COVIDGAN - HOMEWORK

FELIX RYDELL, MENA NADUM

Abstract. Using a limited supply of data (932 training samples), in this case X-ray pictures of the lungs of COVID-19 afflicted patients, we generate new data using a standard conditional generative adverserial network cGAN (1068 new training samples). We build cGAN models using two different libraries PyTorch and Tensorflow with semi-similar architectures. Next we transfer learn with the help of VGG16 to detect COVID-19 cases. We find that including the generated data in the training gave an increase from 0.958 to 0.974 in validation accuracy with PyTorch and to 0.98 with a Tensorflow model.

1. Introduction

The problem we are investigating is how generative adverserial networks can be used in the field of medicine. More specifically, in the beginning of a new pandemic, there will always be a lack of data. It turns out that detection and severity of COVID-19 can be diagnosed through X-ray images of the patients lung with good accuracy. Even though there may be methods such as blood or saliva tests that are examined in labs to determine the status of a disease in patient, X-ray lung examinations are great visual indicators of the spread and severity of the disease, which could provide a quicker diagnos. Hence there is interest in classifying the COVID-19 infected chest X-rays. Training an neural network to detect such patterns in images requires a lot of data. If one only has access to a limit supply of data, one could remedy this by generating new data using a generative adversarial network.

We chose 331 COVID-19 X-ray images of lungs and 601 normal lung images for our experiment. We then train a conditional generative adverserial network to generate a bigger set of data, so that we have 1000 COVID-19 images and 1000 normal ones. We compare the results from transfer learning of the last layer of a VGG16 network with both the small and bigger dataset that we have generated. For both trainings we use the same validation set of 72 COVID-19 images and 120 normal images.

The fact that we choose specifically 331/601 samples to start with is because we wanted to recreate the experiment of [4], which we discuss below.

Date: August 11, 2021.

2. Related Work

In [4], which was published not long after the outbreak of the pandemic (April 30, 2020), the authors find that the available dataset of COVID-19 and normal lung X-rays that were available to them (331 COVID-19 images and 601 normal images), were only enough to train a VGG16 network to about 85 % accuracy. Note that we did not use the same dataset for our replication. They use a generative adverserial network to generate 1669 COVID-19 images and 1399 normal ones. In total the new training dataset consisted of 2000/2000 (covid/normal.) With this, VGG16 gave them about 95 % accuracy. The general structure they used for their GAN is a typical one, illustrated by Figure 1.

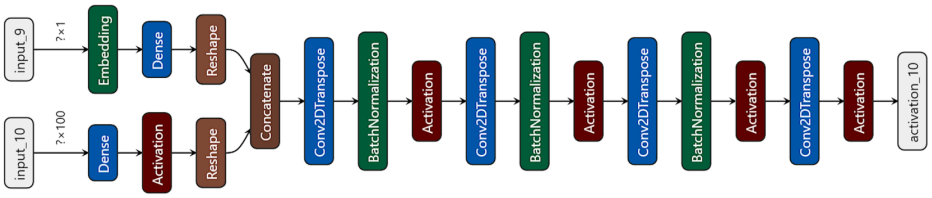

FIGURE 1. The generator structure of [4]. input_9 corresponds to the class label and input_10 corresponds to the random noise vector.

fig:one

In our implementation, we don't follow all details from this article precisely, instead we follow [2], where Sharma details and explains how to set up a conditional GAN in detail in both tensorflow and pytorch.

3. Methods

The work is conducted using python Tensorflow and Pytorch libaries on two different experiments with semi-similar network architectures. We divide this section into three parts. We first describe details of CovidGAN, which is the network used in [4]. Secondly, we describe our experiment using tensorflow. Thirdly, we describe our implementation in PyTorch.

3.1. CovidGAN.

A generative adversal network (GAN) is a neural network for training generative models. It consists of a generator and a discriminator model. The generator as per its name tries to generate an image using a noise input similar to the images in the training data, it tries to fool the discriminator into thinking that the image is real. While the discriminator is fed both the real and the generated image where it guesses which one is real. Based on the feedback from the discriminator through backpropagation and loss function the generator tries to improve itself by generating more realistic images. The optimization problem can be formulated by the combined loss as in equation below [1].

$$L = \min_G \max_D [\log(D(x)) + \log(1 - D(G(z)))].$$

The discriminator tries to maximize the final loss while the generator is competing against it by minimizing the loss.

