Understanding Implication of Using U-Net as Discriminator in DC-GANs

Param Chordiya
Electrical & Computer Engineering
A69026988

Ninad Ekbote
Electrical & Computer Engineering
A69026968

Abstract

Generative Adversarial Networks or GANs are based on a unique architecture where a generator and a discriminator model compete against each other to improve the generator's performance. This Project is aimed at implementing GANs with a general architecture as well as a modified version of GANs where a U-Net-based encoder-decoder architecture is used to understand how different discriminator architectures can make a difference in Generating better-quality images. We try to explore how U-net GANs capacity over vanilla GAN where it also describes the part of the generated image that looks real or fake rather than a binary output of vanilla GANs model.

1 Introduction

In this project, we are developing a DC-GAN model (with U-Net as its discriminator) that generates realistic lung X-ray images and then comparing these images with the images generated by the regular DC-GAN model. The discriminator of any Generative model tries to map the P(x) to P(Y|X) where x is features and Y represents the class. Similarly, the generator of any Generative model tries to map the P(Y) to P(X|Y). For both these mappings we require the parametric model P (x,y; theta) where theta is the weight of our model, this is the general form of any generative and discriminative model. Additionally, we would like to use the parametric model of both DC-GAN Unet and regular DC-GAN and determine which model performs better. Image generative models are very interesting, and useful, and support a wide range of applications. One of these applications is data augmentation. A good DC GAN model can be used to create better-augmented datasets. The DC GAN model we trained can be used to generate lung X-ray images and these images can be later on can be used for training different machine learning models.

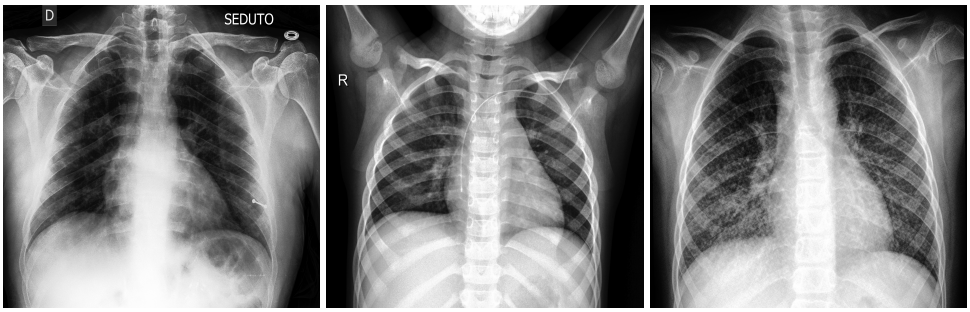
2 Related Work

DC-GANs were a huge jump from the base vanilla GAN model developed by Goodfellow et al [1]. Deep Convolutional Generative Adversarial Networks (DC-GANs) have become a cornerstone for generating high-fidelity images. Pioneering work by Radford et al [2]. established the DC-GAN architecture, utilizing convolutional layers in both the generator and discriminator. Since then, advancements have focused on improving training stability and generated image fidelity. Recent work by Isola et al [3]. introduced Pix2Pix GANs, which condition image generation on additional input channels. To address limitations in capturing fine-grained details, Schönfeld et al [4]. proposed a U-Net based discriminator for DC-GANs. Inspired by segmentation networks like U-Net, this architecture provides detailed per-pixel feedback to the generator while maintaining global coherence. This innovation has shown promise in improving the quality of generated images, particularly for high-resolution tasks.

3 Methodology

3.1 Data Preparation

The dataset utilized in this study is sourced from Kaggle. This dataset is openly available, adhering to the principles of open science and facilitating reproducibility and transparency in research. The dataset, named Covid19-dataset, is curated to address various aspects related to Covid-19. For this study, only the training subset of the Covid-19 dataset is employed. This subset is utilized to train the Generative Adversarial Networks (GANs) for generating synthetic images resembling the ones present in the dataset. The Covid-19 Chest X-rays dataset comprises images categorized into three classes: Covid, Normal, and Viral Pneumonia. Each class represents different medical conditions related to chest X-ray imaging. Specifically, the dataset includes 111 images labeled as Covid, 70 images labeled as Normal, and 70 images labeled as Viral Pneumonia for training. Additionally, the test set consists of 26 images for Covid, and 20 images each for Normal and Pneumonia. Images undergo preprocessing steps to ensure suitability for model training. These steps include resizing to 64x64 pixels, center cropping for consistency, and normalization to standardize pixel values between -1 and 1. Below are certain examples of images present in the dataset as can be seen in Fig. 1.



