

Medical Image Analysis
Professor Kalluri Ramkrishna
Department of Engineering Design
Indian Institute of Technology, Madras
Lecture 50

GAN Final Demo

(Refer Slide Time: 00:15)

The screenshot shows a presentation slide titled "GANS for Medical Image Synthesis" by Kalluri Ramakrishna, Research Scholar at IIT Madras. The slide includes a diagram titled "What are GANs?" showing a "Generator" (a person with a gun) and a "Discriminator" (a person with a magnifying glass). The Generator is labeled "Wants to print fake money" and the Discriminator is labeled "Wants to distinguish between fake and real money". A text box explains that GANs are a class of machine learning techniques consisting of two networks playing an adversarial game. It states that the Generator generates dollar bills indistinguishable from real ones, and the Discriminator is forced to guess with a probability of 1/2. A note at the bottom states that both the Discriminator and Generator start from scratch and are simultaneously trained. On the right side of the slide, there are handwritten notes in red ink: ① DCGAN: Algorithm, ② Dataset, Discriminator, ③ Implementing DCGAN using Keras or PyTorch while in Google Colab.

Hi everyone, I am Ramakrishna Kalluri one of the TAS of Medical Image Analysis course I am research scholar under Doctor Ramkrishnamurthi in the Department of Engineering Design at IIT Madras. By the end of this lecture, you will get to know how to use GANS for medical image synthesis. Mostly I will discuss on implementing DCGAN algorithm or generating images having data distribution almost close to the given training data distribution mainly I have divided the entire lecture into three parts.

First one is DCGAN algorithm introduction. Second one is data set discussion that we are going to use to implement this algorithm, dataset discussion and the third part is implementing DCGAN algorithm using PyTorch deep learning framework in Google colab.

(Refer Slide Time: 01:36)

The slide is titled "What are GANs?". It features a diagram of a generator (a person with a backpack) and a discriminator (a person with a magnifying glass). The generator is labeled "The Generator" and the discriminator is labeled "The Discriminator". The generator is described as "Wants to print fake money" and the discriminator as "Wants to distinguish between fake and real money". The slide explains that GANs are a class of machine learning techniques that consist of two networks playing an adversarial game against each other. It states that in the end, the generator generates dollar bills indistinguishable from real ones, and the discriminator is forced to guess with a probability of 1/2. A note mentions that both the discriminator and generator actually start from scratch, meaning they are both randomly initialized at start and then simultaneously trained. The mathematical formula for the generator's loss is given as $\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{\text{data}}(x)} [\log D(x)] + \mathbb{E}_{z \sim p_z(z)} [\log (1 - D(G(z)))]$. The slide also mentions that GANs are incredibly sensitive to hyperparameters and provides guidelines to overcome this: replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator), and use batchnorm in both the generator and the discriminator. On the right side, there are handwritten notes in red ink: "1) DCGAN :-", "Deep Convolution", "Gene Ad.", "Neural networks", and "Network" with a checkmark.

Let us take the first part that is DCGAN Introduction this you can call this Deep Convolutional Generative Adversarial Neural Networks. GANS mainly consists of two networks first one is generator and the second one is discriminator usually, in many machine learning or deep learning problems, we will be given some data set.

Let us take such a case and discuss the importance of generator and discriminator in GANS. Generator always try to generate fake or artificial images that are almost indistinguishable from the given real data training data that means, the probability distribution of the generated or fake images is almost close to the probability distribution of the real data distribution.

(Refer Slide Time: 02:48)

This slide is identical to the one above, showing the same content about GANs. The handwritten notes on the right side are slightly different: "1) DCGAN :-", "Deep Convolution", "Gene Ad.", "Neural networks", "Network" with a checkmark, and "Min max" with a checkmark.

Now, coming to discriminator, the discriminator's task is to just distinguish the real and fake images whatever the images generated by the generator are to be rejected by the discriminator continuously. So, likewise they compete each other and will reach to a point where discriminator will not be in a position to distinguish between real and fake, that means at that point fake images generated by the generator are almost close to the real images.

This is the loss function or you can call objective function that is used by the GANS with respect to generator this function will be minimized whereas, with respect to the discriminator this function will be maximize. That is why people call this one as min max algorithm. Sorry, min max objective function that is the basic idea of GANS.

In summary, generator always try to generate the fake images that are almost similar to the data that is given in the data set or we can say training data. Discriminator's task is to always to reject the images generated by the generator whereas, to accept the images that are generally present in the real images or given data set.

(Refer Slide Time: 04:28)

GANs are incredibly sensitive to hyperparameters. To overcome following guidelines will help

Architecture guidelines for stable Deep Convolutional GANs

- Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).
- Use batchnorm in both the generator and the discriminator.
- Remove fully connected hidden layers for deeper architectures.
- Use ReLU activation in generator for all layers except for the output, which uses Tanh.
- Use LeakyReLU activation in the discriminator for all layers.

GENERATOR

1. CNN Transposed Layers or fractional Strided CNN layers
2. BatchNorm
3. ReLU activation Functions for all the layers except for the output which uses Tanh

Input: Latent Vector z drawn from standard normal distribution.

Post what?

D

GANS are incredibly sensitive to hyper parameters to overcome following guidelines will help in implementing the DCGAN. These are the 5 guidelines suggested by the others in the paper deep convolutional generative adversarial neural networks.

(Refer Slide Time: 04:48)

convolutions (generator).

- Use batchnorm in both the generator and the discriminator.
- Remove fully connected hidden layers for deeper architectures.
- Use ReLU activation in generator for all layers except for the output, which uses Tanh.
- Use LeakyReLU activation in the discriminator for all layers.

GENERATOR

1. CNN Transposed Layers or fractional Strided CNN layers
2. BatchNorm
3. ReLU activation Functions for all the layers except for the output which uses Tanh

Input: Latent Vector z drawn from standard normal distribution.
Output: RGB image (3,64,64)

Discriminator

1. Strided CNN
2. BatchNorm
3. Leaky ReLU activation Functions

Input: RGB image = (3,64,64)
Output: Scalar probability score (real value) that indicates whether the input is from the real data distribution.

Handwritten notes on the right side of the slide:

- Now \rightarrow All pooling layers removed
- DCGAN
- ① CNN \rightarrow Strided
- Transposed CNN layers

Now, if you observe clearly in this entire DCGAN algorithm, generator will be like this, whereas discriminator will be like this. If you observe clearly go to the first point replace any pooling layers with strided convolutions and fractional strided convolutions with generator.

Here if you observe generators and discriminators will only contain only CNN layers in discriminator, the CNN are strided convolutions only, strided CNN layers whereas in generator we will use fractional strided convolution or in other way transposed convolutions transposed CNN layers. That means all the max pooling layers or minimum pooling layers or average pooling if any of the all the pooling layers were removed all the pooling layers are removed all the pooling layers removed.

Now get back to using batch normalization in both generator and discriminator we will use batch normalization in both generator and discriminator. Next thing is remove fully connected layers also from deeper architectures that means, we will avoid using fully connected layers also.

(Refer Slide Time: 06:26)

The slide displays a GAN architecture diagram with the following components:

- Generator:**
 - 1. Strided CNN
 - 2. BatchNorm
 - 3. Leaky ReLU activation Functions
 - Input: Latent Vector z drawn from standard normal distribution.
 - Output: RGB image (3,64,64)
- Discriminator:**
 - 1. Strided CNN
 - 2. BatchNorm
 - 3. Leaky ReLU activation Functions
 - Input: RGB image = (3,64,64)
 - Output: Scalar probability score (real value) that indicates whether the input is from the real data distribution.

Step By Step Process:

- 1) Input parameters set (Hyperparameters)
- 2) Dataset preparation and Data loader
- 3) Weight initialization
- 4) Generator Network Design, Discriminator Network Design, Loss functions
- 5) Training loop in detail.

Handwritten Notes:

- ① CNN → Strided
- Transposed Conv layers
- only CNN layers - BN

The slide provides detailed instructions for the GAN architecture:

- Use batchnorm in both the generator and the discriminator.
- Remove fully connected hidden layers for deeper architectures.
- Use ReLU activation in generator for all layers except for the output, which uses Tanh.
- Use LeakyReLU activation in the discriminator for all layers.

Generator Details:

- 1. CNN Transposed Layers or fractional Strided CNN layers
- 2. BatchNorm
- 3. ReLU activation Functions for all the layers except for the output which uses Tanh
- Input: Latent Vector z drawn from standard normal distribution.
- Output: RGB image (3,64,64)

Discriminator Details:

- 1. Strided CNN
- 2. BatchNorm
- 3. Leaky ReLU activation Functions
- Input: RGB image = (3,64,64)
- Output: Scalar probability score (real value) that indicates whether the input is from the real data distribution.

Handwritten Notes:

- Max → All other layers
- Removed
- Discard
- ① CNN → Strided
- Transposed Conv layers
- only CNN layers

So, the summary is only CNN layers are present only CNN layers are present in the network and batch normalization is present everywhere both generator and discriminator and the last thing is we will use ReLU activation function in generator for all the layers except for the output which uses tan(h).

(Refer Slide Time: 06:47)

The screenshot shows a presentation slide with a table of contents on the left and a list of points on the right. The table of contents includes:

- 2) Dataset preparation and Data loader
- 3) Weight initialization
- 4) Generator Network Design, Discriminator Network Design, Loss functions
- 5) Training loop in detail.
- 6) Evaluation Of Results

The right side of the slide contains the following text:

- Both models are trained simultaneously, and since each is penalized when the other does well, balancing their learning rates is critical. If one model learns too fast, its feedback becomes useless to the other.
- We've seen how our discriminator and our generator need to learn alongside each other at compatible rates. We've also seen how to use the loss graphs to adjust their relative learning rates.

We'll stop here for now, but there's more we can do to fine tune the learning of our model aside from learning rates:

Rather than increasing LR, we can try increasing the number of steps per epoch. This might help to improve resolution and address the fuzziness we saw.

We can also train for longer, and add early stopping, to make sure we're reaching a point of convergence.

Handwritten notes on the right side of the slide include:

- G: $Act \rightarrow ReLU, Act$
- last layer $\rightarrow Tanh$ (circled)
- A graph showing a step function from -1 to +1.
- $-1 \rightarrow +1$ (circled)
- Normalized images (with an arrow pointing to the graph)

In generator all the other layers use ReLU activation function except for the last layer where we use the Tan h function. The ReLU range of this tan(h) activation function is -1 to +1 that means, if the last layer of the entire output at the last layer generated will consist of all the pixel values will in between -1 to +1 more or less like normalized images.

(Refer Slide Time: 07:29)

The screenshot shows a presentation slide with a list of points on the left and handwritten notes on the right. The list of points includes:

- Both models are trained simultaneously, and since each is penalized when the other does well, balancing their learning rates is critical. If one model learns too fast, its feedback becomes useless to the other.
- We've seen how our discriminator and our generator need to learn alongside each other at compatible rates. We've also seen how to use the loss graphs to adjust their relative learning rates.

We'll stop here for now, but there's more we can do to fine tune the learning of our model aside from learning rates:

Rather than increasing LR, we can try increasing the number of steps per epoch. This might help to improve resolution and address the fuzziness we saw.

We can also train for longer, and add early stopping, to make sure we're reaching a point of convergence.

Handwritten notes on the right side of the slide include:

- Normalized images (with an arrow pointing to the graph)
- Real data \rightarrow Normalized
- Real data \rightarrow Discriminator
- fake data \rightarrow

Why we generate normalized images because real data are the data set of the training data or the data that is generated by the this is generated by the generator and this is the original ReLU train data, before passing it before passing the real data to discriminator we have normalized all the training data images nominalized all the training data images to -1 to +1 range.

(Refer Slide Time: 08:04)

The slide shows a presentation interface with a blackboard for handwritten notes. The text on the slide includes:

- Rather than increasing LR, we can try increasing the number of steps per epoch. This might help to improve resolution and address the fuzziness we saw.
- We can also train for longer, and add early stopping, to make sure we're reaching a point of convergence.

Handwritten notes on the blackboard:

- Real data: $\{-1, +1\}$ Normalized.
- Take dot a: $[-1, +1]$ ✓
- Tanh

So, now, whatsoever the images that are generated by the generator also to be present in the same format as that of the data that is being fed into discriminator that means, the generator data by the generator also should be should contain all the all its pixel values should be in the range of -1 to +1 that is why we have used tan(h) function at the end of lost layer in the generator.

(Refer Slide Time: 08:41)

The slide shows a presentation interface with a blackboard for handwritten notes. The text on the slide includes:

- GANs are incredibly sensitive to hyperparameters. To overcome following guidelines will help
- Architecture guidelines for stable Deep Convolutional GANs:
 - Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).
 - Use batchnorm in both the generator and the discriminator.
 - Remove fully connected hidden layers for deeper architectures.
 - Use ReLU activation in generator for all layers except for the output, which uses Tanh.
 - Use LeakyReLU activation in the discriminator for all layers.
- GENERATOR:
 - 1. CNN Transposed Layers or fractional Strided CNN layers
 - 2. BatchNorm
 - 3. ReLU activation Functions for all the layers except for the output which uses Tanh
- Discriminator:
 - 1. Strided CNN

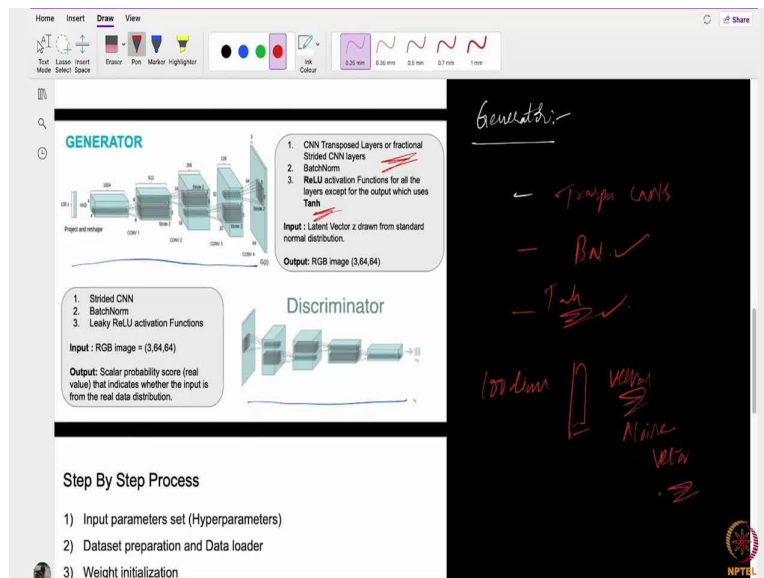
Handwritten notes on the blackboard:

- DCGAN: ✓
- Stable Training
- Not: All deep layers
- Leaky ReLU
- DCGAN: ✓
- 1. CNN? Strided

And the last point is use leaky ReLU function in the discriminator for all the layers that have been in discriminate section all the layers will contain leaky ReLU activation function.

So, these are the 5 guidelines one has to keep in mind while implementing this DCGAN algorithm. So, that the network will be somewhat less serious due to the hyper parameters so, that we can expect a stable training, stable training that is the summary.

(Refer Slide Time: 09:19)



Now, let us speak about generator for some time. Generator contains only transpose CNNs transpose CNNs and the second thing is BatchNormalization is present and all ReLU activation functions for all the layer except for the output which uses tan(h) activation function.

Now what is the input to the generator and what is the output of the generator let us see. For the generator in this case they have used 100 dimensional vector, 100 dimensional vector some random noise vector will be fed into a generator.

(Refer Slide Time: 10:16)

The slide displays the 'Step By Step Process' for training a Generative Adversarial Network (GAN). The process is outlined in six steps:

- 1) Input parameters set (Hyperparameters)
- 2) Dataset preparation and Data loader
- 3) Weight initialization
- 4) Generator Network Design, Discriminator Network Design, Loss functions
- 5) Training loop in detail.
- 6) Evaluation Of Results

Handwritten notes on the right side of the slide include:

- 100 dim (referring to the latent vector z)
- $(3, 64 \times 64)$ (referring to the output image dimensions)
- $\text{Real} = (64, 64)$ (referring to the real image dimensions)

So that output generated by this generator will be a 3 channel, 64×64 image that is we can say RGB 64 height and 64 width.

(Refer Slide Time: 10:41)

The slide illustrates the architecture of the Generator and Discriminator. The Generator is described as follows:

- 1. CNN Transposed Layers or fractional Strided CNN layer
- 2. BatchNorm
- 3. ReLU activation Functions for all the layers except for the output which uses Tanh

The input to the Generator is a Latent Vector z drawn from standard normal distribution. The output is an RGB image $(3, 64, 64)$.

The Discriminator is described as follows:

- 1. Strided CNN
- 2. BatchNorm
- 3. Leaky ReLU activation Functions

The input to the Discriminator is an RGB image $(3, 64, 64)$. The output is a Scalar probability score (real value) that indicates whether the input is from the real data distribution.

Handwritten notes on the right side of the slide include:

- 100 dim (referring to the latent vector z)
- $(3, 64 \times 64)$ (referring to the output image dimensions)

Now observe here in generator we have fed under dimensional noise vector and all the convolutional transpose convolution transpose convolution transpose layers are used at the end we will get 3 channel having height 64 similarly width also 64 image has been generated.

(Refer Slide Time: 11:04)

GENERATOR

1. CNN Transposed Layers or fractional Strided CNN layers
2. BatchNorm
3. ReLU activation Functions for all the layers except for the output which uses Tanh

Input : Latent Vector z drawn from standard normal distribution.

Output: RGB image (3,64,64)

Discriminator

1. Strided CNN
2. BatchNorm
3. Leaky ReLU activation Functions

Input : RGB image = (3,64,64)

Output: Scalar probability score (real value) that indicates whether the input is from the real data distribution.

Step By Step Process

- 1) Input parameters set (Hyperparameters)
- 2) Dataset preparation and Data loader

Handwritten Diagram:

Discriminator

Real: $(3, 64, 64)$

Fake: \rightarrow

$D \rightarrow$

Prob. score

Now, let us speak up discriminator, now discriminator will take input inputs as either real images or fake images that means, the input to the discriminator will be a 3 channel 64 plus 64 image, that means, RGB any of these images either real or fake images, fake image generated by the generator or real images that are brought from original data set will be fed it to the discriminator and now, the discriminator will give the probability score, probability score.

(Refer Slide Time: 11:52)

GENERATOR

1. CNN Transposed Layers or fractional Strided CNN layers
2. BatchNorm
3. ReLU activation Functions for all the layers except for the output which uses Tanh

Input : Latent Vector z drawn from standard normal distribution.

Output: RGB image (3,64,64)

Discriminator

1. Strided CNN
2. BatchNorm
3. Leaky ReLU activation Functions

Input : RGB image = (3,64,64)

Output: Scalar probability score (real value) that indicates whether the input is from the real data distribution.

Step By Step Process

- 1) Input parameters set (Hyperparameters)
- 2) Dataset preparation and Data loader
- 3) Weight initialization
- 4) Generator Network Design, Discriminator Network Design, Loss

Handwritten Diagram:

Discriminator

Real: $(3, 64, 64)$

Fake: \rightarrow

$D \rightarrow$

Prob. score

$P(\text{Real}) = \dots$

The score indicates whether this particular any of these RGB belongs to real data that is given image being real that is what this score indicates.

(Refer Slide Time: 12:06)

The slide titled "Discriminator" illustrates a neural network architecture for image classification. It lists the following components:

1. Strided CNN
2. BatchNorm
3. Leaky ReLU activation Functions

Input: RGB image = (3,64,64)
Output: Scalar probability score (real value) that indicates whether the input is from the real data distribution.

Step By Step Process

- 1) Input parameters set (Hyperparameters)
- 2) Dataset preparation and Data loader
- 3) Weight initialization
- 4) Generator Network Design, Discriminator Network Design, Loss functions
- 5) Training loop in detail.
- 6) Evaluation Of Results

Handwritten notes on the right side of the slide include:

- $p(\text{Real}) = \frac{1}{N}$
- A diagram showing a circle with a plus sign and a circle with a minus sign, labeled "Real" and "Fake" respectively.
- Text: "Real" and "Fake" with arrows pointing to the respective circles.
- Text: "Real" and "Fake" with arrows pointing to the respective circles.

Suppose, the output score for any given image is very near to 1 what it means? It is a real image. There are more chances that the image that has been fed is can be considered as a real image.