For CovidGAN we basically use what is called a conditional GAN where both the discriminator and generator are conditioned on some extra information. In our case they are conditioned on a class label where we have either COVID-19 infected or normal chest x-rays. Based on these the model will generate an image of the desired label. This way we can generate more data after training the model on a small dataset.

3.2. **Architectures.** We implement the GAN in both Tensorflow and PyTorch similarly. A detailed architecure of the GAN in PyTorch is shown in Figure 3. The architecture for both submodels of the Tensorflow implementation are shown if Figure 2.

We now describe the general setup in words. The discriminator takes in 3x128x128 images (1x128x128 in Tensorflow.) It inputs an image and a label that corresponds to COVID-19 or normal. It outputs one value between 0 and 1, which is to be interpreted as the probability that the image is a real image of an X-ray corresponding to the label. Our generator inputs a random noise vector of length 100 and a label that corresponds to COVID-19 or normal. It outputs a 3x128x128 (or 1x128x128) image that corresponds to the label, which depends on the random noise.

3.3. **Training.** For the tensorflow network the model is trained on a smaller batch size 32 (64 in Pytorch). Both models use batch normalization after each layer except for the input and output layers (0.99 for the discriminator and 0.5 for the generator). Binary cross entropy loss is applied in both models and Adam optimizers are employed with learning rate 0.0002 and beta value 0.5.

4. Data, Experiments and Findings

The data we use has been obtained from https://www.kaggle.com/preetviradiya/covid19-radiography-dataset [3]. This consists of X-ray images of lungs of patients, including 3615 images of COVID-19 patients, 10192 images of patients not infected by COVID-19 (although as explained previously, we chose 331 COVID-19 images and 601 normal images for our training, and 72/120 for validation.) This data has been collected by a team of researchers from Qatar University, Doha, Qatar, and the University of Dhaka, Bangladesh. This was done together with their collaborators from Pakistan and Malaysia with medical doctors. We resized the images to 3x128x128, but did no other preprocessing. The article we follow also only did resizing. Our experiment went as follows. We transfer learned using VGG16 and training only the last layer with the original 331/601 data. We then trained the GAN to generate new images, giving us a dataset of 1000/1000. We then trained the last layer of a new VGG16 network.

The reason for this experiment was to see if we could recreate [4]. But more generally to see how we can use GANs and the generation of new data in applications.

4.1. **Tensorflow.** The tensorflow model was trained for 1000 epochs and the result for the loss per epoch for both generator and discriminator are presented in Figure 3. At first the model was trained with small momentum for the batch normalization but the generated images where too dark and the discriminator loss was reaching close to zero quickly. Increasing the momentum for batch normalization and introducing L1 regulization (0.0001) for the discriminator helped achieve better results. Figure 6 shows the final results of the generated images. After the training we generated new images so that total 1000/1000 (Covid/Normal) was used to train the final layer of the VGG16 classifier. The classifer achieved 0.98 validation accuracy after 20 epochs.

4.2. **PyTorch.** The PyTorch model was also trained for 1000 epochs and the loss is plotted in Figure 5. As above, we generated 669 COVID-19 images and 399 normal images so that our total new dataset consisted of 1000 images of each class. We then trained the last layer of a VGG16 network with this new dataset and got an accuracy of 0.974 after 20 epochs. Generated images of this model as give in Figure 4.

4.3. **Comparison.** The results we got differ from [4] for several reasons and in several different ways. First of all, they only got an accuracy of 0.85 when doing their first VGG16 training with 331/601 training data. This is most likely because they used a different dataset that was most likely worse. This is not so strange because they were in a hurry to get a workable dataset, while our dataset most likely better since it was curated with more care, when a big amount of data was readily available.

5. Challenges and Conclusions

One challenge we faced was that the more epochs we trained the generator with, there was less variance in the images being generated. This should theoretically be counteracted by the discriminator, because if all the images that are generated look the same, then it should be easy for the discriminator to find see this pattern, ultimately giving a greater cost for the generator during training. As illustrated in Figure 5, the loss of the generator in the PyTorch implementation gets succesively higher (even though at the same time, the images generated are getting better.) To remedy this problem, we believe it is necessary to give more parameters to the generator, or making the network deeper.

Not all details of the GAN were in the paper, and we therefore looked for other sources that helped us fill the gaps. We did not spend alot of time optimizing the hyperparameters, because it worked quite well with our first guess.