(a) Covid Affected Lungs (b) Normal Lungs (c) Pneumonia Affected Lungs

Figure 1: Different condition Lungs as seen in chest X-rays

3.2 About U-Net

In this project, we try to make use of U-Net as a discriminator with a DC-GAN. U-Net was initially proposed by Ronneberger et al. for biomedical image segmentation.

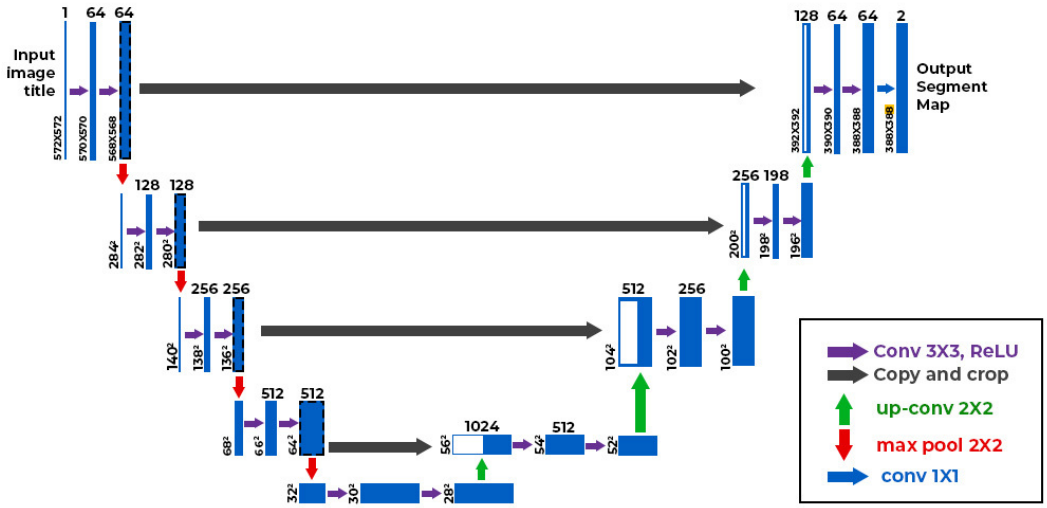


Figure 2: U-net architecture as shown by Ronneberger et. al.

We have tried to replicate the original U-Net architecture as seen in the Figure 2

3.3 Weights Initialization

In our project, the initialization of weights is a crucial step for ensuring the effectiveness and efficiency of our deep learning model. We have implemented a custom weight initialization method, which we refer to as weights_init. This method is designed to initialize the weights of convolutional layers and batch normalization layers appropriately.

We have incorporated this initialization method into our neural network model using the PyTorch framework. Below is the algorithmic representation of how we initialized the weights in our model:

Algorithm 1 Weights Initialization

```
function WEIGHTS_INIT(m)
    classname ← M.__CLASS__.__NAME__
    if classname.find('Conv') ≠ −1 then
        NN.INIT.NORMAL_(m.weight.data, 0.0, 0.02)
    else if classname.find('BatchNorm') ≠ −1 then
        NN.INIT.NORMAL_(m.weight.data, 1.0, 0.02)
        NN.INIT.CONSTANT_(m.bias.data, 0)
    end if
end function
```

This initialization strategy helps in promoting stable training dynamics and preventing issues like vanishing or exploding gradients, which are common challenges in deep learning. By setting appropriate initial weights, we ensure that the model starts training with a good initialization point, which facilitates faster convergence and better performance overall.

3.4 Training

For training a GAN-based model we need to iteratively update the generator as well as the discriminator network. This needs to be done to improve the ability of the generator to generate more realistic images. The generator (G) and discriminator (D) networks are initialized with random weights. We make use of the Binary cross-entropy (BCE) loss for loss calculations for both Generator and Discriminator. We save the losses for both G and D in arrays for specific iteration counts. And we perform this over n epochs.