Suppose this value is very near to 0 that means, there are very less chances for the given image to be real what we can conclude now, maybe that image is fake images, fake image. This is how discriminator and generator will function.

(Refer Slide Time: 12:45)

The screenshot shows a Jupyter Notebook titled "Copy of Chest X-Ray ShenanGANs (DCGAN).ipynb". The code in the notebook is as follows:

```
import os
import random
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as dset
import torchvision.transforms as transforms
import torchvision.utils as vutils
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML

# Set random seed for reproducibility
manualSeed = 999
#manualSeed = random.randint(1, 10000) # use if you want new results
random.seed(manualSeed)
torch.manual_seed(manualSeed)

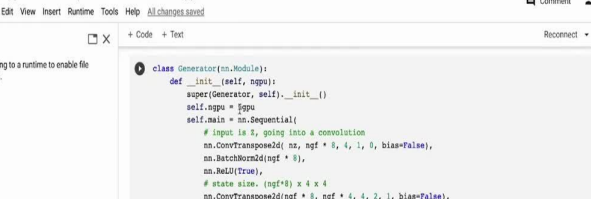
#<torch._C_Generator at 0x70ef7bb2ee50>

[ ] # Load dataset
!wget https://data.mendeley.com/datasets/rcub/jbr9s///files/011b2c02-11c3-4424-996e-1be3f3660990/ZhangLabData.zip
!unzip /content/ZhangLabData.zip
datasetroot = '/content/Celldata/chest_xray/train'

infiating: CellData/OCT/test/DME/DME-1839247-1.jpeg
infiating: CellData/OCT/test/DME/DME-5961731-1.jpeg
infiating: CellData/OCT/test/DME/DME-2328103-1.jpeg
infiating: CellData/OCT/test/DME/DME-1805165-1.jpeg
infiating: CellData/OCT/test/DME/DME-1805165-1.jpeg
infiating: CellData/OCT/test/DME/DME-1805165-1.jpeg
14s completed at 2:00 AM
```

Now, let me show you in what way generator and discriminator are designed in the network using these 5 guidelines. Suppose this code has been designed for this chest X ray image

(Refer Slide Time: 13:16)



```
class Generator(nn.Module):
    def __init__(self, ngpu):
        super(Generator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is 1, going into a convolution
            nn.ConvTranspose2d(1, ngf * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            # state size. (ngf*8) x 4 x 4
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # state size. (ngf*4) x 8 x 8
            nn.ConvTranspose2d(ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # state size. (ngf*2) x 16 x 16
            nn.ConvTranspose2d(ngf * 2, ngf, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.ReLU(True),
            # state size. (ngf) x 32 x 32
            nn.ConvTranspose2d(ngf, 1, 4, 2, 1, bias=False),
            nn.Tanh()
        )

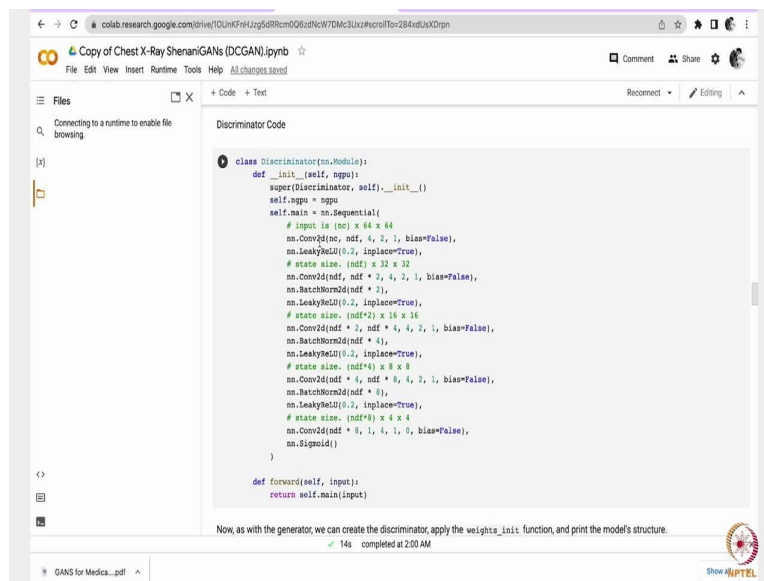
    def forward(self, input):
        return self.main(input)
```

Now, we can instantiate the generator and apply the weights_init function. Check out the printed model to see how the generator object is

1/43 completed at 2:00 AM

So in generator almost all the convolutional 2D, transpose 2D networks are used at the same time BatchNormalization is used and ReLU activation functions are used at every layer except at the output layer where they have used tan(h) activation function. Here if you observe clearly I will get back to what is this noise vector and all the stuff later on just the basic idea this is or in another way we can see let me tell you.

(Refer Slide Time: 14:26)



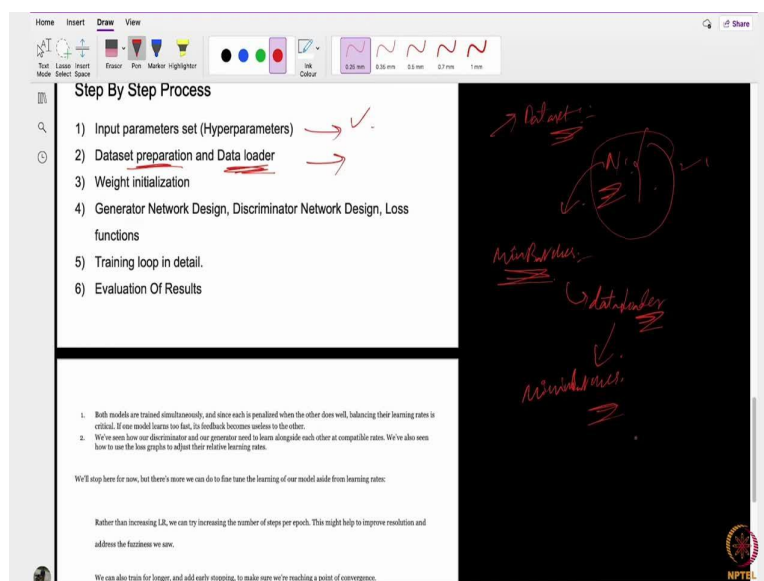
```
class Discriminator(nn.Module):
    def __init__(self, ngpu):
        super(Discriminator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is (nc) x 64 x 64
            nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # state size: (ndf) x 32 x 32
            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),
            # state size: (ndf*2) x 16 x 16
            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # state size: (ndf*4) x 8 x 8
            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),
            # state size: (ndf*8) x 4 x 4
            nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, input):
        return self.main(input)
```

Now let us get back to discriminator code. Here if you observe they have used strided convolution only just normal CONV 2D layers and leaky ReLU, leaky ReLU, leaky ReLU, leaky ReLU and CONV 2D. Everywhere they have used leaky ReLU activation functions. But at the end, they have use sigmoid that is for outputting the probability score that is either near to 0 or near to 1.

That is will having sigma, since sigmoid activation functions, output values that lie between 0 to 1. Just as a basic idea discriminator has been designed like this. Similarly, generator has been designed like this, according to the 5 rules framed by the others in the DCGAN paper.

(Refer Slide Time: 15:15)



Step By Step Process

- 1) Input parameters set (Hyperparameters)
- 2) Dataset preparation and Data loader
- 3) Weight initialization
- 4) Generator Network Design, Discriminator Network Design, Loss functions
- 5) Training loop in detail.
- 6) Evaluation Of Results

Both models are trained simultaneously, and since each is penalized when the other does well, balancing their learning rates is critical. If our model learns too fast, its feedback becomes useless to the other.

We've seen how our discriminator and our generator need to learn alongside each other at compatible rates. We've also seen how to use the log graphs to adjust their relative learning rates.

We'll stop here for now, but there's more we can do to fine tune the learning of our model aside from learning rates:

Rather than increasing LR, we can try increasing the number of steps per epoch. This might help to improve resolution and address the fuzziness we saw.

We can also train for longer, and add early stopping, to make sure we're reaching a point of convergence.

Now, let us see what exactly we are going to discuss later on. First of all, we will set some hyper parameter values next thing is creating the data set preparation and data loader preparation. In deep learning, we will first prepare the data set after downloading the data set whatsoever the necessary transformations are to be done please do it and later on, we will suppose the data set contains some N images.

Total N images are there in the data set you should have directly feeding these N total of N images into the model directly we prefer mini batches we prefer feeding mini batches rather than feeding the whole data set we will prefer feeding mini batches dataset. So, that is where this data loader part will come. That means, it will help you to load data in mini batches, mini batches from the entire data set.

(Refer Slide Time: 16:31)

The image shows a presentation slide titled "Step By Step Process" with a list of 6 steps. The steps are:

- 1) Input parameters set (Hyperparameters)
- 2) Dataset preparation and Data loader
- 3) Weight initialization
- 4) Generator Network Design, Discriminator Network Design, Loss functions
- 5) Training loop in detail.
- 6) Evaluation Of Results

Handwritten notes on the right side of the slide include:

- Red arrows pointing from steps 1, 2, and 3 to a central point.
- A blue bracket grouping steps 4 and 5.
- Handwritten text: "Dataset", "Minibatches", "Discriminator", "Minibatches", and "Weights and Biases".
- A diagram showing a circle with "N" inside, and arrows pointing to "Dataset" and "Minibatches".
- A box labeled "Weights and Biases" with a checkmark.

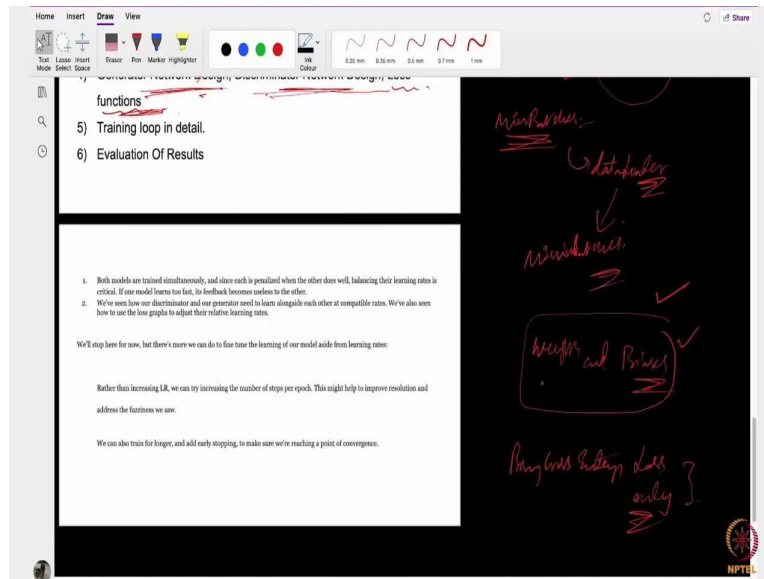
Below the list of steps, there is a small text box with the following content:

1. Both models are trained simultaneously, and since each is penalized when the other does well, balancing their learning rates is critical. If one model learns too fast, its feedback becomes useless to the other.

2. We've seen how our discriminator and our generator need to learn alongside each other at compatible rates. We've also seen how to use the loss graphs to adjust their relative learning rates.

We'll stop here for now, but there's more we can do to fine tune the learning of our model aside from learning rates:

Rather than increasing LR, we can try increasing the number of steps per epoch. This might help to improve resolution and



After that, we will prepare the model after preparing this model we should initialize the weights and biases, weights and biases. Further in simple way, we call it as weight initialization step we will follow some criteria, to initialize these weights end of the day even weights are also just numbers only, but we will have we will put some restrictions like we will keep all these weights or these numbers we will take it from either some normal distribution or some particular probability distribution that is up to us.

But in the paper they have used standard normal distribution only what the code whichever I show you also they have used the normal distribution only. Weight initialization and the next step is generator network design, discriminator network design, loss functions as I said earlier, they are just prepare a weight initialization function later on after designing these models after designing generator network and discriminator network, those initialization weight initialization function will be called on here so that the weights and biases of these two networks will be initialized using this function.

Later on they have designed the loss functions also end of the day here we are using the classification only be the particular image being real or fake that means, we just here using binary cross entropy loss only, binary cross entropy loss only. Later on when we go and see the code, I will explain you there I will show you what sort of loss function they have used binary cross entropy.

Next thing is training loop since the network is designed data loader, data everything is designed and the loss functions are also mentioned. Now, our target is to train start the

training that is where this training loop in detail will be explained later on we will evaluation of the result.

At the end of the training what we will do is using the particular model we will generate some images from the GAN sorry generator and I will compare those images with the original real data images. This is what the entire process we are going to see while implementing the while going through the code.

(Refer Slide Time: 19:03)

The screenshot shows a presentation slide with a list of steps on the left and a handwritten diagram on the right. The steps are:

- 1) Input parameters set (Hyperparameters)
- 2) Dataset preparation and Data loader
- 3) Weight initialization
- 4) Generator Network Design, Discriminator Network Design, Loss functions
- 5) Training loop in detail.
- 6) Evaluation Of Results

The handwritten diagram on the right is titled 'Discriminator' and shows a box labeled 'discriminator' with arrows pointing to it from 'real' and 'fake' inputs. Below the box, there are two rows of handwritten text: '(real, 1)' and '(fake, 0)', with a 'BCE' label next to them.

Below the list of steps, there is a small text box with the following content:

1. Both models are trained simultaneously, and since each is penalized when the other does well, balancing their learning rates is critical. If one model learns too fast, its feedback becomes useless to the other.

2. We've seen how our discriminator and our generator need to learn alongside each other at compatible rates. We've also seen how to use the loss graphs to adjust their relative learning rates.

We'll stop here for now, but there's more we can do to fine tune the learning of our model aside from learning rates:

Rather than increasing LR, we can try increasing the number of steps per epoch. This might help to improve resolution and address the fuzziness we saw.

We can also train for longer, and add early stopping, to make sure we're reaching a point of convergence.

Before that let us focus on discriminator training, discriminator training where here real images and fake images both will be fed into the discriminator. Now its task is to classify. So, what it will do is? Here suppose the given input is real image, real image the output the target value will be kept as 1 whereas if you provide a fake image, I mean the image generated by the generator at the time the target value is 0. This is how the loss function is designed there. And end of the day we will calculate that corresponding binary cross entropy loss only.

But given these targets or maintaining this target is very important here. While training the discriminator we will feed both real images and fake images real images from the training data, fake images from the images generated by the generator.

So, how it will so, how it will train leads? The lost will be suppose real input is given as input at the time keep the target values as 1 whereas, if you feed the fake images at the time keep them target value as 0 this is how visual prepare.

(Refer Slide Time: 20:40)

Step By Step Process

- 1) Input parameters set (Hyperparameters)
- 2) Dataset preparation and Data loader
- 3) Weight initialization
- 4) Generator Network Design, Discriminator Network Design, Loss functions
- 5) Training loop in detail
- 6) Evaluation Of Results

Both models are trained simultaneously, and since each is penalized when the other does well, balancing their learning rates is critical. If one model learns too fast, its feedback becomes useless to the other.

We've seen how our discriminator and our generator need to learn alongside each other at compatible rates. We've also seen how to use the loss graphs to adjust their relative learning rates.

We'll stop here for now, but there's more we can do to fine tune the learning of our model aside from learning rates:

Rather than increasing LR, we can try increasing the number of steps per epoch. This might help to improve resolution and address the formatness issue.

Handwritten notes on blackboard:
Generator Training
①: X'ed images
[fake images, 1]
BCE

Now, coming back to generator training while training the generator, we will not consider real images, we will not consider real images. So, we only have fake images only, fake images. Here suppose, this is fake image, but here we will give its corresponding target value as 1 that means, we are while training the generator we are fooling the discriminator in such a way that whatsoever the images generated by the generator will their targets are kept as 1 that means, we are fooling the discriminator. This is how the target values are chosen and later on again they will use binary cross entropy loss only. Please keep these points in mind.

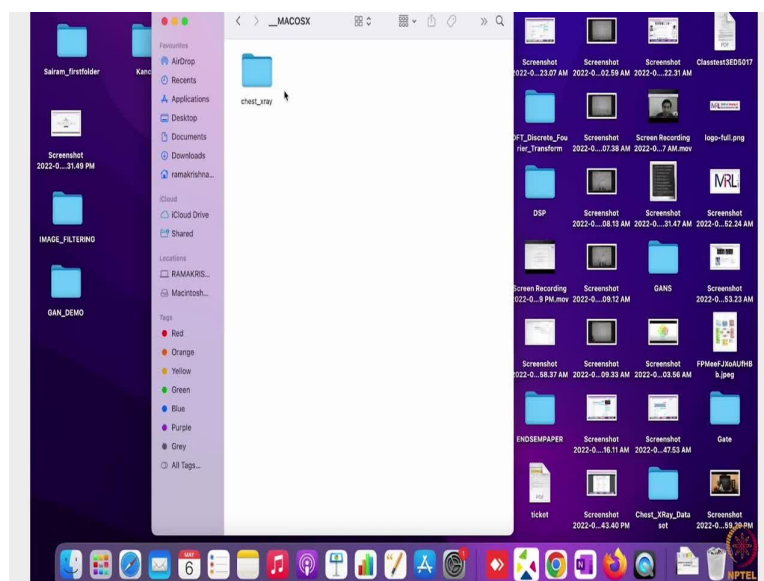
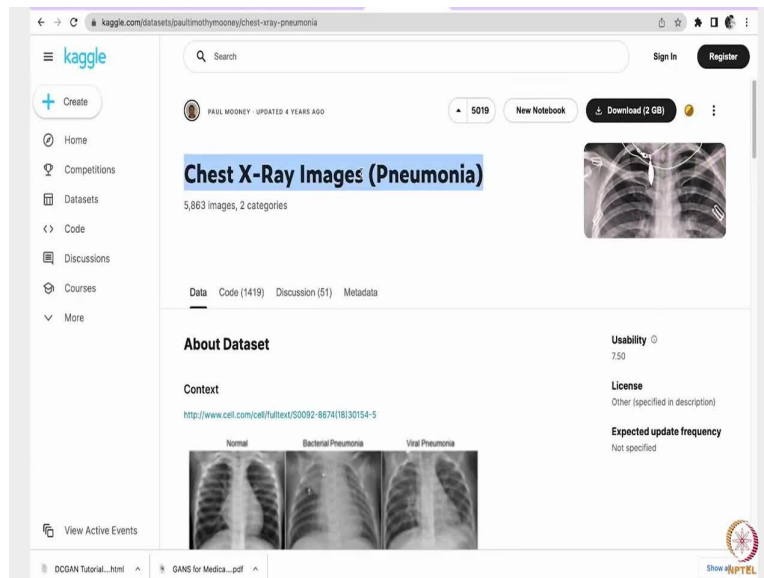
(Refer Slide Time: 21:41)

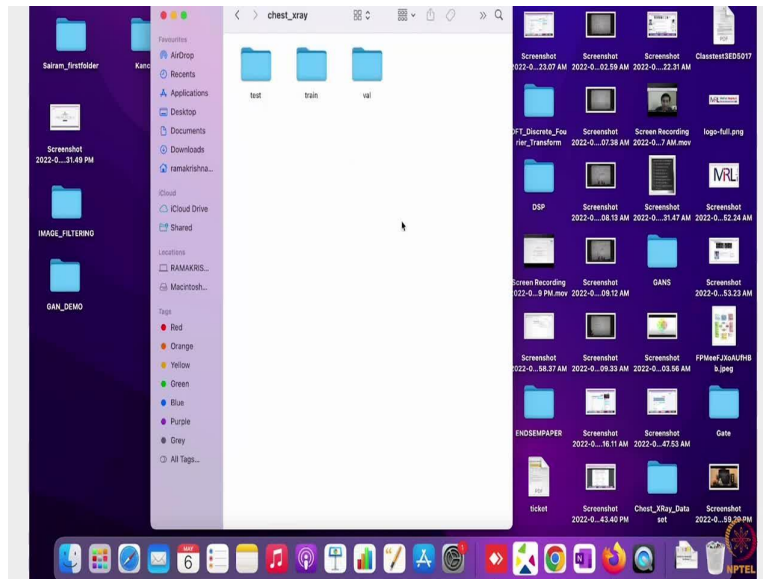
Handwritten notes on blackboard:
Data set:
Chest x-ray Dataset
- Pneumonia and
- Abnormal
Kaggle

While going through the code, I will tell you I will show you now, let us discuss about the dataset that we are going to use to implement this algorithm data set that is chest X ray data

set, chest X ray data set it contains pneumonia and normal of 2 classes of images out there and the data has been downloaded from Kaggle, data has been downloaded from Kaggle .