Our key result was that we were able to improve the performance of the VGG16 network by generating data via a GAN and adding it to the training set. More precisely, the accuracy increased from 0.958 to 0.974 in validation accuracy with PyTorch and to 0.98 with Tensorflow

Further studies can be done into questions of what the best hyperparameters are, what the lowest resolution of the X-ray images that still gives a good accuracy, and an approximatitive threshold for what the smallest number of X-ray images are required to give good accuracy when using the CovidGAN.

6. Ethical consideration, societal impact, potential alignment with UN SDGs

The societal impact of research in the direction of [4] could led to more accurate, cheaper and simpler diagnosis of diseases. Provided an X-ray machine and internet connection, this technology could be accesible anywhere in the world for diagnosis. This could make work for healthcare proffesionals anywhere in the world more efficient. In this way, it could help promote the third sustainable development goal set up by the UN.

Ethical considerations lie in the treatment and understanding of outliers, where the neural network is less trained. For example, does the diagnosis work as well for biological females as well and biological males? Does the dataset consist of more females than males? How does deformities in the skeletal structure affect the predicted diagnosis? These are questions that must be considered, and that are not necessary reflected in a high accuracy of the validation set.

References

odfellow2020generative
condtutorial
kaggledata
waheed2020covidgan

1. Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio, *Generative adversarial networks*, Communications of the ACM **63** (2020), no. 11, 139–144.
2. Aditya Sharma, *Conditional gan (cgan) in pytorch and tensorflow*, 2021.
3. Preet Viradiya, *Covid-19 radiography dataset*, 2021.
4. Abdul Waheed, Muskan Goyal, Deepak Gupta, Ashish Khanna, Fadi Al-Turjman, and Plácido Rogerio Pinheiro, *Covidgan: data augmentation using auxiliary classifier gan for improved covid-19 detection*, Ieee Access **8** (2020), 91916–91923.

Model: "Generator"

Layer (type)	Output Shape	Param #	Connected to
input_80 (InputLayer)	[(None, 1)]	0	
input_79 (InputLayer)	[(None, 100)]	0	
tf.compat.v1.nn.embedding_looku	(None, 1, 1)	0	input_80[0][0]
dense_53 (Dense)	(None, 65536)	6619136	input_79[0][0]
dense_54 (Dense)	(None, 1, 64)	128	tf.compat.v1.nn.embedding_lookup_
tf.nn.relu_45 (TFOpLambda)	(None, 65536)	0	dense_53[0][0]
reshape_39 (Reshape)	(None, 8, 8, 1)	0	dense_54[0][0]
reshape_38 (Reshape)	(None, 8, 8, 1024)	0	tf.nn.relu_45[0][0]
concatenate_26 (Concatenate)	(None, 8, 8, 1025)	0	reshape_39[0][0] reshape_38[0][0]
conv_transpose_1 (Conv2DTranspo	(None, 16, 16, 512)	13120000	concatenate_26[0][0]
bn_1 (BatchNormalization)	(None, 16, 16, 512)	2048	conv_transpose_1[0][0]
tf.nn.relu_46 (TFOpLambda)	(None, 16, 16, 512)	0	bn_1[0][0]
conv_transpose_2 (Conv2DTranspo	(None, 32, 32, 256)	3276800	tf.nn.relu_46[0][0]
bn_2 (BatchNormalization)	(None, 32, 32, 256)	1024	conv_transpose_2[0][0]
tf.nn.relu_47 (TFOpLambda)	(None, 32, 32, 256)	0	bn_2[0][0]
conv_transpose_3 (Conv2DTranspo	(None, 64, 64, 128)	819200	tf.nn.relu_47[0][0]
bn_3 (BatchNormalization)	(None, 64, 64, 128)	512	conv_transpose_3[0][0]
tf.nn.relu_48 (TFOpLambda)	(None, 64, 64, 128)	0	bn_3[0][0]
conv_transpose_4 (Conv2DTranspo	(None, 128, 128, 64)	204800	tf.nn.relu_48[0][0]
bn_4 (BatchNormalization)	(None, 128, 128, 64)	256	conv_transpose_4[0][0]
tf.nn.relu_49 (TFOpLambda)	(None, 128, 128, 64)	0	bn_4[0][0]
conv_transpose_6 (Conv2DTranspo	(None, 128, 128, 1)	1600	tf.nn.relu_49[0][0]

Total params: 24,045,504
Trainable params: 24,043,584
Non-trainable params: 1,920

Model: "Discriminator"