In each iteration, the discriminator (D) is changed first. The gradients of D are cleared, and true images are obtained from the data loader. These genuine photos are sent through D, and the loss ($errD\_real$) is calculated based on how successfully D distinguishes between real and fake images. Gradients are then computed and used to update D according to $errD\_real$. phony images are created with noise vectors and the generator (G), then phony labels are applied to them. These fake photos are sent through D, and a new loss ($errD\_fake$) is calculated. Gradients are calculated again and used to update D based on $errD\_fake$. The overall discriminator loss ($errD$) is determined as the sum of $errD\_real$ and $errD\_fake$, and D's parameters are modified by the optimizer.

The generator (G) is updated subsequent to changing D. The created bogus images are given fake labels, and gradients of G are cleared. After passing these fake images through D, a loss ($errG$) is calculated to determine how well G tricked D. Based on $errG$, gradients are calculated and applied to update G. The optimizer is then used to update G's parameters.

The current epoch, iteration, discriminator loss, generator loss, $D(x)$ (average discriminator output for real images), and $D(G(z))$ (average discriminator output for false images) are among the training statistics that are frequently output to track progress. The generator's progress is tracked by creating photos with a fixed noise vector at regular intervals. Losses are recorded for visualization at a later time. The created photos are added to the image list so they may be examined at a later time. Following each iteration, the count of iterations is increased, and the training loop is continued until the predetermined number of epochs is attained.

Algorithm 2 Training Loop

```
img_list ← []
G_losses ← []
D_losses ← []
iters ← 0
print("Starting Training Loop...")
for epoch in range(num_epochs) do
    for i, data in enumerate(dataloader, 0) do
        netD.zero_grad()
        real_cpu ← data[0].to(device)
        b_size ← real_cpu.size(0)
        label ← torch.full((b_size,), real_label, dtype=torch.float, device=device)
        output ← netD(real_cpu).view(-1)
        errD_real ← criterion(output, label)
        errD_real.backward()
        D_x ← output.mean().item()
        noise ← torch.randn(b_size, nz, 1, 1, device=device)
        fake ← netG(noise)
        label.fill_(fake_label)
        output ← netD(fake.detach()).view(-1)
        errD_fake ← criterion(output, label)
        errD_fake.backward()
        D_G_z1 ← output.mean().item()
        errD ← errD_real + errD_fake
        optimizerD.step()
        netG.zero_grad()
        label.fill_(real_label)
        output ← netD(fake).view(-1)
        errG ← criterion(output, label)
        errG.backward()
        D_G_z2 ← output.mean().item()
        optimizerG.step()
        G_losses.append(errG.item())
        D_losses.append(errD.item())
        if (iters % 500 == 0) or ((epoch == num_epochs-1) and (i == len(dataloader)-1)) then
            with torch.no_grad():
                fake ← netG(fixed_noise).detach().cpu()
            img_list.append(vutils.make_grid(fake, padding=2, normalize=True))
        end if
        iters += 1
    end for
end for
```

3.5 Evaluation

We make use of the Inception score for evaluating our model and how good are the generated images as compared to the real images. Higher Mean represents a better ability to reproduce results for the model.

Algorithm 3 Calculate Inception Score

```
function INCEPTIONSCORE(images, batch_size = 64, splits = 10)
    inception_model ← models.inception_v3(pretrained=True, transform_input=False, aux_logits=True).cuda()
    inception_model.eval()
    up ← nn.Upsample(size=(299, 299), mode='bilinear', align_corners=False).cuda()

    transform ← transforms.Compose([
      transforms.Resize((299, 299)),
      transforms.ToTensor(),
      transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
    ])

    fake_dataset ← ImageFolder(root=images, transform=transform)
    fake_loader ← DataLoader(fake_dataset, batch_size=batch_size, shuffle=False, num_workers=4)
    preds ← []

    with torch.no_grad():
      for batch in tqdm(fake_loader, desc='Calculating Inception Score'):
        batch ← batch[0].cuda()
        batch ← up(batch)
        pred ← inception_model(batch)
        preds.append(F.softmax(pred, dim=1).cpu().numpy())

    preds ← np.concatenate(preds, axis=0)

    scores ← []
    for i ← 0 to splits − 1:
      part ← preds[i × (preds.shape[0]//splits) : (i + 1) × (preds.shape[0]//splits), :]
      kl ← part × (log(part) − log(expand_dims(mean(part, 0), 0)))
      kl ← mean(sum(kl, 1))
      scores.append(exp(kl))

    return mean(scores), std(scores)
end function
```

4 Experiments

We perform the experimentation first on vanilla implementation of DC-GANs and after generating results, we modify our discriminator to use a U-Net architecture and repeat the experimentation. Below are the experimentation results for both models.