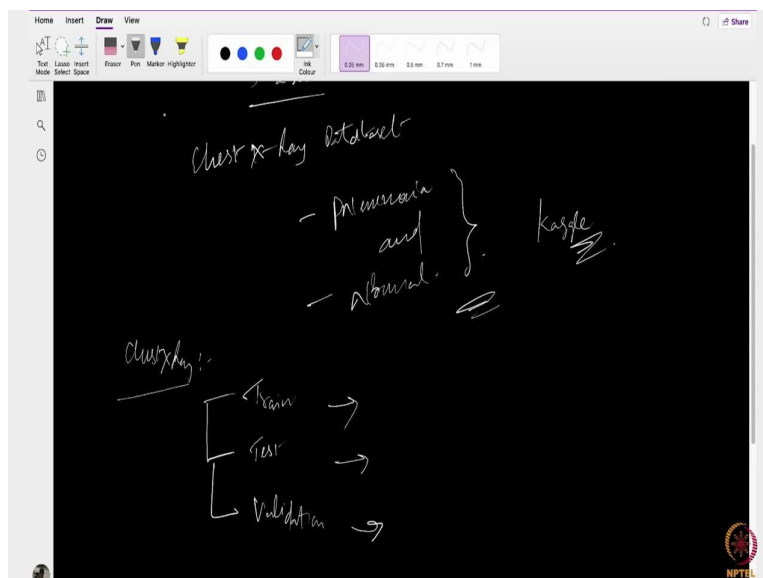
(Refer Slide Time: 22:19)

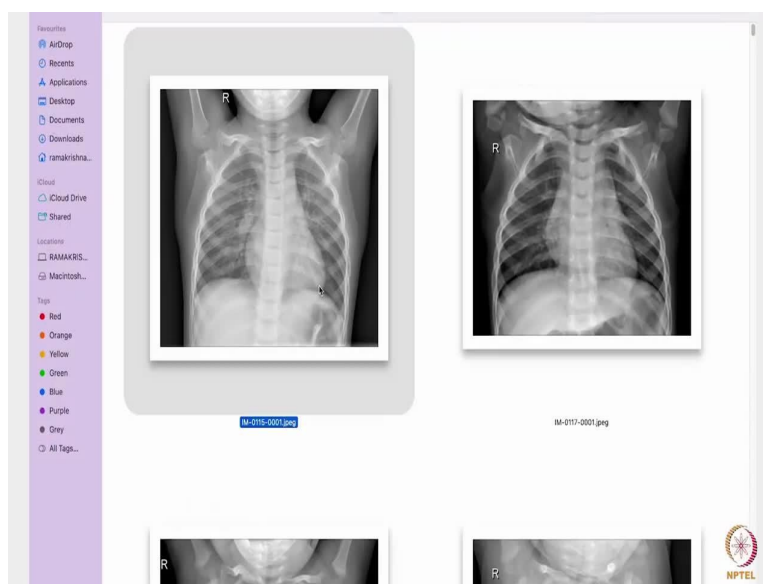
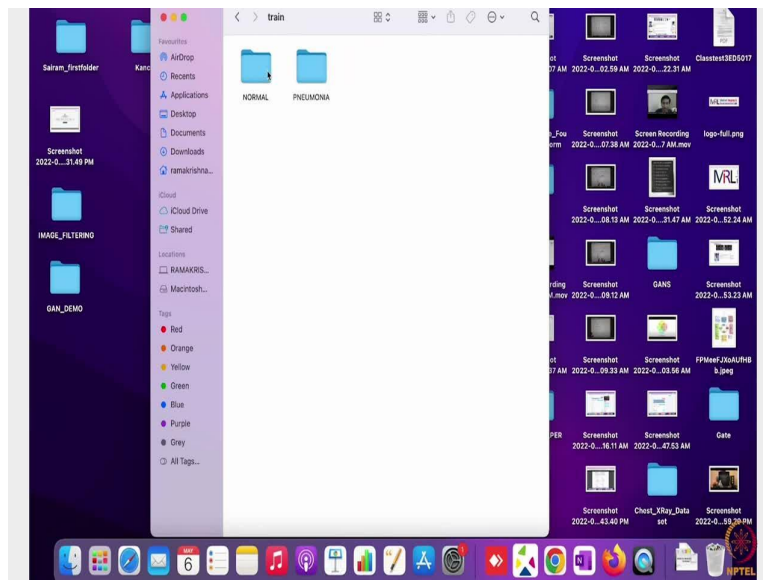
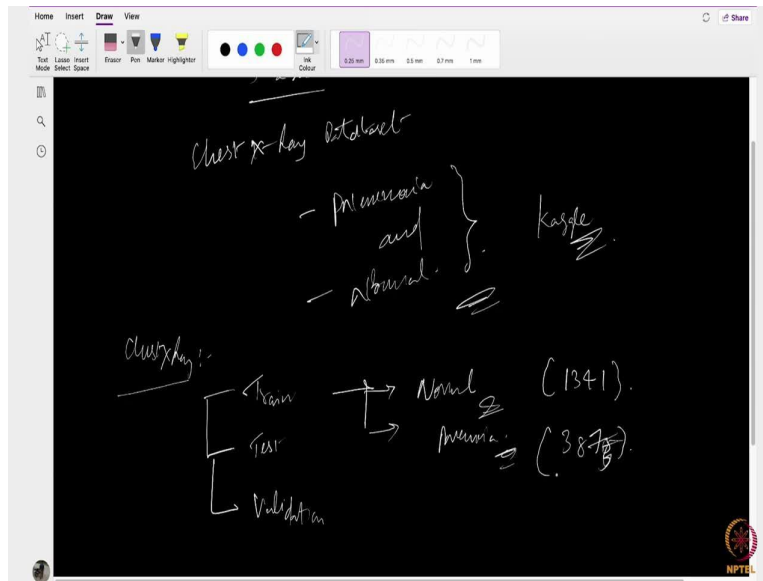


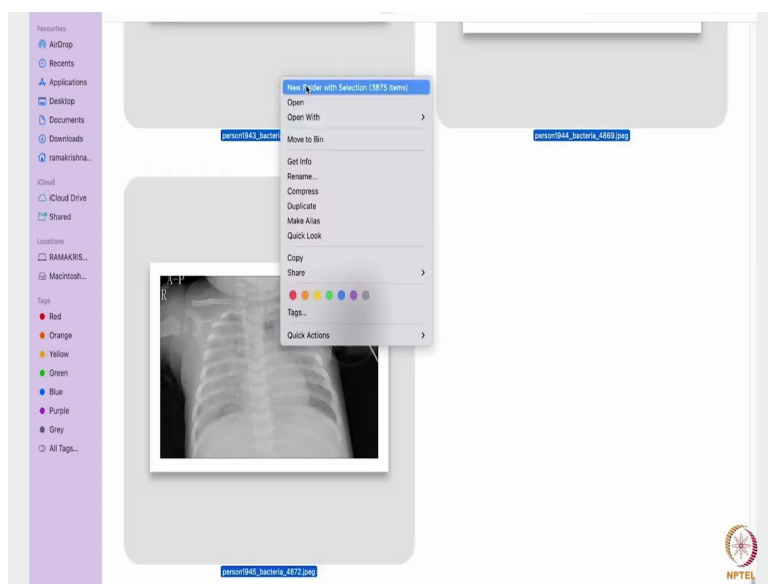
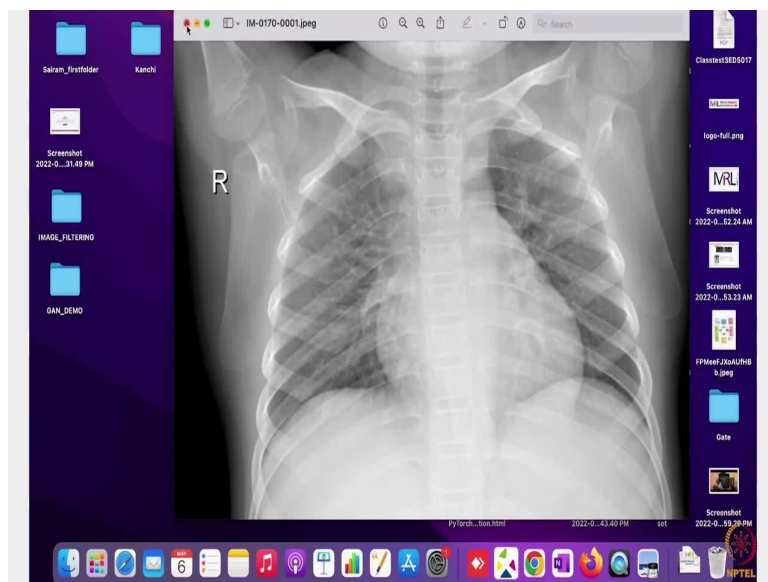
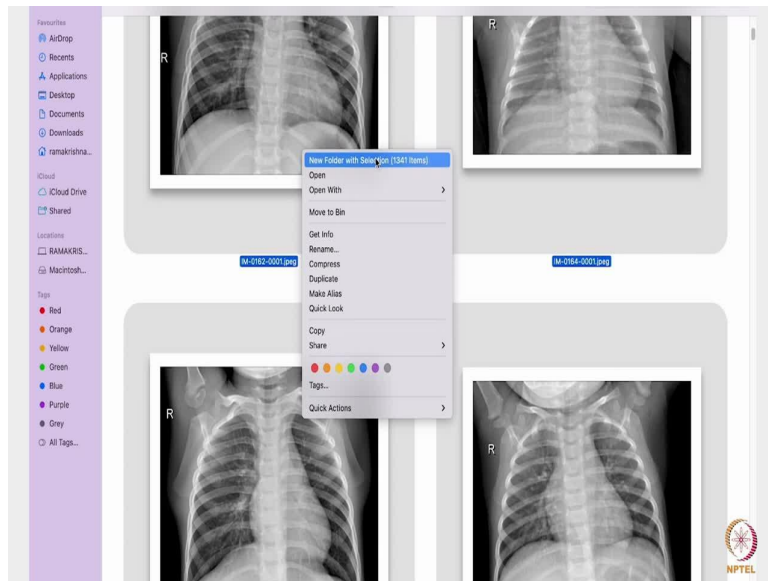


Now let me show you that Kaggle chest X ray images pneumonia just type this one in Kaggle you will get to know about this data there you can see that download 2GB of data, just click here that data gets downloaded to your local computer how it will be? I will show you once you download to your local computer see here, once you download the data from Kaggle in your particular directory wherever you have downloaded this file will be shown this folder that chest X ray and it contains 3 folders.

(Refer Slide Time: 23:00)





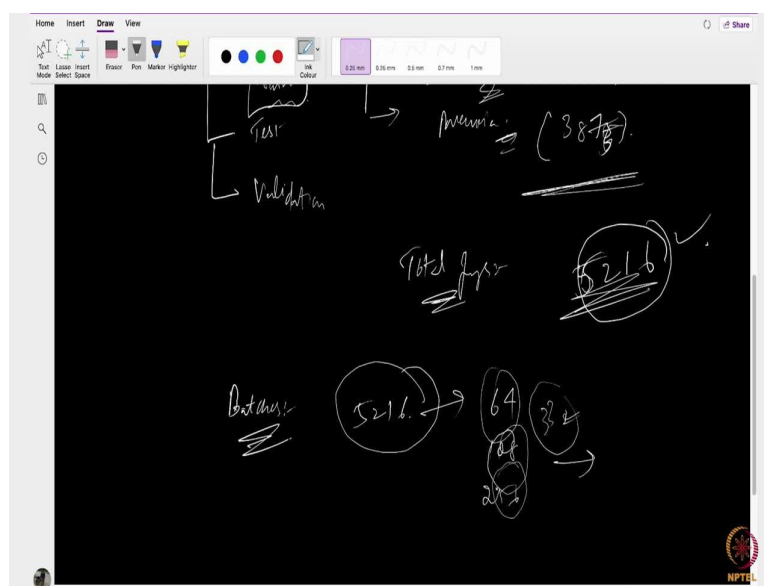


Train, test and validation, let us see what is there in each and every folder as a subdirectory? First we have chest X ray, chest X ray is the main directory and afterwards we have 3 folders now again subdirectory will be there. First one let us see, inside the train directory we have again normal and pneumonia, normal and pneumonia particular class of chest X rays.

Let us see how many files are there in training data even while implementing this algorithm also we mainly focus on this training data itself. Here we are not focusing on test data and validation. Let us see how many files are there instead the training folder, you can see normal and pneumonia

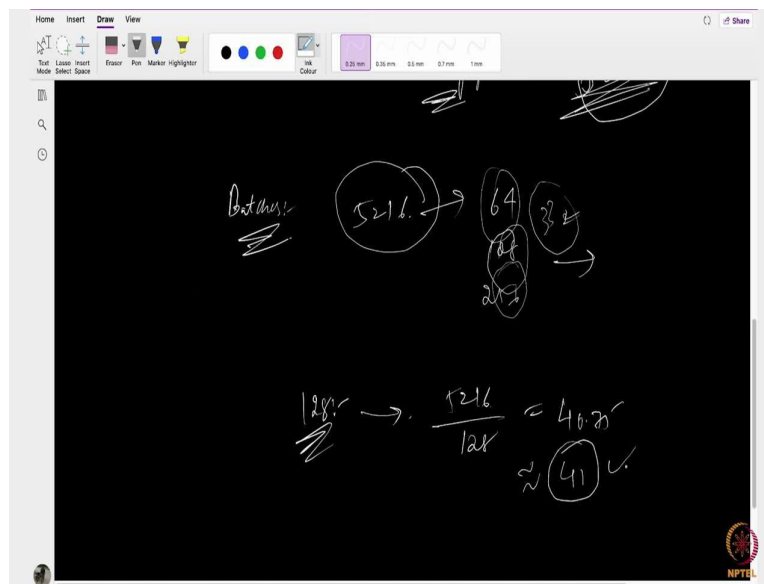
Let me open normal you can see, this is for the images we look like. And all these images are in dot jpeg format, all these images are in JPEG format. See here. Let me count how many images are there total 1341 items in normal folder. In normal folder 1341 chest X ray images similarly let me open one image and let me let us this is for the images look like. You can see clearly. Now in back the data looks like. Let us get back to pneumonia folder and set this pneumonia folder just count the total number of images 3875 the sum total of 3875.

(Refer Slide Time: 25:35)



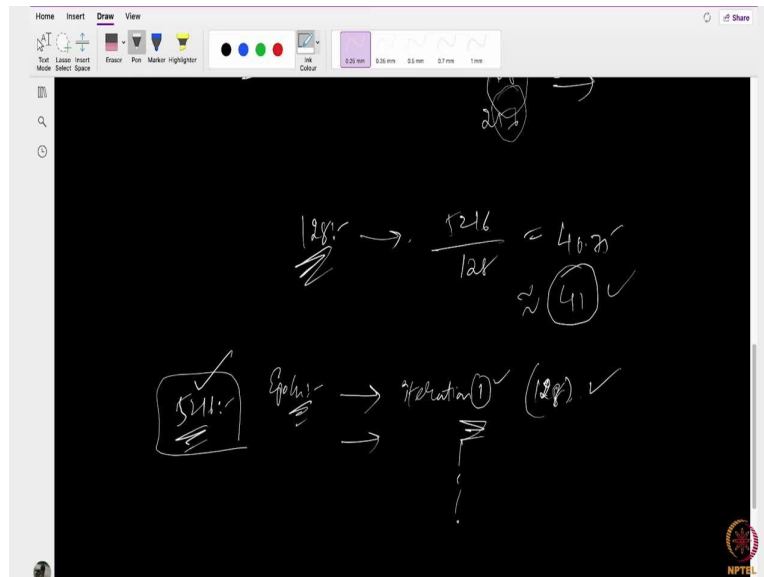
So, total images in the entire training data, training data folder is $(1341+3875)$ that means sum of these two numbers 5216. Just remember this one, once we go and use this data, we need this number because usually for deep learning algorithms, we will give the feed the data in terms of batches, out of this 5216 data, we will keep batch size say some 64, 128, 256 or 32 likewise, we will give several batch sizes.

(Refer Slide Time: 26:24)



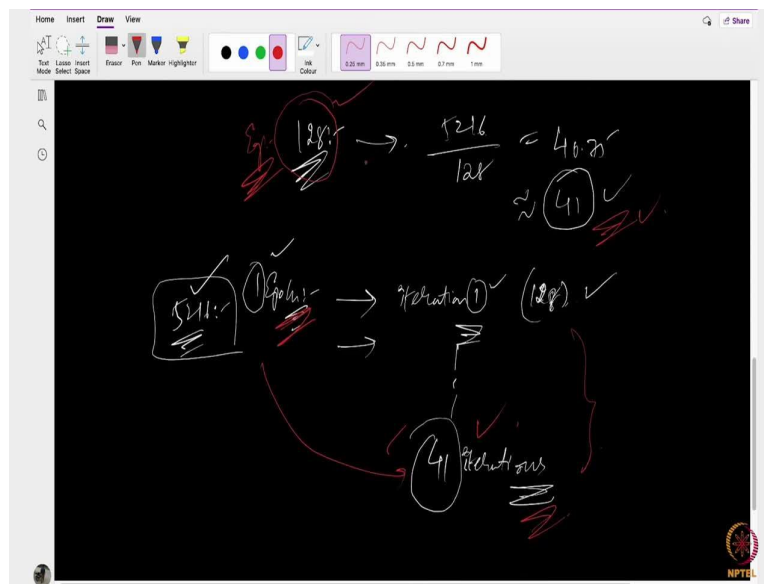
Suppose I take batch size as 128. Let us see, in a single epoch, how many iterations will be there. So, 5216 total images whole divided by 128. So, the answer will be 40.75, so, rounded up to 41.

(Refer Slide Time: 26:54)



That means, total images are 5216, that means one single epoch is, one single epoch means we have to go through entire all the batches of images. That means, on a single batch size, we will say one iteration, one iteration is nothing but one single batch size that means 128 out of these entire images 128 images will be taken and training will be done and we call that phases as one iteration.

(Refer Slide Time: 27:26)



Similar likewise we have total of 41 iterations. Once you take the training data, sorry, once you take the batch size as 128. Now, we can say that in a single epoch, in one epoch, we have 42 iterations that is it. This is what the discussion of enter dataset summarize just X ray data has been downloaded from Kaggle. And we have mainly 2 sets of diseases pneumonia, and I am sorry, 2 classes pneumonia normal.

Now, coming back to normal images, there are 1341 images and it comes to pneumonia, we have around 3875 so, in total 5216 number of images. So, feeding the data to depending model we just divide the data into several batches. So, for suppose, as an example, I have explained it here with respect to 128 as batch size. If you keep 128 batch size so, 5216 whole divided by 128 that means 41 batches in a single entire training data we can make 41 batches or in other way the deep learning training will be happen on each and every batch that means we have in total 41 iterations to go through the entire data set, entire data set or in other way we can say that in a single epoch we are going to have 41 iterations.

Where I explained this stuff means, explaining the while implementing this algorithm I have used batch size as around 128. This calculation will be helpful for you. This is the end of this data set discussion.

(Refer Slide Time: 29:10)

The screenshot shows the PyTorch DCGAN tutorial page. The left sidebar contains a search bar and a list of tutorials. The main content area is titled 'Inputs' and explains the parameters for running the DCGAN. It lists several inputs: **dataroot** (path to dataset folder), **workers** (number of worker threads), **batch_size** (batch size used in training), **image_size** (spatial size of images), **nc** (number of color channels), **nz** (length of latent vector), **ngf** (depth of feature maps in generator), **ndf** (depth of feature maps in discriminator), **num_epochs** (number of training epochs), **lr** (learning rate), **beta1** (hyperparameter for Adam optimizers), and **ngpu** (number of GPUs available). Below the list, there is a code block showing the configuration for the dataset and the number of workers.

```
# Root directory for dataset
dataroot = "data/celeba"

# Number of workers for dataloader
workers = 2
```

The screenshot shows the PyTorch DCGAN tutorial page. The left sidebar contains a search bar and a list of tutorials. The main content area is titled 'DCGAN TUTORIAL' and provides an introduction to DCGANs. It explains that the tutorial will give an introduction to DCGANs through an example, where a generative adversarial network (GAN) is trained to generate new celebrities by showing pictures of many real celebrities. It also mentions that the code is from the DCGAN implementation in PyTorch examples, and the document will give a thorough explanation of the implementation and shed light on how and why this model works. The section 'Generative Adversarial Networks' is also visible, along with a sub-section 'What is a GAN?'.