Layer (type)	Output Shape	Param #	Connected to
input_78 (InputLayer)	[(None, 1)]	0	
tf.compat.v1.nn.embedding_looku	(None, 1, 2)	0	input_78[0][0]
dense_51 (Dense)	(None, 1, 16384)	49152	tf.compat.v1.nn.embedding_lookup_
input_77 (InputLayer)	[(None, 128, 128, 1)	0	
reshape_37 (Reshape)	(None, 128, 128, 1)	0	dense_51[0][0]
concatenate_25 (Concatenate)	(None, 128, 128, 2)	0	input_77[0][0] reshape_37[0][0]
conv2d_65 (Conv2D)	(None, 128, 128, 32)	608	concatenate_25[0][0]
activation_65 (Activation)	(None, 128, 128, 32)	0	conv2d_65[0][0]
batch_normalization_65 (BatchNo	(None, 128, 128, 32)	128	activation_65[0][0]
conv2d_66 (Conv2D)	(None, 64, 64, 64)	18496	batch_normalization_65[0][0]
batch_normalization_66 (BatchNo	(None, 64, 64, 64)	256	conv2d_66[0][0]
activation_66 (Activation)	(None, 64, 64, 64)	0	batch_normalization_66[0][0]
conv2d_67 (Conv2D)	(None, 32, 32, 128)	73856	activation_66[0][0]
batch_normalization_67 (BatchNo	(None, 32, 32, 128)	512	conv2d_67[0][0]
activation_67 (Activation)	(None, 32, 32, 128)	0	batch_normalization_67[0][0]
conv2d_69 (Conv2D)	(None, 16, 16, 512)	590336	activation_67[0][0]
batch_normalization_69 (BatchNo	(None, 16, 16, 512)	2048	conv2d_69[0][0]
activation_69 (Activation)	(None, 16, 16, 512)	0	batch_normalization_69[0][0]
flatten_13 (Flatten)	(None, 131072)	0	activation_69[0][0]
dense_52 (Dense)	(None, 1)	131073	flatten_13[0][0]

Total params: 866,465
Trainable params: 0
Non-trainable params: 866,465

FIGURE 2. The discriminator and generator architectures for the tensorflow model.

fig:two

FIGURE 3. The loss per epoch for the Tensorflow model. The discriminator and generator training loss are shown in first and second graph while the genrator test loss is in the third.

fig:three

```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.label_condition_disc=nn.Sequential(nn.Embedding(n_classes, embedding_dim),
                        nn.Linear(embedding_dim, 3*128*128))
        self.model=nn.Sequential(nn.Conv2d(6, 64, 4, 2, 1, bias=False),
                        nn.LeakyReLU(0.2, inplace=True),
                        nn.Conv2d(64, 64*2, 4, 3, 2, bias=False),
                        nn.BatchNorm2d(64*2, momentum=0.1,  eps=0.8),
                        nn.LeakyReLU(0.2, inplace=True),
                        nn.Conv2d(64*2, 64*4, 4, 3,2, bias=False),
                        nn.BatchNorm2d(64*4, momentum=0.1,  eps=0.8),
                        nn.LeakyReLU(0.2, inplace=True),
                        nn.Conv2d(64*4, 64*8, 4, 3, 2, bias=False),
                        nn.BatchNorm2d(64*8, momentum=0.1,  eps=0.8),
                        nn.LeakyReLU(0.2, inplace=True),
                        nn.Flatten(),
                        nn.Dropout(0.4),
                        nn.Linear(4608, 1),
                        nn.Sigmoid()
                        )

    def forward(self, inputs):
        img, label = inputs
        label_output = self.label_condition_disc(label)
        label_output = label_output.view(-1, 3, 128, 128)
        concat = torch.cat((img, label_output), dim=1)
        #print(concat.size())
        output = self.model(concat)
        return output
```

```
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()

        self.label_conditioned_generator=nn.Sequential(nn.Embedding(n_classes, embedding_dim),
                    nn.Linear(embedding_dim, 16))
        self.latent=nn.Sequential(nn.Linear(latent_dim, 4*4*512),
                                  nn.LeakyReLU(0.2, inplace=True))
        self.model=nn.Sequential(nn.ConvTranspose2d(513, 64*8, 4, 2, 1, bias=False),
                    nn.BatchNorm2d(64*8, momentum=0.1,  eps=0.8),
                    nn.ReLU(True),
                    nn.ConvTranspose2d(64*8, 64*4, 4, 2, 1,bias=False),
                    nn.BatchNorm2d(64*4, momentum=0.1,  eps=0.8),
                    nn.ReLU(True),
                    nn.ConvTranspose2d(64*4, 64*2, 4, 2, 1,bias=False),
                    nn.BatchNorm2d(64*2, momentum=0.1,  eps=0.8),
                    nn.ReLU(True),
                    nn.ConvTranspose2d(64*2, 64*1, 4, 2, 1,bias=False),
                    nn.BatchNorm2d(64*1, momentum=0.1,  eps=0.8),
                    nn.ReLU(True),
                    nn.ConvTranspose2d(64*1, 3, 4, 2, 1, bias=False),
                    nn.Sigmoid())

    def forward(self, inputs):
        noise_vector, label = inputs
        label_output = self.label_conditioned_generator(label)
        label_output = label_output.view(-1, 1, 4, 4)
        latent_output = self.latent(noise_vector)
        latent_output = latent_output.view(-1, 512,4,4)
        concat = torch.cat((latent_output, label_output), dim=1)
        #print(concat.size())
        image = self.model(concat)
        #print(image.size())
        return image
```

