quality data. Video data is usually in tons, thus surveillance systems utilize it for monitoring, detecting anomalies and facial recognition. But real world data is real and not as crystalline as the model data and one may realize that the data set used is a subset of the complete data in the real world context. GANs is the solution to these problems and generate synthetic data, enhance the quality of the images and provide the avenue for interactive data augmentation.

## Input Dataset



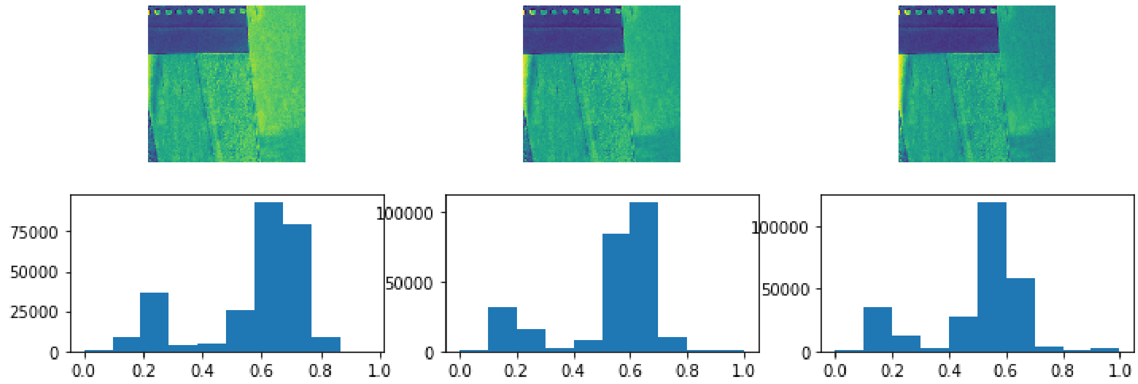



## GAN Model Architecture

```
from keras.callbacks
 import ModelCheckpoint, LearningRateScheduler, EarlyStopping, ReduceLROnPlateau, TensorBoard
from keras import optimizers, losses, activations, models
from keras.layers import Convolution2D, Dense, Input, Flatten, Dropout, MaxPooling2D, BatchNormalization, GlobalAveragePooling2D, GlobalMaxPool2D, concatenate
def gap_drop(in_layer):
    gap_layer = GlobalAveragePooling2D()(Convolution2D(16, kernel_size = 1)(in_layer))
    gmp_layer = GlobalMaxPool2D()(Convolution2D(16, kernel_size = 1)(in_layer))
return Dropout(rate = 0.5)(concatenate([gap_layer, gmp_layer]))
def create_model():
    inp = Input(shape=(None, None, 3))
    norm_inp = BatchNormalization()(inp)
    gap_layers = []
    img_1 = Convolution2D(16, kernel_size=3, activation=activations.relu, padding="same")(norm_inp)
    img_1 = Convolution2D(16, kernel_size=3, activation=activations.relu, padding="same")(img_1)
img_1 = MaxPooling2D(pool_size=(2, 2))(img_1)
```

```
    img_1 = Dropout(rate=0.2)(img_1)
    img_1 = Convolution2D(32, kernel_size=3, activation=activations.relu, padding="same")(img_1)
    img_1 = Convolution2D(32, kernel_size=3, activation=activations.relu, padding="same")(img_1)
    gap_layers += [gap_drop(img_1)]
    img_1 = MaxPooling2D(pool_size=(2, 2))(img_1)
    img_1 = Dropout(rate=0.2)(img_1)
    img_1 = Convolution2D(64, kernel_size=2, activation=activations.relu, padding="same")(img_1)
    img_1 = Convolution2D(64, kernel_size=2, activation=activations.relu, padding="same")(img_1)
    gap_layers += [gap_drop(img_1)]
    gap_cat = concatenate(gap_layers)
    dense_1 = Dense(32, activation=activations.relu)(gap_cat)
    dense_1 = Dense(nclass, activation='softmax')(dense_1)
model = models.Model(inputs=inp, outputs=dense_1)
    opt = optimizers.Adam(lr=1e-3) # karpathy's magic learning rate
    model.compile(optimizer=opt,
                  loss='categorical_crossentropy',
                  metrics=['acc'])
    model.summary()
    return model
```

## Output




## Summary:

The paper elaborates on how generative adversarial networks—initially introduced by Ian Goodfellow way back in 2014—are going to dramatically change the face of camera surveillance. Basically, GANs involve a creative duel between two neural networks: one is a generator that crafts fake data, and another is a discriminator that identifies the fakes. This continues until the generated data is nearly indistinguishable from the real thing.