4.1 Regular DC-GANs Implementation

We perform hyperparameter tuning on normal implementation of DC-GANs to try and optimize the performance of the model. We obtain BCE Loss for each model to understand the effect of certain parameters on our model. Some of the experimentation loss plots are shown in the plots below.

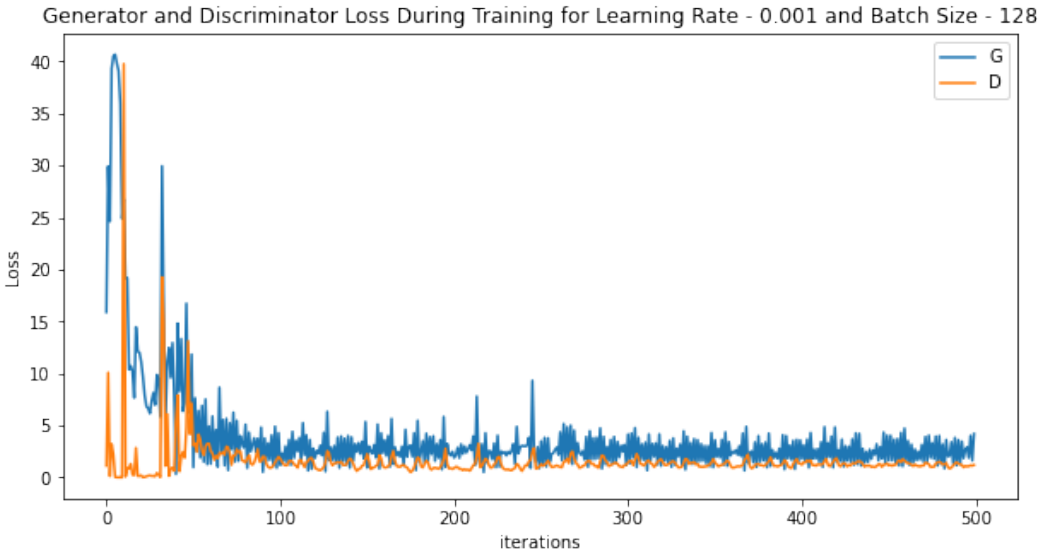


Figure 3: Batch Size = 128, learning rate = 0.001

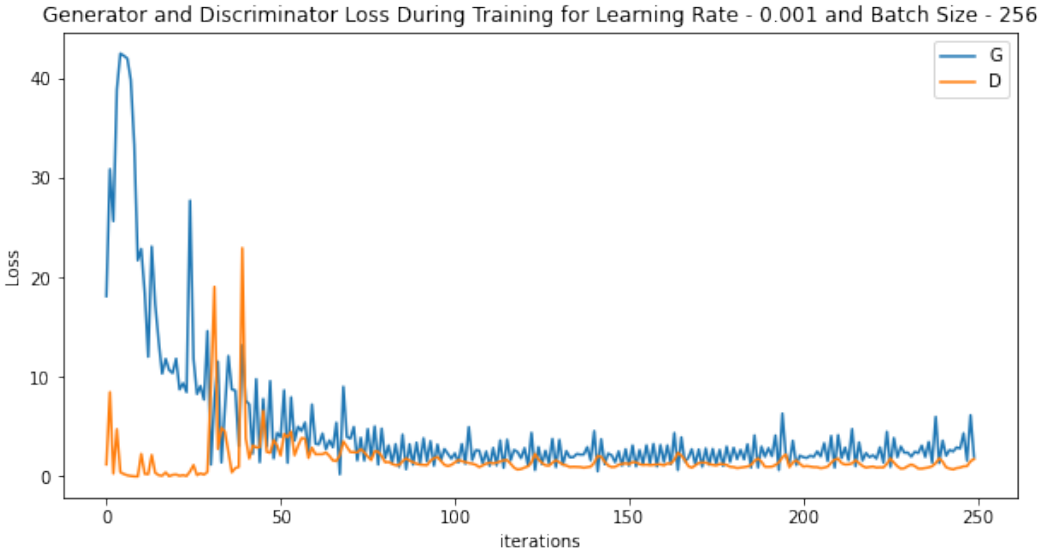


Figure 4: Batch Size = 256, learning rate = 0.001

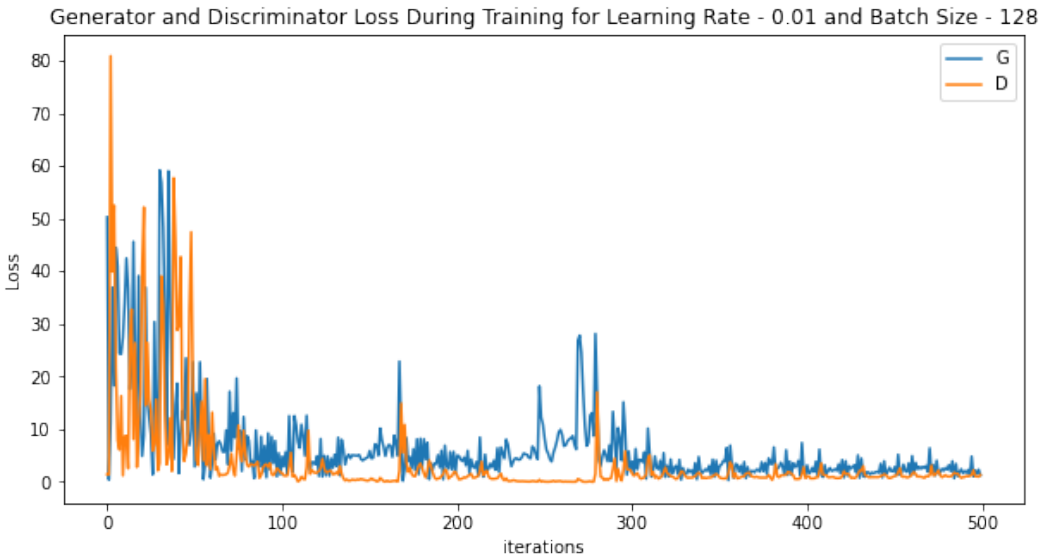