DCGAN TUTORIAL

Author: Nathan Inkawich Introduction

This tutorial will give an introduction to DCGANs through an example. We will train a generative adversarial network (GAN) to generate new celebrities after showing it pictures of many real celebrities. Most of the code here is from the DCGAN implementation in `pytorch/examples`, and this document will give a thorough explanation of the implementation and shed light on how and why this model works. But don't worry, no prior knowledge of GANs is required, but it may require a first-timer to spend some time reasoning about what is actually happening under the hood. Also, for the sake of time it will help to have a GPU, or two. Let's start from the beginning.

Generative Adversarial Networks

What is a GAN?

GANs are a framework for teaching a DL model to capture the training data's distribution so we can generate new data from that same distribution. GANs were invented by Ian Goodfellow in 2014 and first described in the paper *Generative Adversarial Nets*. They are made of two distinct models, a generator and a discriminator. The job of the generator is to spawn "fake" images that look like the training images. The job of the discriminator is to look at an image and output whether or not it is a real training image or a fake image from the generator. During training, the generator is constantly trying to outsmart the discriminator by generating better and better fakes, while the discriminator is working to become a better detective and correctly classify the real and fake images. The equilibrium of this game is when the generator is

Home Insert Draw View

Test Lasso Select Space Eraser Pin Marker Highlighter Ink Colour 0.25 mm 0.50 mm 0.8 mm 0.7 mm 1 mm

DCGAN Tutorial — PyTorch Tutorials 1.11.0+cu102 documentation

Friday, 6 May 2022 9:02 AM

DCGAN Tutorial — ...

06/05/2022 09:00 DCGAN Tutorial — PyTorch Tutorials 1.11.0+cu102 documentation

Table of Contents

Colab Notebook GitHub

DCGAN TUTORIAL

Author: Nathan Ingham's introduction

This tutorial will give an introduction to DCGANs through an example. We will train a generative adversarial network (GAN) to generate new celebrities after showing it pictures of many real celebrities. Most of the code here is from the `dcgan` implementation in `pytorch/examples`, and this document will give a thorough explanation of the implementation and shed light on how and why this model works. But don't worry, no prior knowledge of GANs is required, but it may require a first-timer to spend some time reasoning about what is actually happening under

NPTEL

Home Insert Draw View

Test Lasso Select Space Eraser Pin Marker Highlighter Ink Colour 0.25 mm 0.50 mm 0.8 mm 0.7 mm 1 mm

So, $D(G(x))$ is the probability (scalar) that the output of the generator G is a real image. As described in Goodfellow's paper, D and G play a minimax game in which D tries to maximize the probability it correctly classifies real and false ($\log D(x)$), and G tries to minimize the probability that D will predict its outputs are false ($\log(1 - D(G(x)))$). From the paper, the GAN loss function is

$$\min_G \max_D (D, G) = \mathbb{E}_{x \sim p_{\text{data}}(x)} [\log D(x)] + \mathbb{E}_{z \sim p_z(z)} [\log(1 - D(G(z)))]$$

In theory, the solution to this minimax game is where $p_z = p_{\text{data}}$, and the discriminator guesses randomly if the inputs are real or false. However, the convergence theory of GANs is still being actively researched and in reality models do not always train to this point.

What is a DCGAN?

A DCGAN is a direct extension of the GAN described above, except that it explicitly uses convolutional and convolutional-transpose layers in the discriminator and generator, respectively. It was first described by Radford et al. in the paper *Unsupervised Representation Learning With Deep Convolutional Generative Adversarial Networks*. The discriminator is made up of strided convolution layers, batch norm layers, and LeakyReLU activations. The input is a 3x64x64 input image and the output is a scalar probability that the input is from the real data distribution. The generator is comprised of convolutional-transpose layers, batch norm layers, and ReLU activations. The input is a latent vector, z , that is drawn from a standard normal distribution and the output is a 3x64x64 RGB image. The strided conv-transpose layers allow the latent vector to be transformed into a volume with the same shape as an image. In the paper, the authors also give some tips about how to setup the optimizers, how to calculate the loss functions, and how to initialize the model weights, all of which will be explained in the coming sections.

```
from __future__ import print_function
import argparse
import os
import random
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as dset
import torchvision.transforms as transforms
import torchvision.utils as vutils
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML

# Set random seed for reproducibility
```

NPTEL

Home Insert Draw View

Test Lasso Select Space Eraser Pin Marker Highlighter Ink Colour 0.25 mm 0.50 mm 0.8 mm 0.7 mm 1 mm

Saved offline (error) Share

being actively researched and in reality models do not always train to this point.

What is a DCGAN?

A DCGAN is a direct extension of the GAN described above, except that it explicitly uses convolutional and convolutional-transpose layers in the discriminator and generator, respectively. It was first described by Radford et al. in the paper *Unsupervised Representation Learning With Deep Convolutional Generative Adversarial Networks*. The discriminator is made up of strided convolution layers, batch norm layers, and LeakyReLU activations. The input is a 3x64x64 input image and the output is a scalar probability that the input is from the real data distribution. The generator is comprised of convolutional-transpose layers, batch norm layers, and ReLU activations. The input is a latent vector, z , that is drawn from a standard normal distribution and the output is a 3x64x64 RGB image. The strided conv-transpose layers allow the latent vector to be transformed into a volume with the same shape as an image. In the paper, the authors also give some tips about how to setup the optimizers, how to calculate the loss functions, and how to initialize the model weights, all of which will be explained in the coming sections.

```
from __future__ import print_function
import argparse
import os
import random
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as dset
import torchvision.transforms as transforms
import torchvision.utils as vutils
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML

# Set random seed for reproducibility
manualSeed = 999
manualSeed = random.randint(1, 20000) # use if you want new results
print("Random Seed: ", manualSeed)
random.seed(manualSeed)
torch.manual_seed(manualSeed)
```

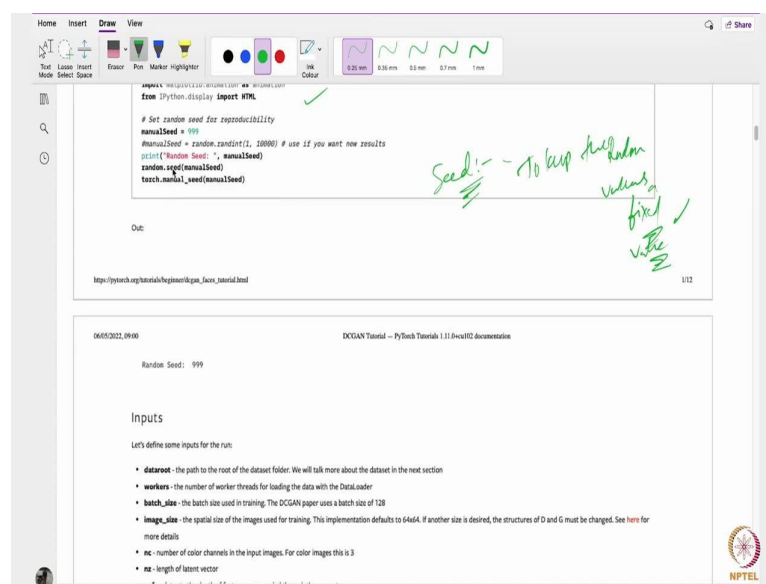
NPTEL

In this section, we will now see how to implement the code downloaded data set of chest X rays. So here if you go to PyTorch tutorials, here you can see DCGAN tutorial and you can directly go there and go through this, but let me explain you what is happening in each and every block. So that once it is done, we will directly go to our code, whatever I have will build here, see here, this is the entire code first relationship taken here.

See here DCGAN tutorial they just explain what exactly GANS and what we have already discussed in the first code min max algorithm now coming to coming back to DCGAN what they have done is first of all we have to here we are using PyTorch framework first of all PyTorch framework.

Next thing is they have imported all the module, requires modules, NumPy matplotlib apart from that random torch, here the deep learning library torch dot nn dot nn parallel all these modules are to be imported before going to write code.

(Refer Slide Time: 30:58)



Next thing is they have imported here manual seed and you know usually we use random values especially for initializing the weights or some other places to repeat the same values we always go for seed to keep the random value as a fixed value to keep the random value as fixed value throughout the programme. That means for a particular type of running, for a particular type of running the value should not change that is how we go our manual seeds similar in torch and even random seed.

(Refer Slide Time: 31:38)

Home Insert Draw View

Test Lasso Select Space Eraser Pen Marker Highlighter Ink Colour 0.25 mm 0.30 mm 0.5 mm 0.7 mm 1 mm

Inputs

Let's define some inputs for the run:

- dataset** - the path to the root of the dataset folder. We will talk more about the dataset in the next section
- workers** - the number of worker threads for loading the data with the Dataloader
- batch_size** - the batch size used in training. The DCGAN paper uses a batch size of 128
- image_size** - the spatial size of the images used for training. This implementation defaults to 64x64. If another size is desired, the structures of D and G must be changed. See [here](#) for more details
- nc** - number of color channels in the input images. For color images this is 3
- nz** - length of latent vector
- ngf** - relates to the depth of feature maps carried through the generator
- nlf** - sets the depth of feature maps propagated through the discriminator
- num_epochs** - number of training epochs to run. Training for longer will probably lead to better results but will also take much longer
- lr** - learning rate for training. As described in the DCGAN paper, this number should be 0.0002
- beta1** - beta1 hyperparameter for Adam optimizers. As described in paper, this number should be 0.5
- ngpu** - number of GPUs available. If this is 0, code will run in CPU mode. If this number is greater than 0 it will run on that number of GPUs

```
# Root directory for dataset
dataset = "data/cats"

# Number of workers for dataloader
workers = 2

# Batch size during training
batch_size = 128

# Spatial size of training images. All images will be resized to this
# size using a transformer.
image_size = 64

# Number of channels in the training images. For color images this is 3
nc = 3

# Size of z latent vector (i.e. size of generator input)
```

NPTEL

Features

- AirDrop
- Recents
- Applications
- Desktop
- Documents
- Downloads
- ramakrishna...

iCloud

- iCloud Drive
- Shared

Locations

- RAMAKRIS...
- Macintosh...

Tags

- Red
- Orange
- Yellow
- Green
- Blue
- Purple
- Grey
- All Tags...

MACOSX Data Set

NPTEL

Features

- AirDrop
- Recents
- Applications
- Desktop
- Documents
- Downloads
- ramakrishna...

iCloud

- iCloud Drive
- Shared

Locations

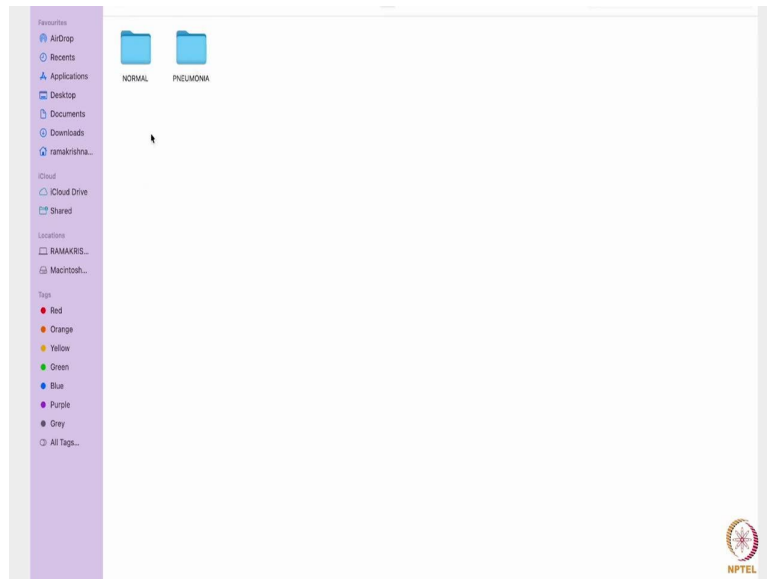
- RAMAKRIS...
- Macintosh...

Tags

- Red
- Orange
- Yellow
- Green
- Blue
- Purple
- Grey
- All Tags...

test train val

NPTEL



Now coming back to the inputs or hyper parameters whatever we are keeping here. Here you can see the dataroot, dataroot in the sense the root directory suppose here as I have shown you earlier here I am trying to use chest X rays this is the root chest X rays folder is the root folder inside this again I am going to track train data this is what my root folder, inside that root folder we have normal and pneumonia folders.

This is what we even for your may be for this kind of data set we are we have done like I suppose in case of your data set, you may have some root directory and you may have trained you should have chest X rays you might have some other data set inside this you will definitely have this division of test, train and validation.

Inside trained data you will have, according to a problem you will have N number of classes and I want you to keep this kind of directory structure, root directory main root directory, its sub directory and its subdirectories, this is how the files should be arranged folder should be arranged.

Next thing is workers. Next thing is batch size 128 they have taken 128. Later on you can change this one and you can play around with this one. So, we call this one as a hyper parameter.

(Refer Slide Time: 33:03)

• **dataset** - the path to the root of the dataset folder. We will talk more about the dataset in the next section
 • **workers** - the number of worker threads for loading the data with the DataLoader
 • **batch_size** - the batch size used in training. The DCGAN paper uses a batch size of 128
 • **image_size** - the spatial size of the images used for training. This implementation defaults to 64x64. If another size is desired, the structures of D and G must be changed. See [here](#) for more details
 • **nc** - number of color channels in the input images. For color images this is 3
 • **nz** - length of latent vector
 • **ngf** - relates to the depth of feature maps carried through the generator
 • **nfd** - sets the depth of feature maps propagated through the discriminator
 • **num_epochs** - number of training epochs to run. Training for longer will probably lead to better results but will also take much longer
 • **lr** - learning rate for training. As described in the DCGAN paper, this number should be 0.0002
 • **beta1** - beta1 hyperparameter for Adam optimizers. As described in paper, this number should be 0.5
 • **ngpu** - number of GPUs available. If this is 0, code will run in CPU mode. If this number is greater than 0 it will run on that number of GPUs

```

# Root directory for dataset
dataset = "data/calab"

# Number of workers for dataloader
workers = 2

# Batch size during training
batch_size = 128

# Spatial size of training images. All images will be resized to this
# size using a transformer.
image_size = 64

# Number of channels in the training images. For color images this is 3
nc = 3

# Size of z latent vector (i.e. size of generator input)
nz = 100

# Size of feature maps in generator
ngf = 64

# Size of feature maps in discriminator
nfd = 64

# Number of training epochs
num_epochs = 5
  
```

Handwritten notes: *image_size* is circled in green. *64x64* is written in green with an arrow pointing to the *image_size* line. *128* is written in green next to *batch_size*.

Next thing is image size here for the implementation sake they have used 64×64 I mean even you whatsoever data you are working on for the all the images in this particular data set has to be resized to 64×64 sample. Next number of color channels in the input image for color channel this is 3, number of channels of the particular data set of images you are having.

(Refer Slide Time: 33:33)

• **dataset** - the path to the root of the dataset folder. We will talk more about the dataset in the next section
 • **workers** - the number of worker threads for loading the data with the DataLoader
 • **batch_size** - the batch size used in training. The DCGAN paper uses a batch size of 128
 • **image_size** - the spatial size of the images used for training. This implementation defaults to 64x64. If another size is desired, the structures of D and G must be changed. See [here](#) for more details
 • **nc** - number of color channels in the input images. For color images this is 3
 • **nz** - length of latent vector
 • **ngf** - relates to the depth of feature maps carried through the generator
 • **nfd** - sets the depth of feature maps propagated through the discriminator
 • **num_epochs** - number of training epochs to run. Training for longer will probably lead to better results but will also take much longer
 • **lr** - learning rate for training. As described in the DCGAN paper, this number should be 0.0002
 • **beta1** - beta1 hyperparameter for Adam optimizers. As described in paper, this number should be 0.5
 • **ngpu** - number of GPUs available. If this is 0, code will run in CPU mode. If this number is greater than 0 it will run on that number of GPUs

```

# Root directory for dataset
dataset = "data/calab"

# Number of workers for dataloader
workers = 2

# Batch size during training
batch_size = 128

# Spatial size of training images. All images will be resized to this
# size using a transformer.
image_size = 64

# Number of channels in the training images. For color images this is 3
nc = 3

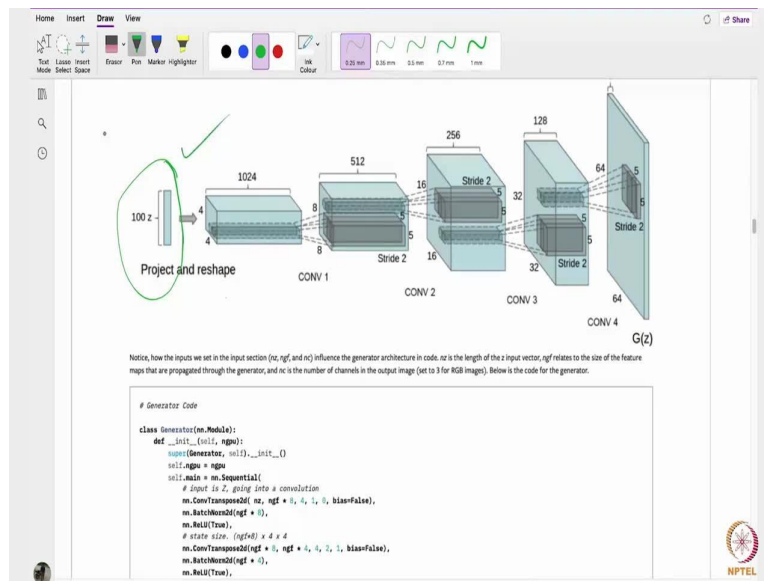
# Size of z latent vector (i.e. size of generator input)
nz = 100

# Size of feature maps in generator
ngf = 64

# Size of feature maps in discriminator
nfd = 64

# Number of training epochs
num_epochs = 5
  
```

Handwritten notes: *image_size* is circled in green. *64x64* is written in green with an arrow pointing to the *image_size* line. *128* is written in green next to *batch_size*.



```
# Root directory for dataset
dataset = "data/calab"

# Number of workers for dataloader
workers = 2

# Batch size during training
batch_size = 128

# Spatial size of training images. All images will be resized to this
# size using a transformer.
image_size = 64

# Number of channels in the training images. For color images this is 3
nc = 3

# Size of z latent vector (i.e. size of generator input)
nz = 100

# Size of feature maps in generator
ngf = 64

# Size of feature maps in discriminator
ndf = 64

# Number of training epochs
num_epochs = 5

# Learning rate for optimizers
lr = 0.0002

# Beta1 hyperparam for Adam optimizers
beta1 = 0.5

# Number of GPUs available. Use 0 for CPU mode.
ngpu = 1
```

Next thing is the nz that is length of the latent vector it means the random noise vector so, I have already told you let us go to in the case of generator it takes 100 dimensional input vector here this is called they are saying as latent vector. You can here also they had kept it as nz equal to 100 this is the number of channels of the input images.

Next thing is size of feature maps in generator and number of size of feature maps in discriminator. Here you can keep it according to your problem at your hand you can keep these values but here they had kept as 64 and 64 for both discriminator and generator I will explain you in generator section.

(Refer Slide Time: 34:16)

The generator, G_z , is designed to map the latent space vector z to data-space. Since our data are images, converting z to data-space means ultimately creating a RGB image with the same size as the training images (e.g. 64x64x3). In practice, this is accomplished through a series of strided two dimensional convolutional transpose layers, each paired with a 2d batch norm layer and a relu activation. The output of the generator is fed through a tanh function to return it to the input data range of $[-1, 1]$. It is worth noting the existence of the batch norm functions after the conv-transpose layers, as this is a critical contribution of the DCGAN paper. These layers help with the flow of gradients during training. An image of the generator from the DCGAN paper is shown below.