They really game the situation in terms of surveillance. GANs do not just synthesize data but also improve the quality of images, making the video data useful for tasks like anomaly detection, facial recognition, etc. Real-world data is messy and incomplete, but GANs resolve this problem by augmenting and refining it.

It also proposes a practical architecture of GAN with Keras, together with convolution, pooling, and dropout layers, all tuned to optimize performance. The ability of these technologies to be harnessed in full by GANs would serve to increase the effectuality of camera surveillance systems in terms of improved monitoring and analysis.

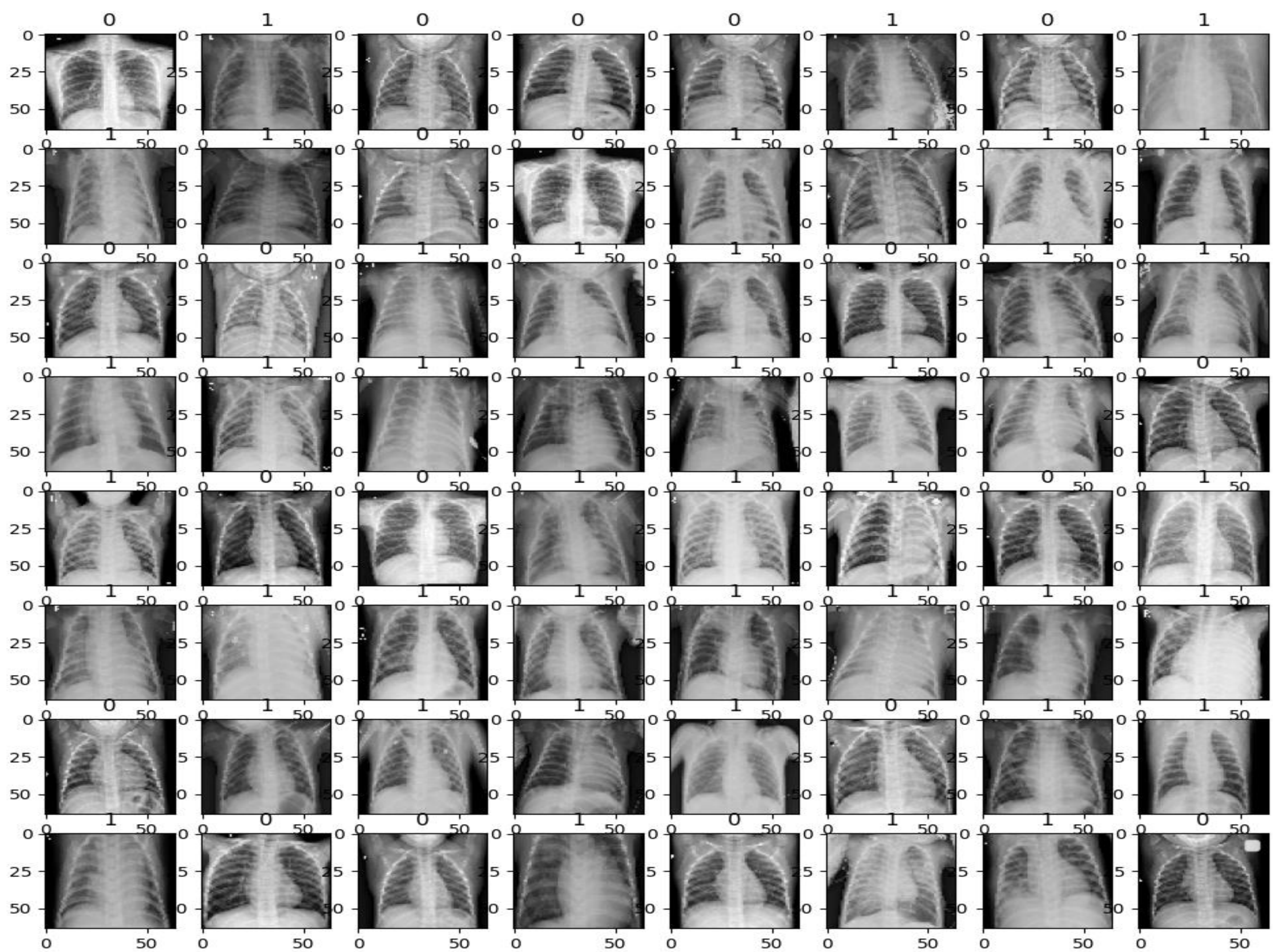
# Applications of GAN in Medical field

## Using GAN to Generate Chest X-Ray Images

### Overview

- The following study presents a model for generating chest X-ray images of normal subjects (without lung disease) and pneumonia patients.
- Through the proposed model, I tried to avoid most of the problems that the GAN models suffer from, in terms of the difficulty of training each of the generator and the discriminant, in addition to the problem of the modal collapse and the perceptual quality, so that I tried through the proposed model, to try to continue the training (to ensure the continuity of the derivability of cost function ) and discovering the features by the discriminator (the most accurate features for each case of the dataset), which leads the generator to focus on them during the training process.
- A conditional model was used for the GAN, and the discriminator was forced to determine whether the medical images are real or not, in addition to identifying the pathological condition in the generated images.
- I used (64, 64, 3) images because I didn't have enough computational resources.
- I used Google Colab For Training.
- Reading the images included in the dataset, which is for the sound health condition, and the other case, which is pneumonia.
- I have included all medical images included in each class, although the number of samples per class varies (thus this would require training for a higher number of Epochs for the GAN).

### Input Dataset



## GAN Model Architecture

```
class Acgan:
    def __init__(self, eta, batch_size, epochs, weight_decay, latent_space,