Figure 5: Batch Size = 128, learning rate = 0.01

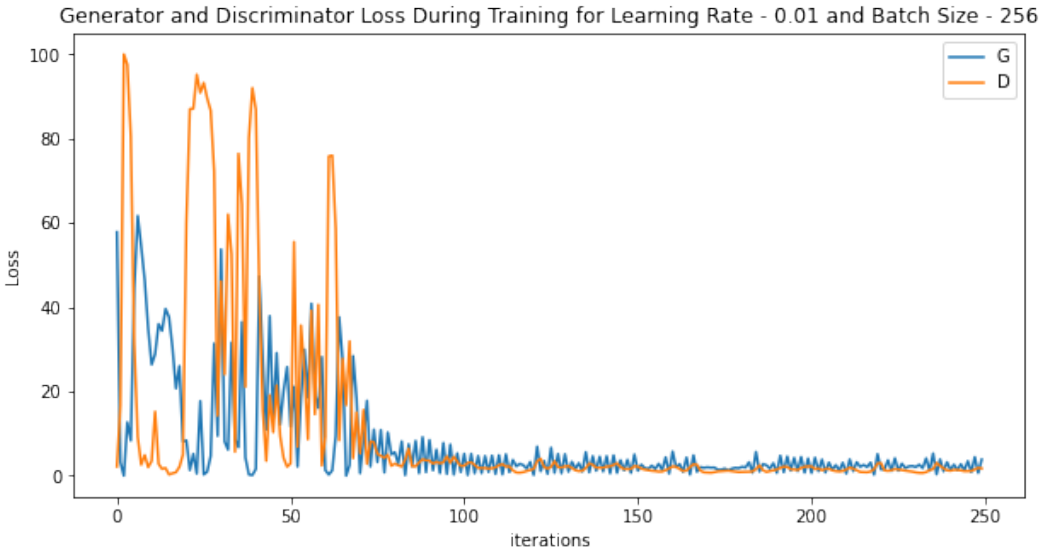


Figure 6: Batch Size =256, learning rate = 0.01

4.2 U-net based DC-GANs Implementation

We experimented with different hyperparameter tuning sets. The hyperparameter tuning sets for DC GAN U-net and regular DC GAN are different. We observed that the learning rate for DC GAN U-net cannot be set too high as the model converged at a local minimum. The hyperparameter tuning sets used are(we are also plotting the BCEloss for each hyperparameter set):