Notice, how the inputs we set in the input section (see ngf and nz) influence the generator architecture in code: nz is the length of the z input vector, ngf relates to the size of the feature maps that are propagated through the generator, and nc is the number of channels in the output image (set to 3 for RGB images). Below is the code for the generator.

```
# Generator Code

class Generator(nn.Module):
    def __init__(self, ngf):
        super(Generator, self).__init__()
        self.ngp = ngf
        self.model = nn.Sequential(
            nn.ConvTranspose2d(100, 1024, kernel_size=4, stride=1, padding=1),
            nn.BatchNorm2d(1024),
            nn.ReLU(True),
            nn.ConvTranspose2d(1024, 512, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(512),
            nn.ReLU(True),
            nn.ConvTranspose2d(512, 256, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(True),
            nn.ConvTranspose2d(256, 128, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(True),
            nn.ConvTranspose2d(128, 64, kernel_size=4, stride=2, padding=1),
            nn.BatchNorm2d(64),
            nn.ReLU(True),
            nn.ConvTranspose2d(64, 3, kernel_size=4, stride=2, padding=1),
            nn.Tanh()
        )

    def forward(self, z):
        z = z.view(-1, 100)
        z = self.model(z)
        return z.view(-1, 3, 64, 64)
```

So that you will get some idea let me continue here the random noise input vector is 100 size here if you observe this feature maps values, we can write this one as 64×16 . For this one we can write 64×8 , this one we can write 64×4 that is right. And again this one can be written as 64×2 that means this value has been fixed.

This is what I am talking about it as ngf value. The same thing will be working out for discriminator session also. Please keep it the point in mind. Maybe this value can be changed according to your according to the problem at your hand, but in this case, we are using this one as 64 only. More or less you can give the same values only.

(Refer Slide Time: 35:13)

• **ndf** - sets the depth of feature maps propagated through the discriminator
 • **num_epochs** - number of training epochs to run. Training for longer will probably lead to better results but will also take much longer
 • **lr** - learning rate for training. As described in the DCGAN paper, this number should be 0.0002
 • **beta1** (beta1) hyperparameter for Adam optimizers. As described in paper, this number should be 0.5
 • **ngpu** - number of GPUs available. If this is 0, code will run in CPU mode. If this number is greater than 0 it will run on that number of GPUs

```
# Root directory for dataset
dataset_root = "/data/datasets"

# Number of workers for dataloader
workers = 2

# Batch size during training
batch_size = 128

# Spatial size of training images. All images will be resized to this
# size using a transformer.
image_size = 64

# Number of channels in the training images. For color images this is 3
nc = 3

# Size of  $z$  latent vector (i.e. size of generator input)
nz = 100

# Size of feature maps in generator
ngf = 64

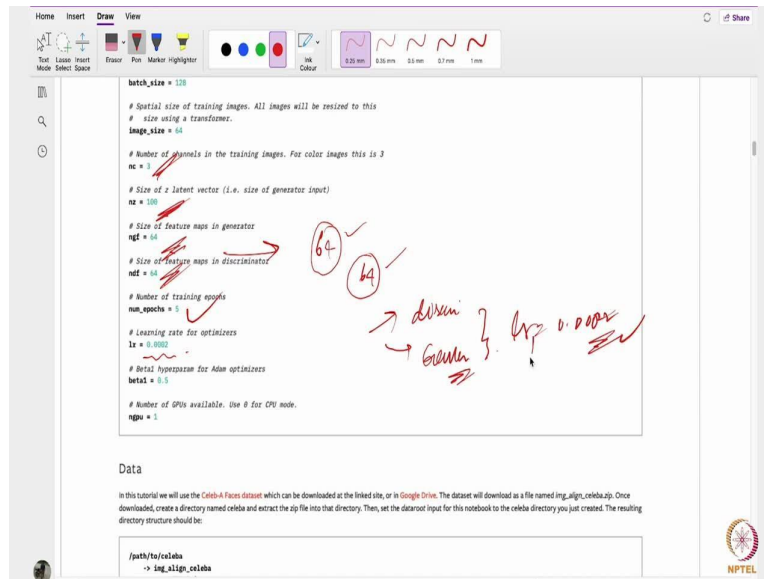
# Size of feature maps in discriminator
ndf = 64

# Number of training epochs
num_epochs = 5

# Learning rate for optimizers
lr = 0.0002

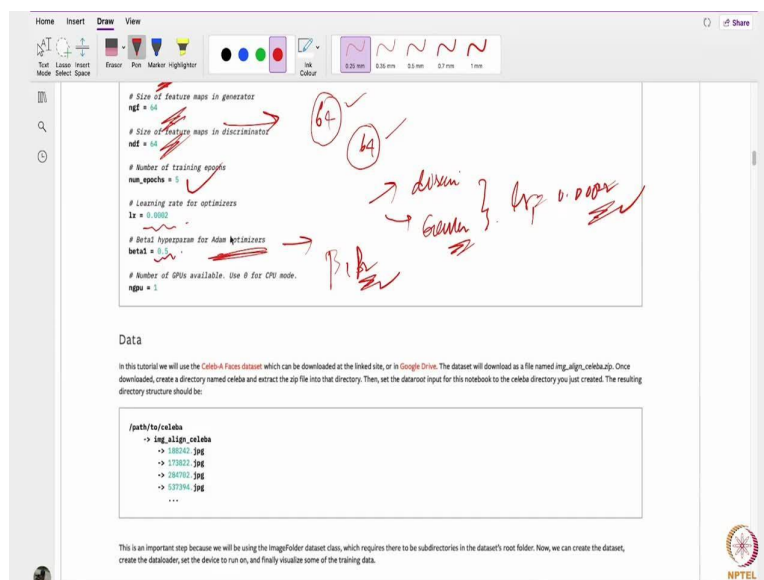
# Beta1 hyperparameter for Adam optimizers
beta1 = 0.5
```

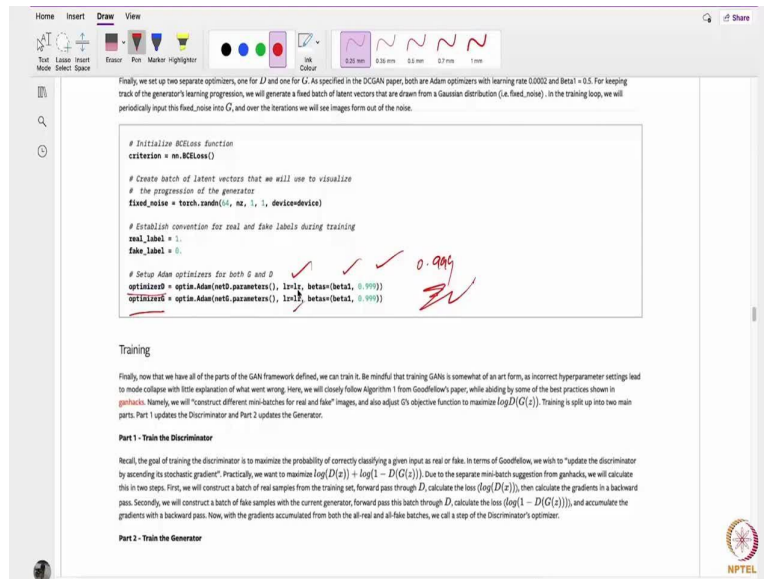
Handwritten red annotations highlight the values 64 for ngf and ndf, and 100 for nz, with arrows pointing to the corresponding code lines.



Now coming back to `ndf`, similarly number of epochs it depends on how many number of epochs you want to go through the entire training process. Similarly learning rate if you observe here number of epochs they have taken as 5 value learning rate has been fixed for both different discriminator we can have 1 learning rate and for generator we can have another learning it, but here they have kept both the values as `lr = 0.002`, so remember 02. Later on they you can play around with them, so, that you can bring some modifications, modifications in terms of improvement.

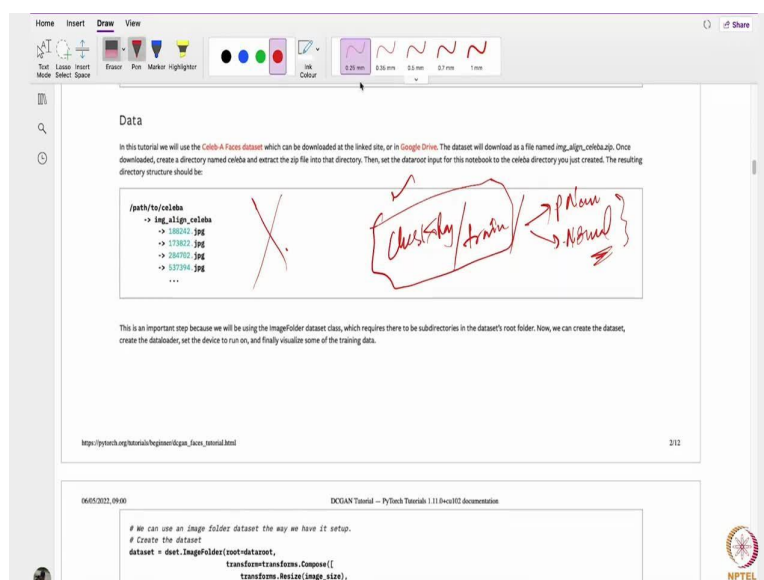
(Refer Slide Time: 35:51)





At the end another thing is they have used the optimizer as rm optimizer, optimizer has two parameters β_1 and β_2 , β_1 has been fixed to 0.5 and β_2 let us see where and see here the optimizer bm optimizer generator for optimizer generator optimizer and discriminator optimizer they have usually the parameters same learning rate here β_1 is 0.5 and the other β_2 is 0.999 these values are fixed you can change a lr value here or later on when you play around with the hyper parameters you can bring a lot of changes here. These are the discussion of hyper parameters.

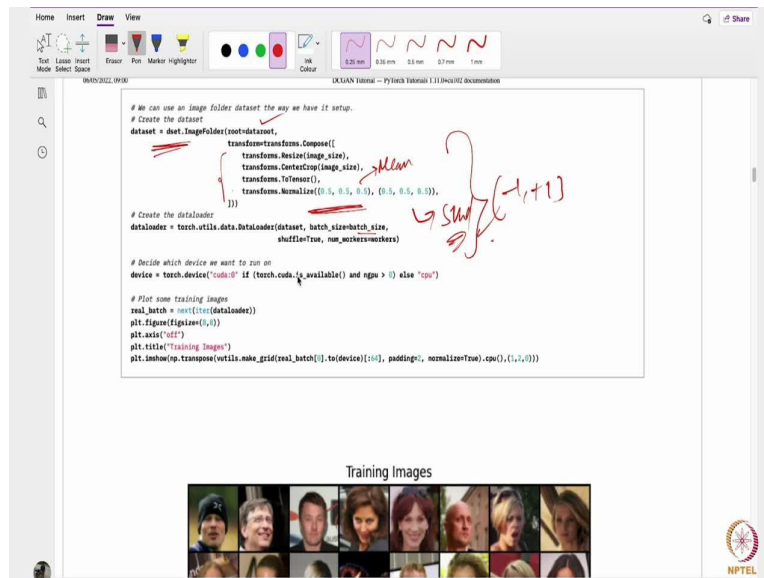
(Refer Slide Time: 36:40)



Here they have some other data set in the tutorial session but in our case we are going to use chest X ray data set. In chest X ray data set we have train folder inside the train folder we have 2 sub directories so, our root directory will be this one, here number of classes, here

pneumonia and another thing is normal, nothing else. Now coming back to, once you keep these hyper parameters and data here data set organization.

(Refer Slide Time: 37:20)



Now coming back to data set and data loader these are the very important concepts here you can see the data root directory will be given these are like transforms I mean you can resize the image you can center crop the image you may have to convert to the usually the entire deep learning will be worked on tensors not a Numpy arrays.

Next thing is we have to normalize the values to minus 1 comma plus 1 these things will first one indicates mean values for each and every channel similarly this indicates standard deviation of each and every channel. Please keep those words in your mind this is more or less like bringing augmentation inside this one.

After creating this data set, we have to go for data loader where we have, we will provide a batch size the entire data set has been given here. So that 128 images, 128 images will be fixed out and will be use it for training. Now please keep this suppose in case you have a local GPU in your at your home or you have local GPU so that you can use this one is command otherwise even in Google Colab also you can use that one.

(Refer Slide Time: 38:36)

The screenshot shows a Jupyter Notebook interface with the following code and annotations:

```

# We can use an ImageFolder dataset the way we have it setup.
# Create the dataset
dataset = datasets.ImageFolder(root=dataset_dir,
                              transform=transforms.Compose([
                                  transforms.Resize(image_size),
                                  transforms.CenterCrop(image_size),
                                  transforms.ToTensor(),
                                  transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])]),
                              loader=torch.utils.data._utils.PickleLoader())

# Create the dataloader
dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                                         shuffle=True, num_workers=workers)

# Decide which device we want to run on
device = torch.device("cuda:0" if (torch.cuda.is_available() and n_gpu > 0) else "cpu")

# Plot some training images
real_batch = next(iter(dataloader))
plt.figure(figsize=(8,8))
plt.axis('off')
plt.title("Training Images")
plt.imshow(np.transpose(utils.make_grid(real_batch[0].to(device)[0:64], padding=5, normalize=True).cpu(), (1,2,3)))
  
```

Handwritten red annotations include:

- A bracket on the `transforms.Compose` list with the note "Mean" and the formula $\frac{1}{1+1}$.
- A circle around the number `64` in the `make_grid` function call, with the note $\frac{1}{128}$.

Below the code, a small grid of 16 training images is displayed, labeled "Training Images".

This screenshot shows the same Jupyter Notebook interface, but the grid of training images is now larger, displaying 64 images arranged in an 8x8 grid. A red checkmark is visible to the right of the grid.

Here they have out of say 128 images as a batch. Let us see 1 2 3 4 5 6 7 8, 1 2 3 4 5 6 7 8, 8 \times 8 64. Here they are bringing out some total number of a single here see real batch of 0 dot to the 64. They are attracting 64 images out of 128 from a particular batch and they have shown here using matplotlib.

(Refer Slide Time: 39:10)

Implementation

With our input parameters set and the dataset prepared, we can now get into the implementation. We will start with the weight initialization strategy, then talk about the generator, discriminator, loss functions, and training loop in detail.

Weight Initialization

https://ytsearch.org/tutorials/PyTorch/DCGAN_tutorial.html

06/05/2022, 09:00 DCGAN Tutorial — PyTorch Tutorial 1.11.0-ws102-documentation

From the DCGAN paper, the authors specify that all model weights shall be randomly initialized from a Normal distribution with mean=0, stdev=0.02. The `weights_init` function takes an initialized model as input and reinitializes all convolutional, convolutional-transpose, and batch normalization layers to meet this criteria. This function is applied to the models immediately after initialization.

```
# custom weights initialization called on netG and netD
def weights_init(m):
    classname = m.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(m.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(m.weight.data, 1.0, 0.02)
        nn.init.constant_(m.bias.data, 0)
```

Generator

The generator, G , is designed to map the latent space vector (z) to data-space. Since our data are images, converting z to data-space means ultimately creating a RGB image with the same size as the training images (e.g. 364x364). In practice, this is accomplished through a series of strided two dimensional convolutional transpose layers, each paired with a 2d batch norm layer and a relu activation. The output of the generator is fed through a tanh function to return it to the input data range of $[-1, 1]$. It is worth noting the existence of the batch norm functions after the conv-transpose layers, as this is a critical contribution of the DCGAN paper. These layers help with the flow of gradients during training. An image of the generator from the DCGAN paper is shown below.

Next thing is weight initialization during the earlier one I have told you we have to initialize the weights.

(Refer Slide Time: 39:19)

Project and reshape

CONV 1 CONV 2 CONV 3 64 CONV 4 $G(z)$

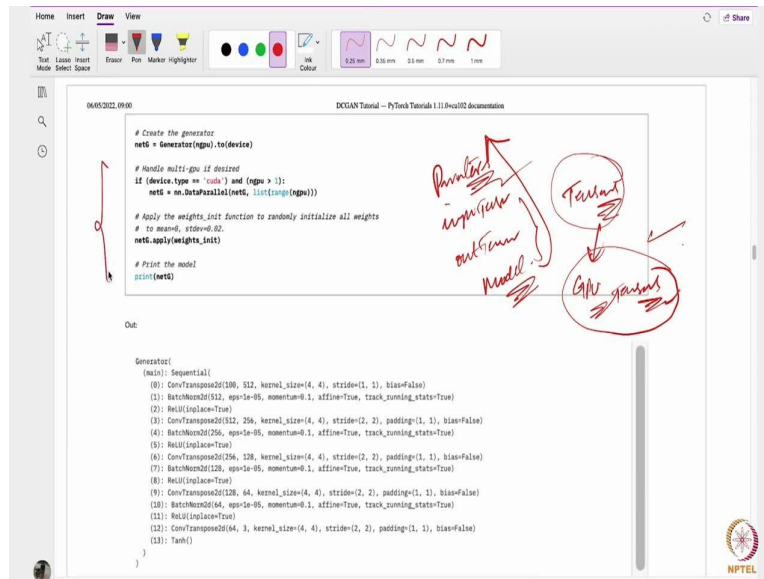
Notice, how the inputs we set in the input section (for `ngf` and `nc`) influence the generator architecture in code: `nc` is the length of the z input vector, `ngf` relates to the size of the feature maps that are propagated through the generator, and `nc` is the number of channels in the output image (set to 3 for RGB images). Below is the code for the generator.

```
# Generator Code
class Generator(nn.Module):
    def __init__(self, ngf):
        super(Generator, self).__init__()
        self.ngf = ngf
        self.model = nn.Sequential(
            # input is z, going into a convolution
            nn.ConvTranspose2d(nc, ngf * 4, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # state size: (ngf*4) x 4 x 4
            nn.ConvTranspose2d(ngf * 4, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # state size: (ngf*4) x 8 x 8
            nn.ConvTranspose2d(ngf * 4, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # state size: (ngf*4) x 16 x 16
            nn.ConvTranspose2d(ngf * 4, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # state size: (ngf) x 32 x 32
            nn.ConvTranspose2d(ngf, nc, 4, 2, 1, bias=False),
            nn.Tanh()
        )
        # state size: (nc) x 64 x 64

    def forward(self, input):
        return self.model(input)
```

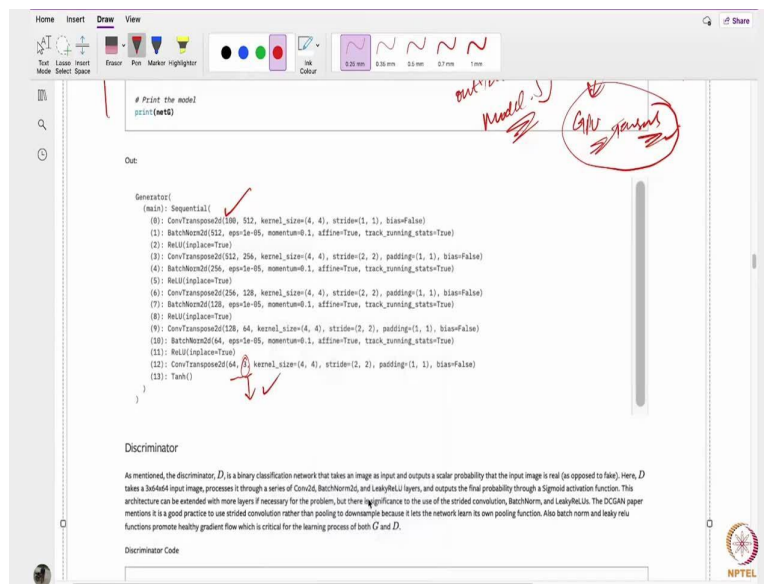
Next thing is generator once we go through here in generator section they use only transpose convolutions and now batch normalization and ReLU activation function at the end only they are used only tan(h) function I have already discussed in the theoretical part. Same thing has been implemented here. Batch normalization is present here everywhere this will follow the same diagram what they have shown here. You will get to know it is not very difficult.