                 image_shape, kernel_size):
        self.eta = eta
        self.batch_size = batch_size
        self.epochs = epochs
        self.weight_decay = weight_decay
        self.latent_space = latent_space
        self.image_shape = image_shape
        self.kernel_size = kernel_size
    def data(self, images, labels):
        ytrain = tf.keras.utils.to_categorical(labels)
        self.images = images
        self.labels = ytrain
    def samples(self, G, noize, labels):
        images = G.predict([noize, labels])
        ys = np.argmax(labels, axis = 1)
        plt.figure(figsize = (12, 4))
        for i in range(16):
            plt.subplot(2, 8, (i + 1))
            plt.imshow(images[i], cmap = 'gray')
            plt.title(ys[i])
        plt.show()
    def generator(self, inputs, labels):
        filters = [256, 128, 64, 32]
        padding = 'same'
        x = inputs
        y = labels
        x = layers.concatenate([x, y])
        x = layers.Dense(1024, )(x)
        x = layers.Dense(8*8*filters[0],
                         kernel_regularizer = tf.keras.regularizers.L2(0.001))(x)
        x = layers.Reshape((8, 8, filters[0]))(x)
        for filter in filters:
            if filter >= 64:
                strides = 2
            else:
                strides = 1
            x = LayerNormalization()(x)
            x = layers.Activation('relu')(x)
            x = Conv2DTranspose(filter, kernel_size = self.kernel_size, padding = padding,
                   strides = strides)(x)
        x = Conv2DTranspose(3, kernel_size = self.kernel_size, padding = padding)(x)
        x = layers.Activation('sigmoid')(x)
        self.generatorModel = models.Model(inputs = [inputs, labels],
                                   outputs = x,
                                   name = 'generator')
    def discriminator(self, inputs):
        x = inputs
        filters = [32, 64, 128, 256]
        padding = 'same'
        for filter in filters:
            if filter < 256:
                strides = 2
            else:
                strides = 1
            x = Conv2D(filter, kernel_size = self.kernel_size, padding = padding,
                   strides = strides,
                   kernel_regularizer = tf.keras.regularizers.L2(0.001))(x)
```