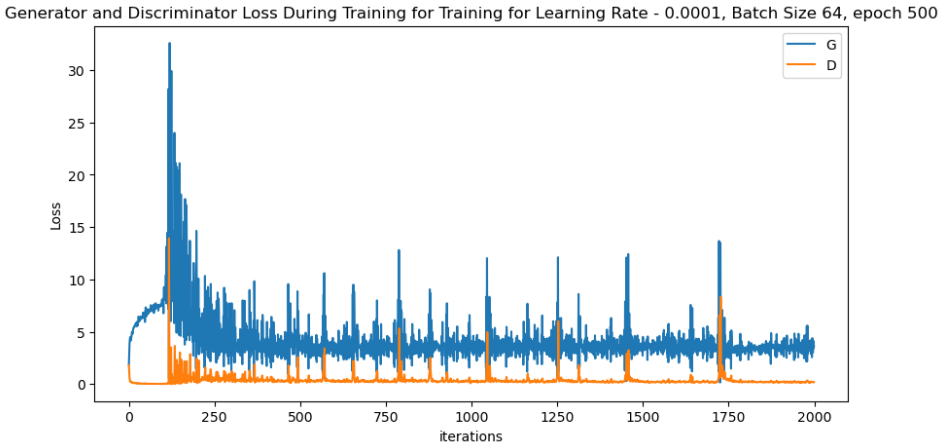


Figure 7: Batch Size = 64, learning rate = 0.0001

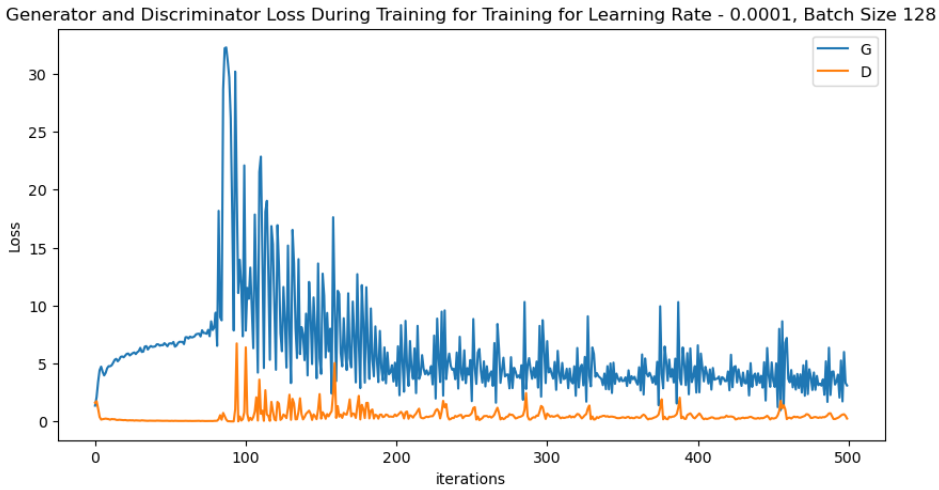


Figure 8: Batch Size = 128, learning rate = 0.0001

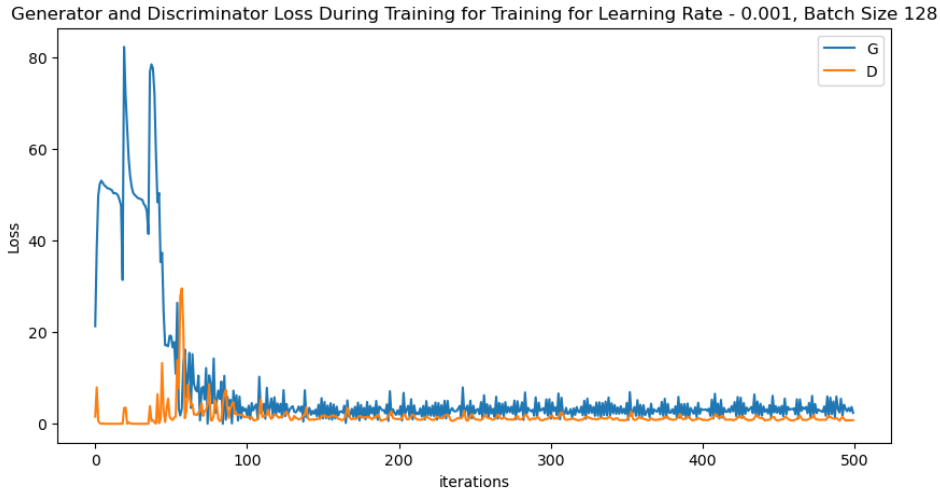


Figure 9: Batch Size = 128, learning rate = 0.001

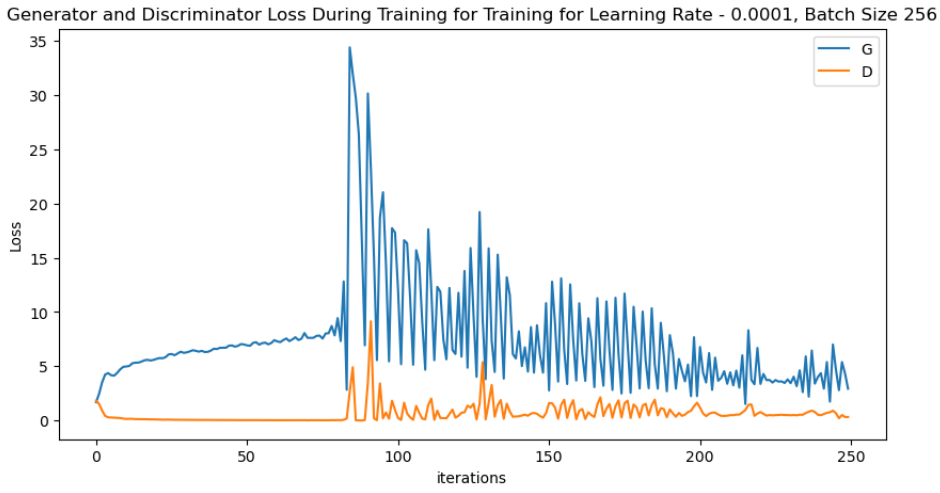


Figure 10: Batch Size =256, learning rate = 0.0001

5 Results

Based on our observations we can create a comparison of fake generated images compared to a set of real images in Figure 11 and Figure 12 where the former represents fake images generated using traditional DC-GANs model while the latter shows fake images generated using a U-Net based discriminator for our DC-GANs.