(Refer Slide Time: 39:49)



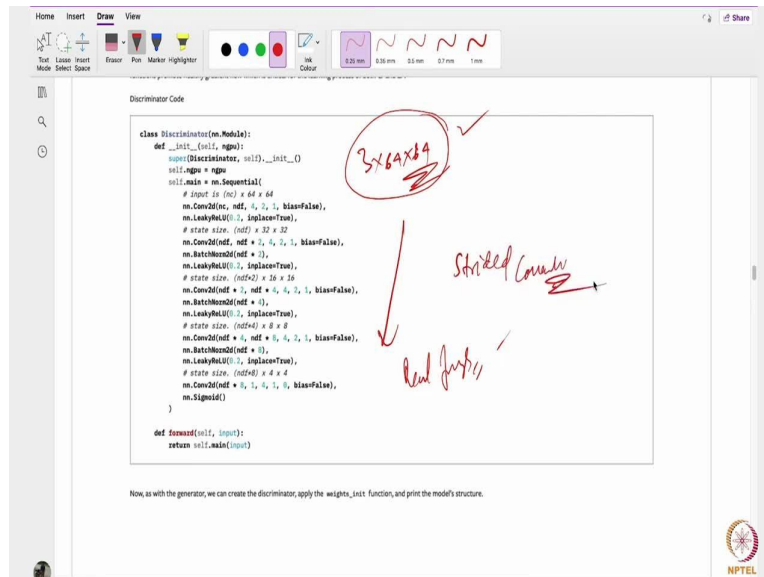
Next thing is implementation of generator block. Here in deep learning all the parameters all the input tensors or the output tensors or even the model means these parameters will come into play. Everything should be in the format of tensors only. Suppose if you are using GPU all these tensor has to be load to GPU tensors This is a fundamental point whatsoever the data you have every all those input images or whatsoever the data irrespective of the kind of data you attached images, images any other everything should be converted into tensors. Suppose if you are using GPU those tensors, tensors should be converted with GPU tensors, this is what they are doing here.

(Refer Slide Time: 40:41)



Now, you can see 100 dimensional random noise vector has been kept as an input 512 feature maps output (512,256) , (256,128) , (128,64) at the end, you will have only 3 feature maps containing output, that is not required here.

(Refer Slide Time: 40:57)



Next thing is discriminator same thing here you will feed the input images 3 64x64. And the output will be probability whether this particular image belongs to that what probability score is particular image belongs to real image that is output of this discriminator same thing here as we have followed leaky ReLU activation functions, batch normalization and strided convolution, strided convolution. Now, this is a discriminator block earlier we have discussed

about generator block, before that we have just to prepare the data set and convert it into data loader in terms of batches.

(Refer Slide Time: 41:40)

Loss Functions and Optimizers

With D and G setup, we can specify how they learn through the loss functions and optimizers. We will use the Binary Cross Entropy loss (`BCELoss`) function which is defined in PyTorch as

$$\ell(x, y) = L = \{l_1, \dots, l_n\}^T, \quad l_i = -[y_i \cdot \log x_i + (1 - y_i) \cdot \log(1 - x_i)]$$

Notice how this function provides the calculation of both log components in the objective function (i.e. $\log(D(x))$ and $\log(1 - D(G(z)))$). We can specify what part of the BCE equation to use with the y input. This is accomplished in the training loop which is coming up soon, but it is important to understand how we can choose which component we wish to calculate just by changing y (i.e. GT labels).

Next, we define our real label as 1 and the fake label as 0. These labels will be used when calculating the losses of D and G , and this is also the convention used in the original GAN paper. Finally, we set up two separate optimizers, one for D and one for G . As specified in the DCGAN paper, both are Adam optimizers with learning rate 0.0002 and $\beta_1 = 0.5$. For keeping track of the generator's learning progression, we will generate a fixed batch of latent vectors that are drawn from a Gaussian distribution (i.e. `fixed_noise`). In the training loop, we will periodically input this `fixed_noise` into G , and over the iterations we will see images form out of the noise.

```
# Initialize BCELoss function
criterion = nn.BCELoss()

# Create batch of latent vectors that we will use to visualize
# the progression of the generator
fixed_noise = torch.randn(16, 1, 1, device=device)

# Establish convention for real and fake labels during training
real_label = 1
fake_label = 0

# Setup Adam optimizers for both G and D
optimizerD = optim.Adam(netD.parameters(), lr=lr, betas=(betas, 0.999))
optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(betas, 0.999))
```

Training

Finally, now that we have all of the parts of the GAN framework defined, we can train it. Be mindful that training GANs is somewhat of an art form, as incorrect hyperparameter settings lead to mode collapse with little explanation of what went wrong. Here, we will closely follow Algorithm 1 from Goodfellow's paper, while abiding by some of the best practices shown in genutils. Namely, we will "construct different mini-batches for real and fake" images, and also adjust G's objective function to maximize $\log(D(G(z)))$. Training is split up into two main parts. Part 1 updates the Discriminator and Part 2 updates the Generator.

Part 1 - Train the Discriminator

Recall, the goal of training the discriminator is to maximize the probability of correctly classifying a given input as real or fake. In terms of Goodfellow, we wish to "update the discriminator by ascending its stochastic gradient". Practically, we want to maximize $\log(D(x)) + \log(1 - D(G(z)))$. Due to the separate mini batch suggestion from genutils, we will calculate this in two steps. First, we will construct a batch of real samples from the training set, forward pass through D , calculate the loss ($\log(D(x))$), then calculate the gradients in a backward pass. Secondly, we will construct a batch of fake samples with the current generator, forward pass this batch through D , calculate the loss ($\log(1 - D(G(z)))$), and accumulate the gradients with a backward pass. Now, with the gradients accumulated from both the all-real and all-fake batches, we call a step of the Discriminator's optimizer.

Part 2 - Train the Generator

Now, coming back to loss functions as I have already mentioned, we are going to use binary cross entropy loss only this loss function.

(Refer Slide Time: 41:47)

periodically input this `fixed_noise` into G , and over the iterations we will see images form out of the noise.

```
# Initialize BCELoss function
criterion = nn.BCELoss()

# Create batch of latent vectors that we will use to visualize
# the progression of the generator
fixed_noise = torch.randn(16, 1, 1, device=device)

# Establish convention for real and fake labels during training
real_label = 1
fake_label = 0

# Setup Adam optimizers for both G and D
optimizerD = optim.Adam(netD.parameters(), lr=lr, betas=(betas, 0.999))
optimizerG = optim.Adam(netG.parameters(), lr=lr, betas=(betas, 0.999))
```

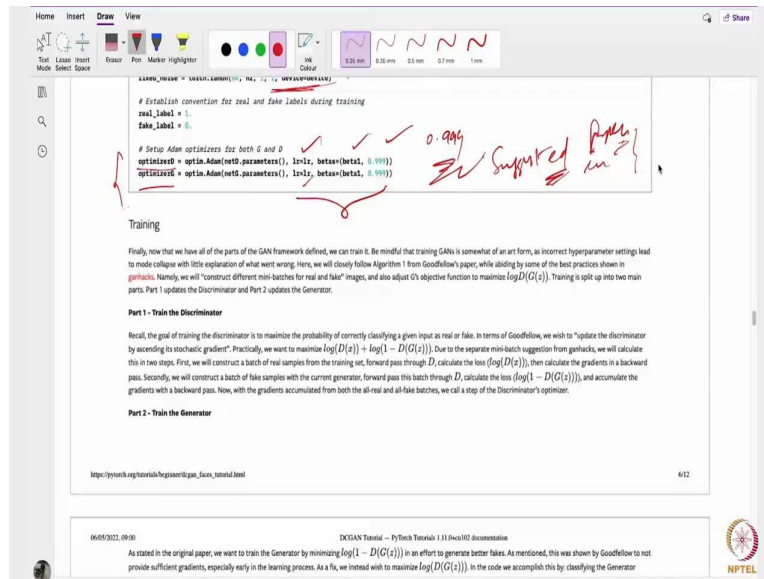
Training

Finally, now that we have all of the parts of the GAN framework defined, we can train it. Be mindful that training GANs is somewhat of an art form, as incorrect hyperparameter settings lead to mode collapse with little explanation of what went wrong. Here, we will closely follow Algorithm 1 from Goodfellow's paper, while abiding by some of the best practices shown in genutils. Namely, we will "construct different mini-batches for real and fake" images, and also adjust G's objective function to maximize $\log(D(G(z)))$. Training is split up into two main parts. Part 1 updates the Discriminator and Part 2 updates the Generator.

Part 1 - Train the Discriminator

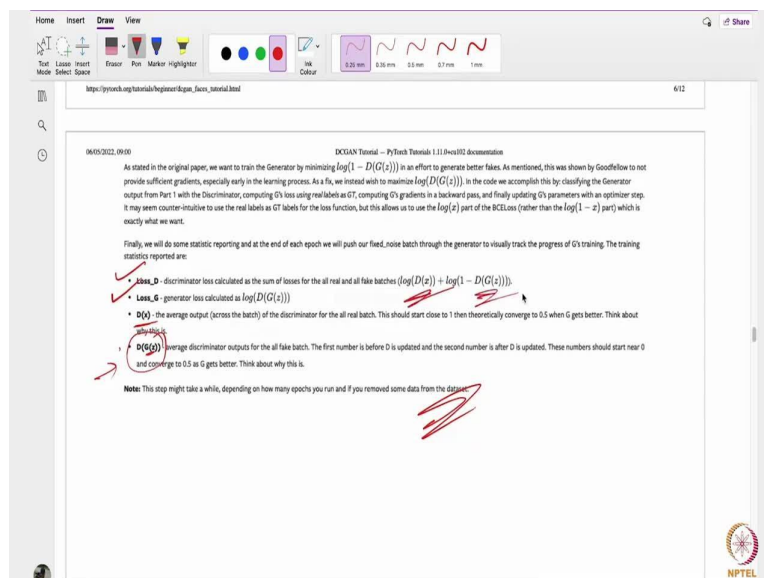
Recall, the goal of training the discriminator is to maximize the probability of correctly classifying a given input as real or fake. In terms of Goodfellow, we wish to "update the discriminator by ascending its stochastic gradient". Practically, we want to maximize $\log(D(x)) + \log(1 - D(G(z)))$. Due to the separate mini batch suggestion from genutils, we will calculate this in two steps. First, we will construct a batch of real samples from the training set, forward pass through D , calculate the loss ($\log(D(x))$), then calculate the gradients in a backward pass. Secondly, we will construct a batch of fake samples with the current generator, forward pass this batch through D , calculate the loss ($\log(1 - D(G(z)))$), and accumulate the gradients with a backward pass. Now, with the gradients accumulated from both the all-real and all-fake batches, we call a step of the Discriminator's optimizer.

Part 2 - Train the Generator



Here they have used the fixed noise this is for the sake of seeing how generator is generating the images real label has given as 1 and fake label has given as 0. Here you can see 2 optimizers will be given individually for generator and discriminator, you can bring a lot of changes here as you can play around with them hyper parameters, this β_1 value and a β_2 values are suggested in the paper actually we have taken the same values suggested in paper they use the same values.

(Refer Slide Time: 42:21)



Now, coming back to training in the training section you can see train the generator, train the discriminator and compute the loss obtained in discriminator loss obtained in generator. $D(x)$ indicates discriminator output then we feed input sorry real image from the data set whereas $G(z)$ indicates output if generated by the generator and it has been fed into discriminator. So,

$D(G(z))$ indicates the probability score that is output given by the discriminator when we feed the fake image generated image generated by the generator this is what is exactly these formulas indicate the terminology.

(Refer Slide Time: 43:13)

def train_D(real_data_loader, real_labels, device):
 # Format batch
 real_gpu = data[0].to(device)
 y_size = real_gpu.size(0)
 label = torch.full((y_size,), real_label, dtype=torch.float, device=device)
 # Forward pass real batch through D
 output = netD(real_gpu).view(-1)
 # Calculate loss on all-real batch
 errD_real = criterion(output, label)
 # Calculate gradients for D in backward pass
 errD_real.backward()
 D_x = output.mean().item()

 # Train with all-fake batch
 # Generate batch of latent vectors
 noise = torch.randn(y_size, nz, 1, 1, device=device)
 # Generate fake image batch with G
 fake = netG(noise)
 label.fill_(-1)
 # Classify all fake batch with D
 output = netD(fake.detach()).view(-1)
 # Calculate D's loss on the all-fake batch
 errD_fake = criterion(output, label)
 # Calculate the gradients for this batch, accumulated (summed) with previous gradients
 errD_fake.backward()
 D_y_1 = output.mean().item()
 # Compute error of D as sum over the fake and the real batches
 errD = errD_real + errD_fake
 # Update D
 optimizerD.step()

 # (2) Update G network: maximize log(D(G(z)))
 errG = 0
 label.fill_(real_label) # fake labels are real for generator cost
 # Since we just updated D, perform another forward pass of all-fake batch through D
 output = netD(fake).view(-1)
 # Calculate G's loss based on this output
 errG = criterion(output, label)
 # Calculate gradients for G
 errG.backward()
 G_x = output.mean().item()

 # Output training status
 if i % 100 == 0:
 print(f'[Epoch {epoch+1}] Loss_D: {errD:.4f} Loss_G: {errG:.4f} D(G(z)): {D_y_1:.4f} / {D_x:.4f}'
 f' G(G(z)): {G_x:.4f} / {G_y_1:.4f}')
 # Save losses for plotting later
 l_losses.append(errD.item())
 g_losses.append(errG.item())

 # Check how the generator is doing by saving G's output on fixed noise
 if (i % 1000 == 0) or (epoch == num_epochs-1) and (i % 1000 != 0):
 with torch.no_grad():
 fake = netG(fixed_noise).detach().cpu()
 img_list.append(utils.make_grid(fake, padding=2, normalize=True))

 i += 1
"

Next thing let us move on this is entire, this one is like training loop this is entire training loop here with respect to generator and this loop has been written with respect to discriminator. I have already told you in discriminator section the real images and the target values will be kept 1 whereas if you give fake images, the output target value will be kept 0 and the loss functions were calculated accordingly.

(Refer Slide Time: 43:38)

 errD_fake = criterion(output, label)
 # Calculate the gradients for this batch, accumulated (summed) with previous gradients
 errD_fake.backward()
 D_y_1 = output.mean().item()
 # Compute error of D as sum over the fake and the real batches
 errD = errD_real + errD_fake
 # Update D
 optimizerD.step()

 # (2) Update G network: maximize log(D(G(z)))
 errG = 0
 label.fill_(real_label) # fake labels are real for generator cost
 # Since we just updated D, perform another forward pass of all-fake batch through D
 output = netD(fake).view(-1)
 # Calculate G's loss based on this output
 errG = criterion(output, label)
 # Calculate gradients for G
 errG.backward()
 G_x = output.mean().item()
 # Update G
 optimizerG.step()

 # Output training status
 if i % 100 == 0:
 print(f'[Epoch {epoch+1}] Loss_D: {errD:.4f} Loss_G: {errG:.4f} D(G(z)): {D_y_1:.4f} / {D_x:.4f}'
 f' G(G(z)): {G_x:.4f} / {G_y_1:.4f}')
 # Save losses for plotting later
 l_losses.append(errD.item())
 g_losses.append(errG.item())

 # Check how the generator is doing by saving G's output on fixed noise
 if (i % 1000 == 0) or (epoch == num_epochs-1) and (i % 1000 != 0):
 with torch.no_grad():
 fake = netG(fixed_noise).detach().cpu()
 img_list.append(utils.make_grid(fake, padding=2, normalize=True))

 i += 1
"

Whereas in terms of this generator what we do for the fake image itself only there they keep the target value as 1. So that we are fooling the discriminator here using generator, using generator this is how it goes and the lost values, error values everything is computed.

(Refer Slide Time: 44:01)

Handwritten red notes on the slide:

- Each time \rightarrow 128 \rightarrow 128 \rightarrow 128
- 41 total, 41 batches or 41 iterations

Code snippet from the notebook:

```

% (epoch, num_epochs, 1, len(data_loader),
  errd.item(), errd.item(), 0.5, 0.5, 0.5, 0.5)

# Save losses for plotting later
g_losses.append(errd.item())
d_losses.append(errd.item())

# Check how the generator is doing by saving g's output on fixed_noise
if (iters % 100 == 0) or (epoch == num_epochs-1 and (1 == len(data_loader)-1)):
    with torch.no_grad():
        fake = netG(fixed_noise).detach().cpu()
    log_list.append(utils.make_grid(fake, padding=2, normalize=True))

    iters += 1
  
```

Output of the training loop:

```

Starting Training Loop...
[0/5][0/1583] Loss_G: 1.4264 Loss_D: 5.5242 D(x): 0.9733 D(G(z)): 0.5501 / 0.0065
[0/5][100/1583] Loss_G: 0.3336 Loss_D: 23.4499 D(x): 0.9562 D(G(z)): 0.0000 / 0.0000
[0/5][150/1583] Loss_G: 0.5438 Loss_D: 7.2868 D(x): 0.9551 D(G(z)): 0.0490 / 0.0012
[0/5][150/1583] Loss_G: 2.7968 Loss_D: 13.1551 D(x): 0.9925 D(G(z)): 0.8622 / 0.0000
[0/5][200/1583] Loss_G: 0.7864 Loss_D: 7.3992 D(x): 0.9516 D(G(z)): 0.4244 / 0.0017
  
```

Suppose, I have already mentioned So, in our case our data set we have around some 5 8 something something like 128 batch size we have taken that means in each epoch, each epoch contains a total of 41 total, 41 batches or 41 iterations.

(Refer Slide Time: 44:27)

Handwritten red notes on the slide:

- Epoch \rightarrow 40 \rightarrow 1/40 \rightarrow 10/40 \rightarrow 20/40 \rightarrow 30/40 \rightarrow 40/40

Code snippet from the notebook:

```

plt.figure(figsize=(10,3))
  
```

Results section:

Finally, let's check out how we did. Here, we will look at three different results. First, we will see how D and G's losses changed during training. Second, we will visualize G's output on the fixed_noise batch for every epoch. And third, we will look at a batch of real data next to a batch of fake data from G.

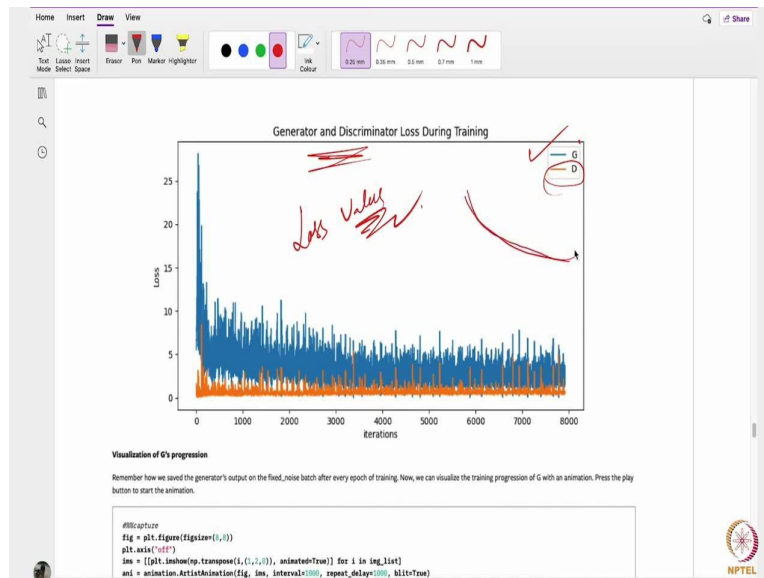
Loss versus training iteration

Below is a plot of D & G's losses versus training iterations.

So, if you observe the results here, here they have use 5 epoch for each and every epoch, we will have 40 iterations out of 40 iterations we are going to see only for first one and 11th one

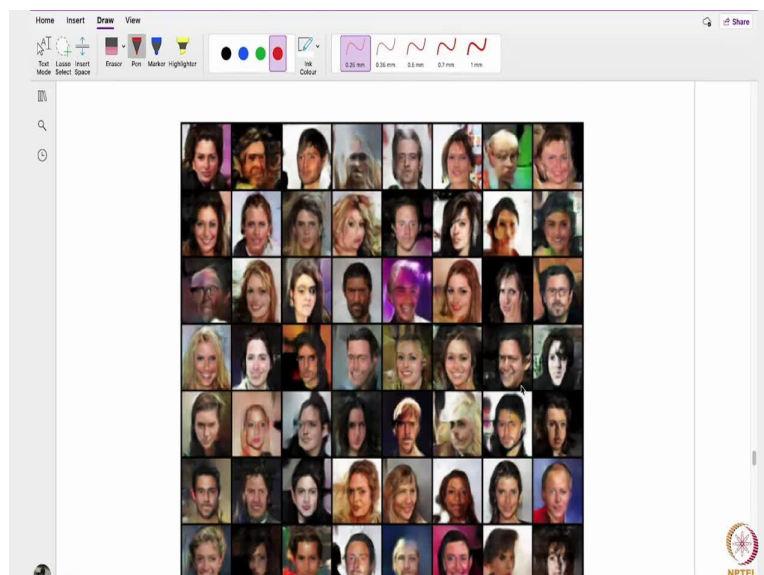
or 10th or 20th iteration likewise we will see the output once I will show you how to run the code also. But as an explanation sake I am saying.