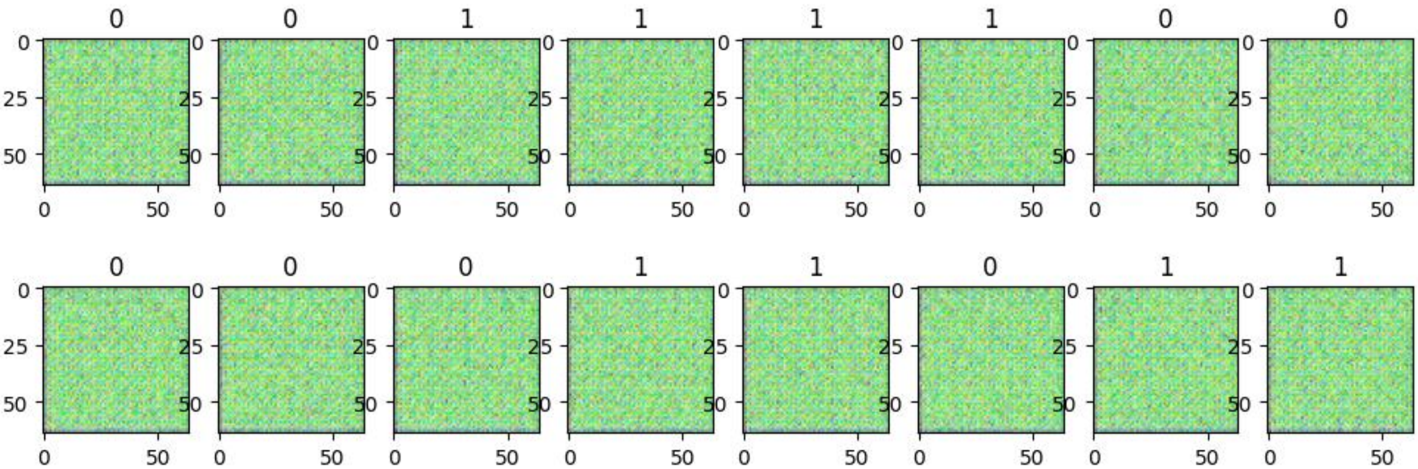
```
        x = LeakyReLU(alpha = 0.2)(x)
    x = layers.Flatten()(x)
    outputs = Dense(1, )(x)
    labelsOutput = Dense(256,
                         kernel_regularizer = tf.keras.regularizers.L2(0.001))(x)
    labelsOutput = Dropout(0.3)(labelsOutput)
    labelsOutput = Dense(2,)(labelsOutput)
    labelsOutput = layers.Activation('softmax')(labelsOutput)
    self.discriminatorModel = models.Model(inputs = inputs,
                                           outputs = [outputs, labelsOutput],
                                           name = 'discriminator')
def build(self,):
    generatorInput = layers.Input(shape = (self.latent_space))
    discriminatorInput = layers.Input(shape = (self.image_shape))
    labelsInput = layers.Input(shape = (2, ))
    self.generator(generatorInput, labelsInput)
    self.discriminator(discriminatorInput)
    G = self.generatorModel
    D = self.discriminatorModel
    D.compile(loss = ['mse', 'binary_crossentropy'],
              optimizer = tf.keras.optimizers.RMSprop(learning_rate = self.eta,
                                                      weight_decay = self.weight_decay))
    D.summary()
    G.summary()
    D.trainable = False
    GAN = models.Model(inputs = [generatorInput, labelsInput],
                       outputs = D(G([generatorInput, labelsInput])))
    GAN.compile(loss = ['mse', 'binary_crossentropy'],
                optimizer = tf.keras.optimizers.RMSprop(learning_rate = self.eta*0.5,
                                                        weight_decay = self.weight_decay*0.5))
    GAN.summary()
    return G, D, GAN
def trainAlgorithm(self, G, D, GAN):
    for epoch in range(self.epochs):
        indexs = np.random.randint(0, len(self.images), size = (self.batch_size, ))
        realImages = self.images[indexs]
        realLabels = self.labels[indexs]
        realTag = tf.ones(shape = (self.batch_size, ))
        noize = tf.random.uniform(shape = (self.batch_size,
                                           self.latent_space), minval = -1,
                                  maxval = 1)
        fakeLabels = tf.keras.utils.to_categorical(np.random.choice(range(2), size = (self.batch_size)),
                                                   num_classes = 2)
        fakeImages = tf.squeeze(G.predict([noize, fakeLabels], verbose = 0))
        fakeTag = tf.zeros(shape = (self.batch_size, ))
        allImages = np.vstack([realImages, fakeImages])
        allLabels = np.vstack([realLabels, fakeLabels])
        allTags = np.hstack([realTag, fakeTag])
        _, dlossTag, dlossLabels = D.train_on_batch(allImages, [allTags, allLabels])
        noize = tf.random.uniform(shape = (self.batch_size,
                                           self.latent_space), minval = -1,
                                  maxval = 1)
        _, glossTag, glossLabels = GAN.train_on_batch([noize, fakeLabels], [realTag, fakeLabels])
        if epoch % 5000 == 0:
            print('Epoch: {}'.format(epoch))
            print('discriminator loss: [tag: {}, labels: {}], generator loss: [tag: {}, labels: {}]'.format(dlossTag,
                                                                                                          dlossLabels,
                                                                                                          glossTag,
                                                                                                          glossLabels))
            self.samples(G, noize, fakeLabels)
```