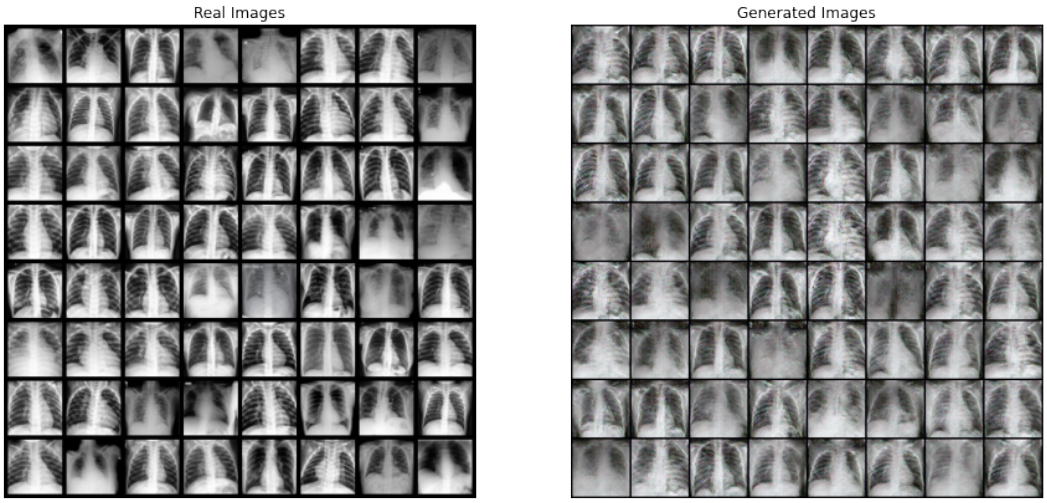


Figure 11: Comparion of DC-GANs generated images with a set of real images

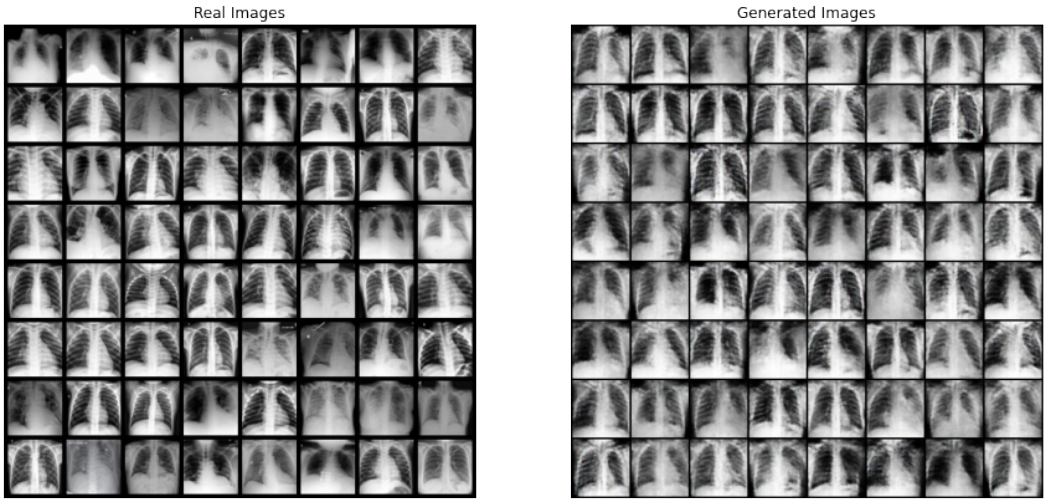


Figure 12: Comparion of U-Net based DC-GANs generated images with a set of real images

Based on our experimentation we achieved the following results on our hyperparameters as seen in Table 1 and Table 2 for traditional DC-GAN and U-net-based DC-GAN.

Sr. No.	**Learning Rate**	**Batch Size**	**Avg G Loss**	**Avg D Loss**	**Inception Score**
1	0.001	128	3.94	1.56	1.52
2	0.001	256	5.03	1.67	1.51
3	0.01	128	6.51	3.47	1.71
4	0.01	256	7.73	11.31	1.77

Table 1: Result Statistics obtained of various parameters for DC-GANs

Sr. No.	Learning Rate	Batch Size	Avg G Loss	Avg D Loss	Inception Score
1	0.0001	64	4.3237	0.3813	1.80
2	0.0001	128	6.018	0.5032	1.69
3	0.0001	256	7.4103	0.6549	1.28
4	0.001	128	8.1266	1.5843	1.81

Table 2: Result Statistics obtained of various parameters for U-Net based DC-GANs

6 Conclusion

DC GAN UNet performs much better than DC GAN. All DC GAN U-Net models gave better Inception scores than DC GAN. Hence the modification of DC GAN to DC GAN U-Net worked and DC GAN UNet gave an inception score that had lower variance than DC GAN, this means that the generated images from DC GAN U-Net are more similar to original dataset when compared to the generated images from DC GAN.

7 Supplementary Material

Here is a link to the ***GitHub*** repository for our project code.

Here is a ***Video*** summarizing our work on this project.

8 Project Contribution

- **Param Chordiya:** Coding and Development for Generator and Discriminator models for GANs, Inception Score, Dataset preprocessing plotting results.
- **Ninad Ekbote:** Coding for U-net Architecture for the Discriminator DC GANS, Dataset preprocessing, plotting results for DC GAN U-Net.
- *Report making was contributed equally by both*

References

[1] Goodfellow, Ian, et al. "Generative adversarial networks." Communications of the ACM 63.11 (2020): 139-144.

[2] Radford, Alec, et al. "Improving language understanding by generative pre-training." (2018).

[3] Isola, Phillip, et al. "Image-to-image translation with conditional adversarial networks." Proceedings of the IEEE conference on computer vision and pattern recognition. 2017.

[4] Schonfeld, Edgar, Bernt Schiele, and Anna Khoreva. "A u-net based discriminator for generative adversarial networks." Proceedings of the IEEE/CVF conference on computer vision and pattern recognition. 2020.

[5] https://www.kaggle.com/datasets/pranavraikokte/covid19-image-dataset