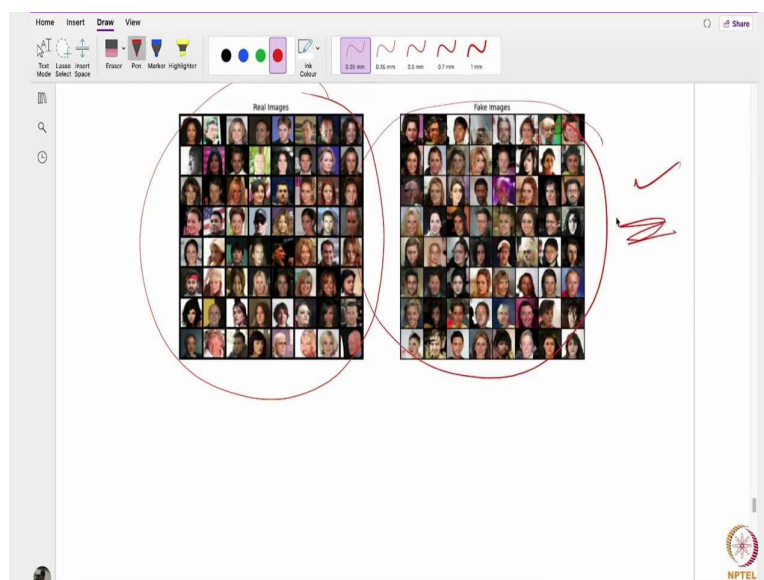
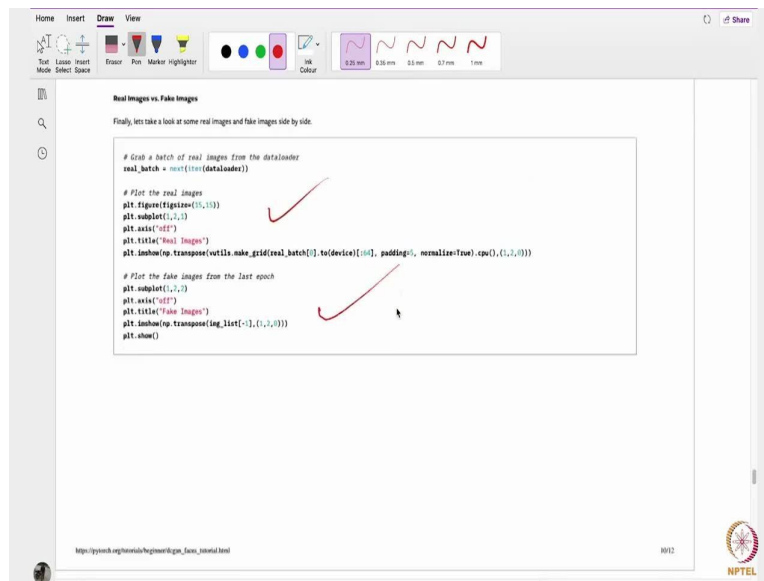
(Refer Slide Time: 44:56)



Now coming back to the results. Here you can see the loss functions loss value, discriminator loss and generator loss both are computed here, this color indicates discriminator and a blue color indicates generator loss. End of the day both the loss should decrease.

(Refer Slide Time: 45:24)





Now, this is just for the sake of how generated is progressing by taking the input random noise vectors and outputting see here, all these images are not completely, not completely drawn, you can see slight modifications are there, that means they are at the initial stage of trainings, but as you pass by if you do very good training, all of them will come in to a very good pictures later on.

At the end of this training, after playing so, many ways after playing around with the hyper parameters during the training, you just have to verify the results here, here they have drawn the real images and fake images as an output. This is what shown here, this is the final thing, this is the complete explanation sake.

(Refer Slide Time: 46:15)

Home Insert Draw View

Test Lesson Insert Mode Select Space Eraser Pen Marker Highlighter Ink Colour 0.25 mm 0.50 mm 0.8 mm 1 mm

Data

In this tutorial we will use the [Celeb-A Faces dataset](#) which can be downloaded at the linked site, or in [Google Drive](#). The dataset will download as a file named `img_align_celeba.zip`. Once downloaded, create a directory named `celeba` and extract the zip file into that directory. Then, set the dataset input for this notebook to the `celeba` directory you just created. The resulting directory structure should be:

```
/path/to/celeba
-> img_align_celeba
  -> 186267.jpg
  -> 179823.jpg
  -> 186787.jpg
  -> 537394.jpg
  ...
```

This is an important step because we will be using the `ImageFolder` dataset class, which requires there to be subdirectories in the dataset's root folder. Now, we can create the dataset, create the dataloader, set the device to run on, and finally visualize some of the training data.

<https://pytorch.org/tutorials/beginner/imagenet.html>

06/10/2022, 09:00 DCGAN Tutorial - PyTorch Tutorial 1.11 (lec10)2 documentation

```
# We can use an image folder dataset the way we have it setup.
# Create the dataset
dataset = datasets.ImageFolder(root=datasetroot,
                              transform=transforms.Compose([

```

Handwritten notes: A large 'X' is drawn over the directory structure. A box labeled 'CelebA/train' is drawn, with an arrow pointing to 'New' and another pointing to 'Normal'.

Home Insert Draw View

Test Lesson Insert Mode Select Space Eraser Pen Marker Highlighter Ink Colour 0.25 mm 0.50 mm 0.8 mm 1 mm

06/10/2022, 09:00 DCGAN Tutorial - PyTorch Tutorial 1.11 (lec10)2 documentation

```
# We can use an image folder dataset the way we have it setup.
# Create the dataset
dataset = datasets.ImageFolder(root=datasetroot,
                              transform=transforms.Compose([
    transforms.Resize(image_size),
    transforms.CenterCrop(image_size),
    transforms.ToTensor(),
    transforms.Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5]),
  ]))


# Create the dataloader
dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                                         shuffle=True, num_workers=workers)

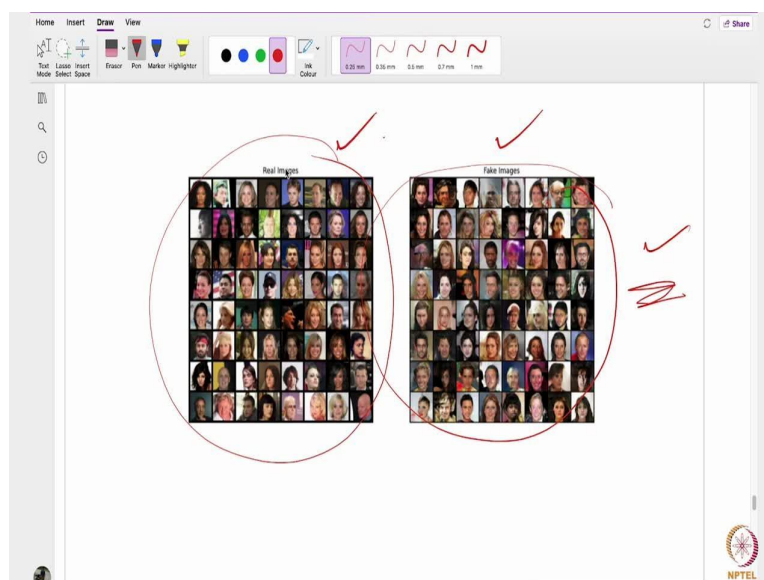
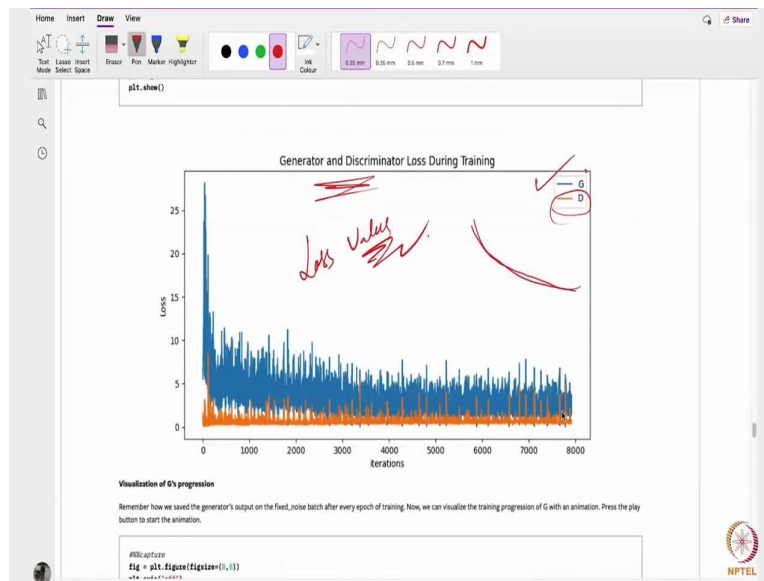
# Decide which device we want to run on
device = torch.device("cuda:0" if (torch.cuda.is_available() and gpu > 0) else "cpu")

# Plot some training images
real_batch = next(iter(dataloader))
plt.figure(figsize=(8,8))
plt.axis('off')
plt.title("Training Images")
plt.imshow(torch.transpose(torch.cat([real_batch[0].to(device)[0, padding:], normalize=True].cpu().float(), 0, 1, 2]))
```

Handwritten notes: A box is drawn around the `transforms.Compose` list. The word 'mean' is written next to the `transforms.Normalize` line. A box labeled '1/28' is drawn next to the `padding` parameter in the `plt.imshow` line.

Training Images

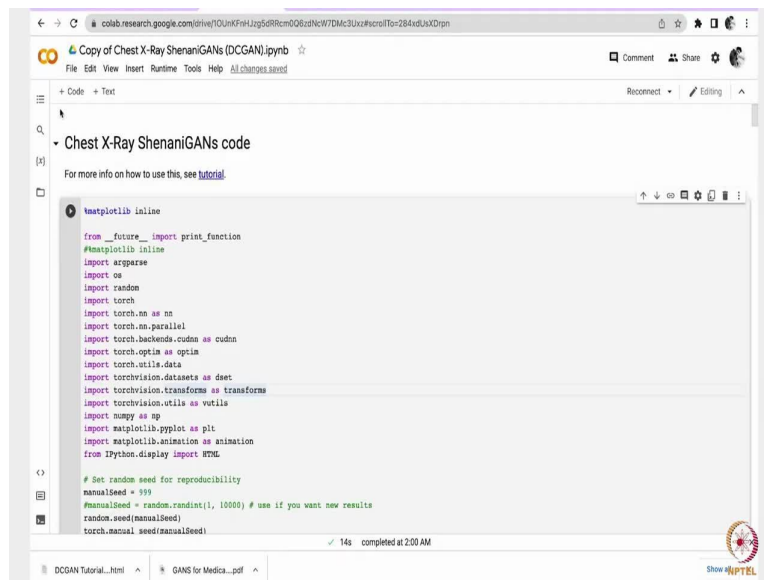




As a summary note, let me explain again, in summary, we have to report all the modules, here we have to set the hyper parameters, we have to set the hyper parameters, here the data set, sub directory has to be kept in a certain format using this data set and data loader we are going to convert the entire image data set into batches.

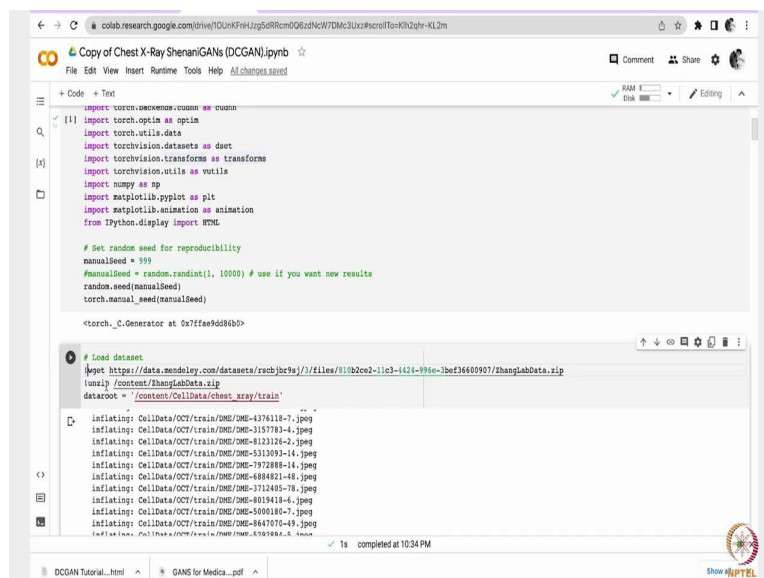
Later on the training loop will come, generator, discriminator, loss functions, training loop and we have drawn the graph of the generator and discriminator loss during training and at the end we have drawn the images generated by generator and real images that are present in the training data that is it, nothing else.

(Refer Slide Time: 47:13)



```
from __future__ import print_function
import argparse
import os
import random
import torch
import torch.nn as nn
import torch.nn.parallel
import torch.backends.cudnn as cudnn
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as dset
import torchvision.transforms as transforms
import torchvision.utils as vutils
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML

# Set random seed for reproducibility
manualSeed = 999
#manualSeed = random.randint(1, 10000) # use if you want new results
random.seed(manualSeed)
torch.manual_seed(manualSeed)
```



```
import torch.backends.cudnn as cudnn
import torch.optim as optim
import torch.utils.data
import torchvision.datasets as dset
import torchvision.transforms as transforms
import torchvision.utils as vutils
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from IPython.display import HTML

# Set random seed for reproducibility
manualSeed = 999
#manualSeed = random.randint(1, 10000) # use if you want new results
random.seed(manualSeed)
torch.manual_seed(manualSeed)

<torch._C.Generator at 0a7f5e9d860>

# Load dataset
url = 'https://data.mendeley.com/datasets/rxch3c9s/1/files/810b2ee2-11c1-4d24-996e-3be16600907/ZhangLabData.zip'
!unzip /content/ZhangLabData.zip
dataroot = '/content/CellData/chest_xray/train'
```

Now, let us get back to code how exactly you can use this notebook this is the code actually, see here I will run each and everything just focus. Once you open the Google colab, you can see this one, it takes some time you need some patients. First part is done, here we are going to download the data set. If you observe clearly from this link, from this link, we are downloading the data that zip data has been unzipped here. See here, this file name containing data set has been downloaded zip file that zip file has been extracted.

(Refer Slide Time: 48:31)

```
Copy of Chest X-Ray ShenanGANs (DCGAN).ipynb
File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

[1] #manualSeed = random.randint(1, 10000) # use if you want new results
    random.seed(manualSeed)
    torch.manual_seed(manualSeed)

<torch._C.Generator at 0x7ffae9d86b0>

# Load dataset
!wget https://data.mendeley.com/datasets/rcb3j9r5j/3/files/810b2c2-11c3-4424-996e-1bef3660997/ZhangLabData.zip
!unzip /content/ZhangLabData.zip
dataroot = '/content/CellData/chest_xray/train'

...
--2022-05-06 17:03:11-- https://data.mendeley.com/datasets/rcb3j9r5j/3/files/810b2c2-11c3-4424-996e-1bef3660997/ZhangLabData.zip
Resolving data.mendeley.com (data.mendeley.com)... 162.159.133.86, 162.159.130.86
Connecting to data.mendeley.com (data.mendeley.com)|162.159.133.86|:443... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: /v1/datasets/rcb3j9r5j/3/files/810b2c2-11c3-4424-996e-1bef3660997/ZhangLabData.zip [following]
--2022-05-06 17:03:11-- https://data.mendeley.com/v1/datasets/rcb3j9r5j/3/files/810b2c2-11c3-4424-996e-1bef3660997/ZhangLabData.zip
Reusing existing connection to data.mendeley.com:443.
HTTP request sent, awaiting response... 302 Found
Location: https://md-datasets-public-files-prod.s3.eu-west-1.amazonaws.com/ea18c27-67da-40a2-8f48-1c12f33de13d [following]
--2022-05-06 17:03:12-- https://md-datasets-public-files-prod.s3.eu-west-1.amazonaws.com/ea18c27-67da-40a2-8f48-1c12f33de13d
Resolving md-datasets-public-files-prod.s3.eu-west-1.amazonaws.com (md-datasets-public-files-prod.s3.eu-west-1.amazonaws.com)... 52.218.30.0
Connecting to md-datasets-public-files-prod.s3.eu-west-1.amazonaws.com (md-datasets-public-files-prod.s3.eu-west-1.amazonaws.com)|52.218.30.0|:443... connect
HTTP request sent, awaiting response... 200 OK
Length: 8441154111 (7.9G) [application/zip]
Saving to: 'ZhangLabData.zip'

ZhangLabData.zip 69[>] 499.71M 20.7MB/s eta 6m 29s

https://md-datasets-public-files-prod.s3.eu-west-1.amazonaws.com/ea18c27-67da-40a2-8f48-1c12f33de13d?response_type=command>_monitor_process>_poll_process

DCGAN Tutorial...html GANS for Medical...pdf Show #PTREL
```

```
Copy of Chest X-Ray ShenanGANs (DCGAN).ipynb
File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

[1] #manualSeed = random.randint(1, 10000) # use if you want new results
    random.seed(manualSeed)
    torch.manual_seed(manualSeed)

<torch._C.Generator at 0x7ffae9d86b0>

# Load dataset
!wget https://data.mendeley.com/datasets/rcb3j9r5j/3/files/810b2c2-11c3-4424-996e-1bef3660997/ZhangLabData.zip
!unzip /content/ZhangLabData.zip
dataroot = '/content/CellData/chest_xray/train'

...
--2022-05-06 17:03:11-- https://data.mendeley.com/datasets/rcb3j9r5j/3/files/810b2c2-11c3-4424-996e-1bef3660997/Zhang
Resolving data.mendeley.com (data.mendeley.com)... 162.159.133.86, 162.159.130.86
Connecting to data.mendeley.com (data.mendeley.com)|162.159.133.86|:443... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: /v1/datasets/rcb3j9r5j/3/files/810b2c2-11c3-4424-996e-1bef3660997/ZhangLabData.zip [following]
--2022-05-06 17:03:11-- https://data.mendeley.com/v1/datasets/rcb3j9r5j/3/files/810b2c2-11c3-4424-996e-1bef3660997/Zh
Reusing existing connection to data.mendeley.com:443.
HTTP request sent, awaiting response... 302 Found
Location: https://md-datasets-public-files-prod.s3.eu-west-1.amazonaws.com/ea18c27-67da-40a2-8f48-1c12f33de13d [following]
--2022-05-06 17:03:12-- https://md-datasets-public-files-prod.s3.eu-west-1.amazonaws.com/ea18c27-67da-40a2-8f48-1c12f3
Resolving md-datasets-public-files-prod.s3.eu-west-1.amazonaws.com (md-datasets-public-files-prod.s3.eu-west-1.amazonaws
Connecting to md-datasets-public-files-prod.s3.eu-west-1.amazonaws.com (md-datasets-public-files-prod.s3.eu-west-1.amazon
HTTP request sent, awaiting response... 200 OK
Length: 8441154111 (7.9G) [application/zip]
Saving to: 'ZhangLabData.zip'

ZhangLabData.zip 15[=>] 1.22G 20.9MB/s eta 5m 37s

Disk 38.86 GB available

Executing [tm 3s] Cell > system() > _system_compact() > _run_command() > _monitor_process() > _poll_process()

DCGAN Tutorial...html GANS for Medical...pdf Show #PTREL
```

```
Copy of Chest X-Ray ShenanGANs (DCGAN).ipynb
File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

[1] #manualSeed = random.randint(1, 10000) # use if you want new results
    random.seed(manualSeed)
    torch.manual_seed(manualSeed)

<torch._C.Generator at 0x7ffae9d86b0>

# Load dataset
!wget https://data.mendeley.com/datasets/rcb3j9r5j/3/files/810b2c2-11c3-4424-996e-1bef3660997/ZhangLabData.zip
!unzip /content/ZhangLabData.zip
dataroot = '/content/CellData/chest_xray/train'

...
inflatings: CellData/chest_xray/train/NORMAL/NORMAL-9020082-0002.jpg
inflatings: CellData/chest_xray/train/NORMAL/NORMAL-3400392-0002.jpg
inflatings: CellData/chest_xray/train/NORMAL/NORMAL-6931624-0001.jpg
inflatings: CellData/chest_xray/train/NORMAL/NORMAL-2748393-0004.jpg
inflatings: CellData/chest_xray/train/NORMAL/NORMAL-996167-0001.jpg
inflatings: CellData/chest_xray/train/NORMAL/NORMAL-2251508-0001.jpg
inflatings: CellData/chest_xray/train/NORMAL/NORMAL-2162388-0001.jpg
inflatings: CellData/chest_xray/train/NORMAL/NORMAL-3488729-0001.jpg
inflatings: CellData/chest_xray/train/NORMAL/NORMAL-3119436-0001.jpg
inflatings: CellData/chest_xray/train/NORMAL/NORMAL-6474992-0001.jpg
inflatings: CellData/chest_xray/train/NORMAL/NORMAL-7749598-0001.jpg
inflatings: CellData/chest_xray/train/NORMAL/NORMAL-6082928-0001.jpg
inflatings: CellData/chest_xray/train/NORMAL/NORMAL-1055629-0002.jpg
inflatings: CellData/chest_xray/train/NORMAL/NORMAL-9346821-0001.jpg
inflatings: CellData/chest_xray/train/NORMAL/NORMAL-385855-0002.jpg
inflatings: CellData/chest_xray/train/NORMAL/NORMAL-8648239-0004.jpg
inflatings: CellData/chest_xray/train/NORMAL/NORMAL-7920731-0001.jpg
inflatings: CellData/chest_xray/train/NORMAL/NORMAL-9084382-0002.jpg
inflatings: CellData/chest_xray/train/NORMAL/NORMAL-9748834-0001.jpg
inflatings: CellData/chest_xray/train/NORMAL/NORMAL-9138087-0001.jpg
inflatings: CellData/chest_xray/train/NORMAL/NORMAL-9322605-0001.jpg
inflatings: CellData/chest_xray/train/NORMAL/NORMAL-1196989-0001.jpg
inflatings: CellData/chest_xray/train/NORMAL/NORMAL-7252162-0003.jpg
inflatings: CellData/chest_xray/train/NORMAL/NORMAL-3848904-0002.jpg

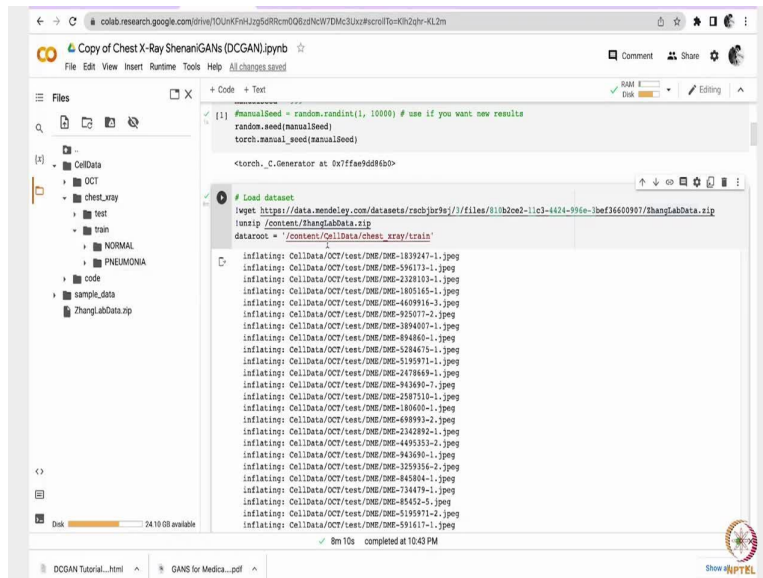
Disk 22.22 GB available

Executing [tm 3s] Cell > system() > _system_compact() > _run_command() > _monitor_process() > _poll_process()

DCGAN Tutorial...html GANS for Medical...pdf Show #PTREL
```