# Output

Epoch: 0

discriminator loss: [tag: 0.5253190994262695, labels: 0.6905013918876648], generator loss: [tag: 0.26063966751098633, labels: 0.7032241821289062]

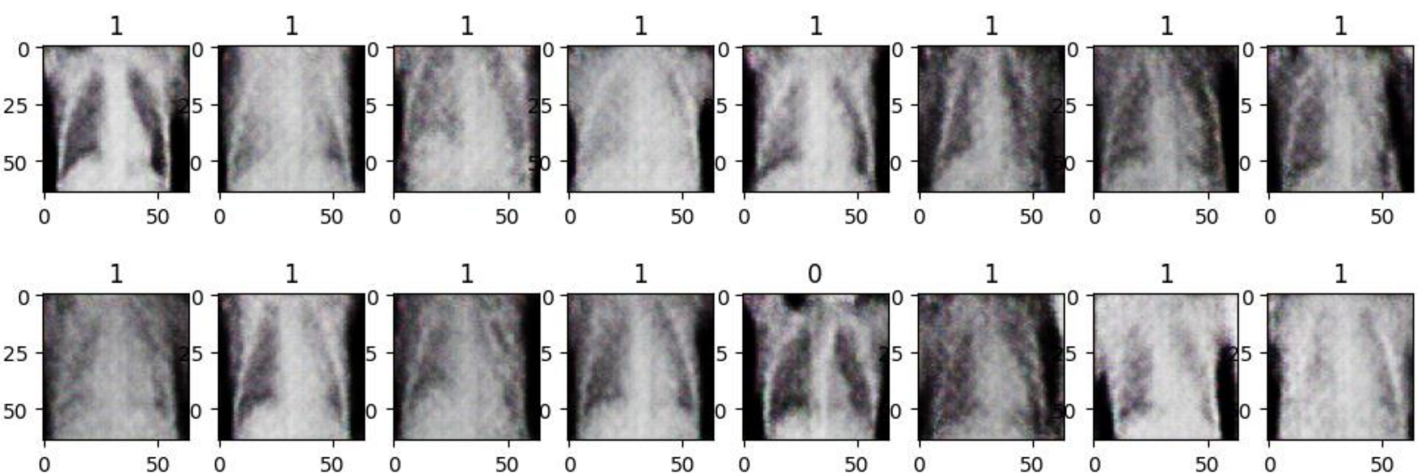
1/1 [==============================] - 0s 20ms/step




Epoch: 5000

discriminator loss: [tag: 0.2465503215789795, labels: 0.056896377354860306], generator loss: [tag: 0.248743936419487, labels: 0.009133875370025635]

1/1 [==============================] - 0s 19ms/step




Epoch: 30000

discriminator loss: [tag: 0.24253657460212708, labels: 0.003363359486684203], generator loss: [tag: 0.23190180957317352, labels: 0.00012597988825291395]

1/1 [==============================] - 0s 18ms/step