FIGURE 4. The discriminator and generator architectures. fig:four

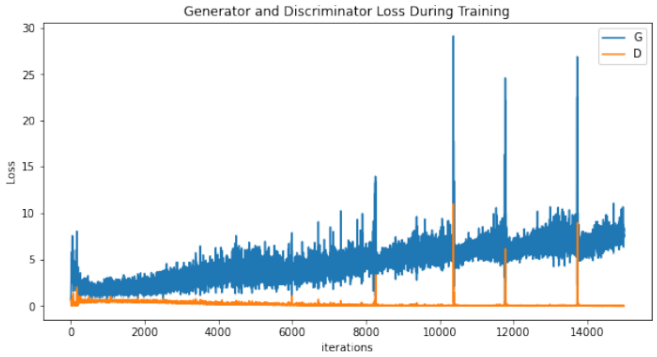

FIGURE 5. The loss of the discriminator and generator of the GAN implemented in PyTorch

fig:five

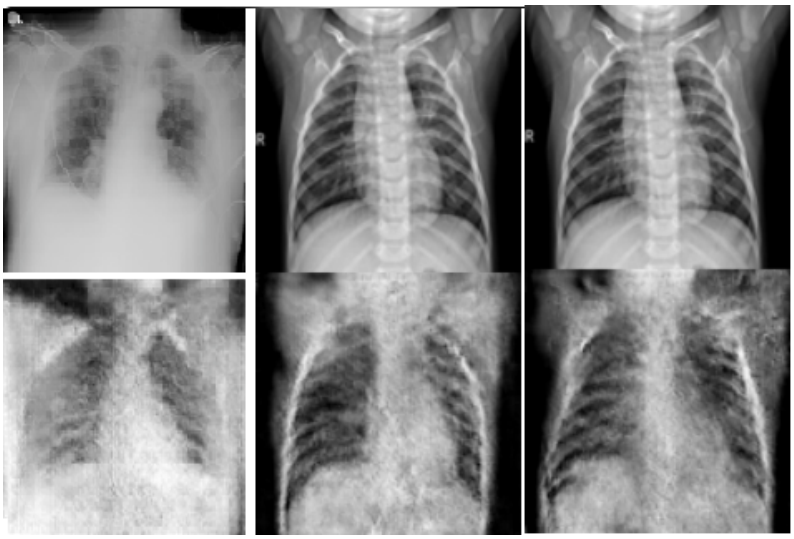

FIGURE 6. Real images(Up) vs Generated images (down) using the tensorflow model.

fig:six

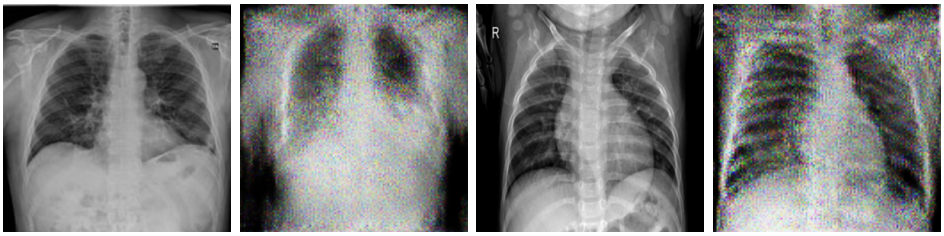

FIGURE 7. The PyTorch model We have in order, COVID-19 image, generated COIVD-19 image, normal image, generated normal image.

fig:seven