Later on we are going to keep the data root value. See how it goes. Let us download the data set. It takes some time.

(Refer Slide Time: 56:45)



```
# ManualSeed = random.randint(1, 10000) # use if you want new results
random.seed(manualSeed)
torch.manual_seed(manualSeed)

<torch._C.Generator at 0x7f9e9d86b2d>

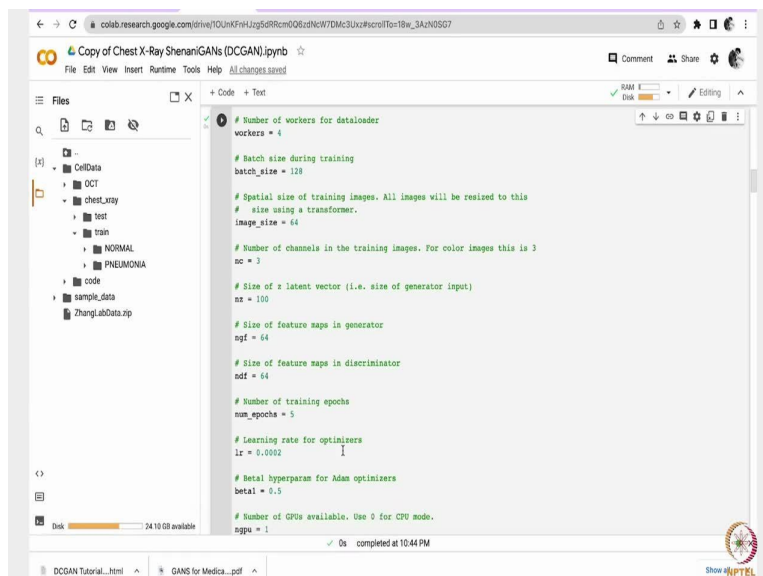
# Load dataset
wget https://data.mendeley.com/datasets/cxrbjbrks/1/files/81b2ce2-1ic3-4d24-996e-bef3660997/ZhangLabData.zip
unzip /content/ZhangLabData.zip
dataroot = '/content/CellData/chest_xray/train'

Inflating: CellData/OCT/test/DME-1839247-1.jpeg
Inflating: CellData/OCT/test/DME-596173-1.jpeg
Inflating: CellData/OCT/test/DME-2328103-1.jpeg
Inflating: CellData/OCT/test/DME-1805145-1.jpeg
Inflating: CellData/OCT/test/DME-4609916-3.jpeg
Inflating: CellData/OCT/test/DME-925077-2.jpeg
Inflating: CellData/OCT/test/DME-1894807-1.jpeg
Inflating: CellData/OCT/test/DME-2479649-1.jpeg
Inflating: CellData/OCT/test/DME-894860-1.jpeg
Inflating: CellData/OCT/test/DME-5284675-1.jpeg
Inflating: CellData/OCT/test/DME-5195971-1.jpeg
Inflating: CellData/OCT/test/DME-2479649-1.jpeg
Inflating: CellData/OCT/test/DME-894860-1.jpeg
Inflating: CellData/OCT/test/DME-943690-7.jpeg
Inflating: CellData/OCT/test/DME-2507510-1.jpeg
Inflating: CellData/OCT/test/DME-180605-1.jpeg
Inflating: CellData/OCT/test/DME-690993-2.jpeg
Inflating: CellData/OCT/test/DME-2342892-1.jpeg
Inflating: CellData/OCT/test/DME-4495353-2.jpeg
Inflating: CellData/OCT/test/DME-943690-1.jpeg
Inflating: CellData/OCT/test/DME-3259355-2.jpeg
Inflating: CellData/OCT/test/DME-845804-1.jpeg
Inflating: CellData/OCT/test/DME-734479-1.jpeg
Inflating: CellData/OCT/test/DME-85452-5.jpeg
Inflating: CellData/OCT/test/DME-5195971-2.jpeg
Inflating: CellData/OCT/test/DME-591617-1.jpeg
```

Now, you can see that after downloading the data, we have personal details downloaded that is failing sorry. The thing is cell data has been extracted from this particular file name, if we observe in the cell data we have was OCT, chest X ray and code. But as of now, we will use chest X ray dataset.

And in chest X ray data set my aim is just to focus on training data, train folder only and inside see here, this is the content to extract the subdirectory of the train, just have to click here and copy path and that path will be this one. So, do this one, one should download the data and everything has been done.

(Refer Slide Time: 57:34)



```
# Number of workers for dataloader
workers = 4

# Batch size during training
batch_size = 128

# Spatial size of training images. All images will be resized to this
# size using a transformer.
image_size = 64

# Number of channels in the training images. For color images this is 3
nc = 3

# Size of z latent vector (i.e. size of generator input)
nz = 100

# Size of feature maps in generator
ngf = 64

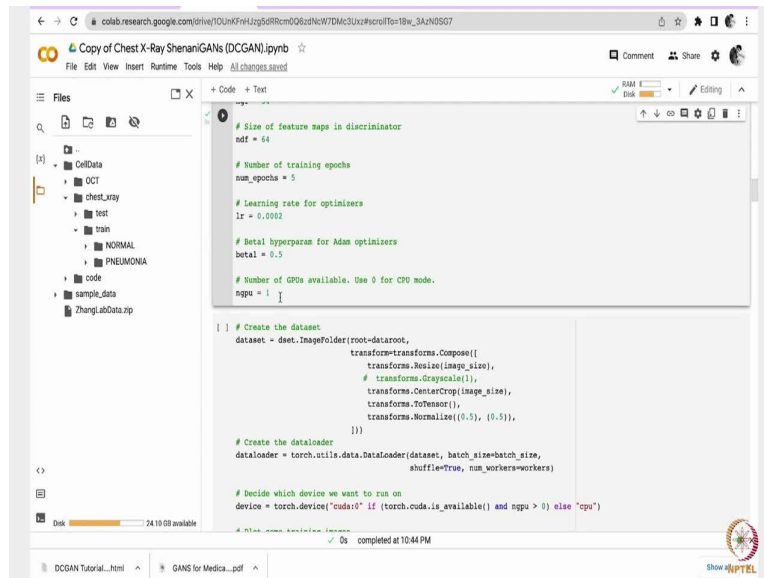
# Size of feature maps in discriminator
ndf = 64

# Number of training epochs
num_epochs = 5

# Learning rate for optimizers
lr = 0.0002

# Beta1 hyperparam for Adam optimizers
beta1 = 0.5

# Number of GPUs available. Use 0 for CPU mode.
ngpu = 1
```

```
# Size of feature maps in discriminator
ndf = 64

# Number of training epochs
num_epochs = 5

# Learning rate for optimizers
lr = 0.0002

# Beta1 hyperparam for Adam optimizers
beta1 = 0.5

# Number of GPUs available. Use 0 for CPU mode.
ngpu = 1

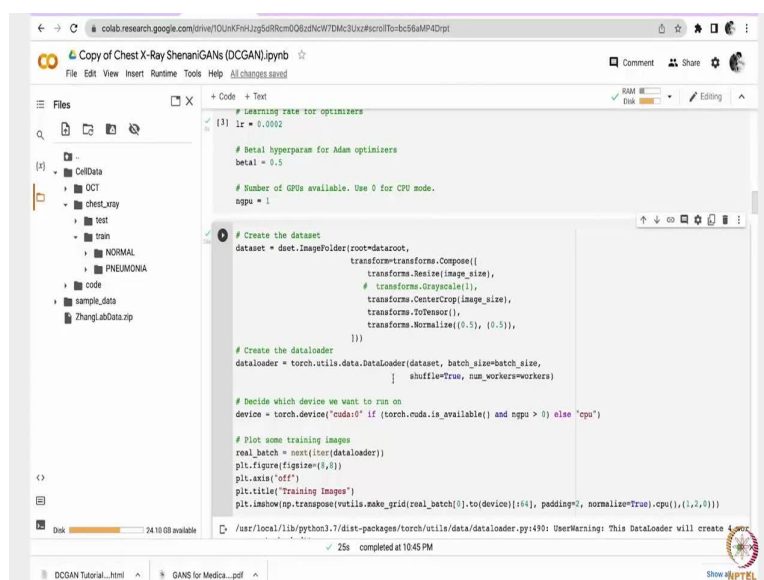
# Create the dataset
dataset = datasets.ImageFolder(root=datasetroot,
                              transform=transforms.Compose([
                                  transforms.Resize(image_size),
                                  transforms.Grayscale(1),
                                  transforms.CenterCrop(image_size),
                                  transforms.ToTensor(),
                                  transforms.Normalize([0.5], [0.5]),
                              ]))

# Create the dataloader
dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                                          shuffle=True, num_workers=workers)

# Decide which device we want to run on
device = torch.device("cuda:0" if (torch.cuda.is_available() and ngpu > 0) else "cpu")
```

Now, let us get back to setting the hyper parameters everything let us run this one, number of channels are 3 GB images next ngf ngf view, next thing is learning rate for both optimizers has been kept as 0.0002 later on when we, we can play around that is not an issue β_1 has been kept as 0.5 for batch optimizers, GPU is number of GPUs are available suppose if you have 4 or 5 GPUs also you can do for parallel processing also. Suppose the data set you have is very large and you have the available computational facility of having 4 or 5 GPUs you can go for parallel processing that has also been accepted.

(Refer Slide Time: 58:12)



```
# Learning rate for optimizers
lr = 0.0002

# Beta1 hyperparam for Adam optimizers
beta1 = 0.5

# Number of GPUs available. Use 0 for CPU mode.
ngpu = 1

# Create the dataset
dataset = datasets.ImageFolder(root=datasetroot,
                              transform=transforms.Compose([
                                  transforms.Resize(image_size),
                                  transforms.Grayscale(1),
                                  transforms.CenterCrop(image_size),
                                  transforms.ToTensor(),
                                  transforms.Normalize([0.5], [0.5]),
                              ]))

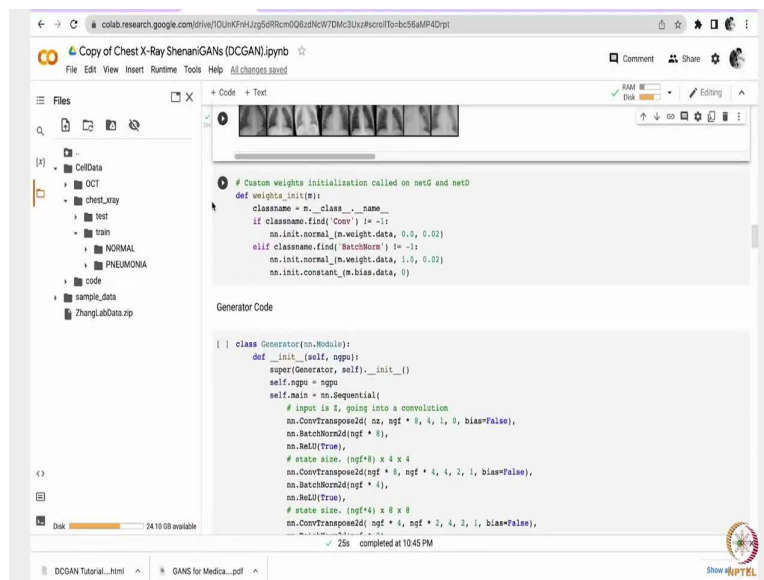
# Create the dataloader
dataloader = torch.utils.data.DataLoader(dataset, batch_size=batch_size,
                                          shuffle=True, num_workers=workers)

# Decide which device we want to run on
device = torch.device("cuda:0" if (torch.cuda.is_available() and ngpu > 0) else "cpu")

# Plot some training images
real_batch = next(iter(dataloader))
plt.figure(figsize=(8,8))
plt.axis('off')
plt.title('Training Images')
plt.imshow(np.transpose(utils.make_grid(real_batch[0].to(device)[:64], padding=0, normalize=True).cpu(),(1,2,3)))
```


Now go for dataset creation, go for dataset creation, it takes some time because in batches there are a lot of difference in batches. Now, the data loader has been generated at the same time we are extracting some 64 images and they are all will be shown here. See here other chest X ray images, there all digital images I will show you see here inside the training data. Normal means see here this is very huge in space they are above 1000 also I mean size of the image, know the pixel. You have seen this one every data contains 1000 by 129, 2930 like 4 digit numbers on the rows and both columns also.

(Refer Slide Time: 59:36)



```
Copy of Chest X-Ray ShenanGANs (DCGAN).ipynb
File Edit View Insert Runtime Tools Help All changes saved
RAM 16GB
Disk 24GB available
Editing

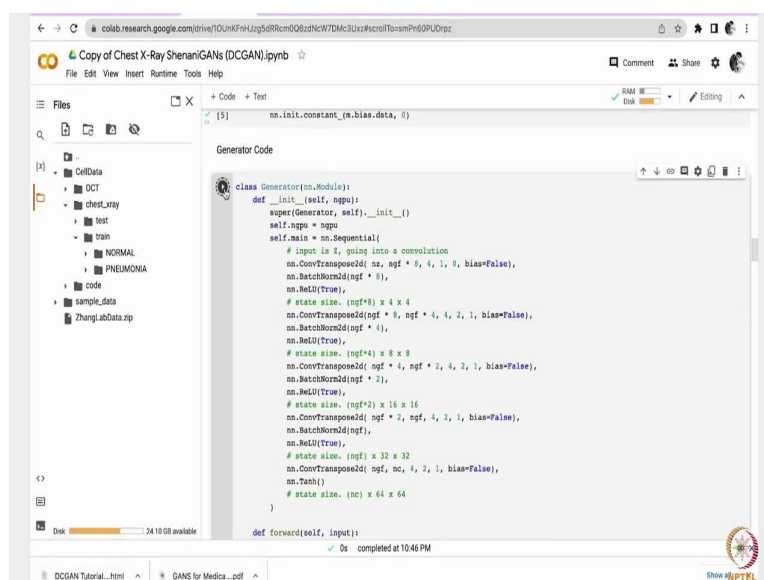
[5]
CellData
├── OCT
├── chest_xray
├── test
├── train
├── NORMAL
├── PNEUMONIA
├── code
├── sample_data
└── ZhangLabData.zip

# Custom weights initialization called on netG and netD
def weights_init(n):
    classname = n.__class__.__name__
    if classname.find('Conv') != -1:
        nn.init.normal_(n.weight.data, 0.0, 0.02)
    elif classname.find('BatchNorm') != -1:
        nn.init.normal_(n.weight.data, 1.0, 0.02)
        nn.init.constant_(n.bias.data, 0)

Generator Code

class Generator(nn.Module):
    def __init__(self, ngpu):
        super(Generator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is 1, going into a convolution
            nn.ConvTranspose2d(1, ngf * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            # state size. (ngf*8) x 4 x 4
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # state size. (ngf*4) x 8 x 8
            nn.ConvTranspose2d(ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # state size. (ngf*2) x 16 x 16
            nn.ConvTranspose2d(ngf * 2, ngf * 1, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 1),
            nn.ReLU(True),
            # state size. (ngf) x 32 x 32
            nn.ConvTranspose2d(ngf, 1, 4, 2, 1, bias=False),
            nn.Tanh()
        )
        # state size. (1) x 64 x 64

    def forward(self, input):
```



```
nn.init.constant_(n.bias.data, 0)

Generator Code

class Generator(nn.Module):
    def __init__(self, ngpu):
        super(Generator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is 1, going into a convolution
            nn.ConvTranspose2d(1, ngf * 8, 4, 1, 0, bias=False),
            nn.BatchNorm2d(ngf * 8),
            nn.ReLU(True),
            # state size. (ngf*8) x 4 x 4
            nn.ConvTranspose2d(ngf * 8, ngf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 4),
            nn.ReLU(True),
            # state size. (ngf*4) x 8 x 8
            nn.ConvTranspose2d(ngf * 4, ngf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 2),
            nn.ReLU(True),
            # state size. (ngf*2) x 16 x 16
            nn.ConvTranspose2d(ngf * 2, ngf * 1, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ngf * 1),
            nn.ReLU(True),
            # state size. (ngf) x 32 x 32
            nn.ConvTranspose2d(ngf, 1, 4, 2, 1, bias=False),
            nn.Tanh()
        )
        # state size. (1) x 64 x 64

    def forward(self, input):
```

Now coming back to custom weight initialization on generator and discriminator. This is generator code.

(Refer Slide Time: 59:50)

```
# Create the generator
netG = Generator(ngpu).to(device)

# Handle multi-gpu if desired
if (device.type == 'cuda') and (ngpu > 1):
    netG = nn.DataParallel(netG, list(range(ngpu)))

# Apply the weights_init function to randomly initialize all weights
# to mean=0, stdev=0.2.
netG.apply(weights_init)

# Print the model
print(netG)

Generator(
  (main): Sequential(
    (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
    (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU(inplace=True)
    (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (5): ReLU(inplace=True)
    (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): ReLU(inplace=True)
    (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (11): ReLU(inplace=True)
    (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
    (13): Tanh()
  )
)
```

Next thing is we will convert the model of generator into GPU model. The initialization weights are also done here.

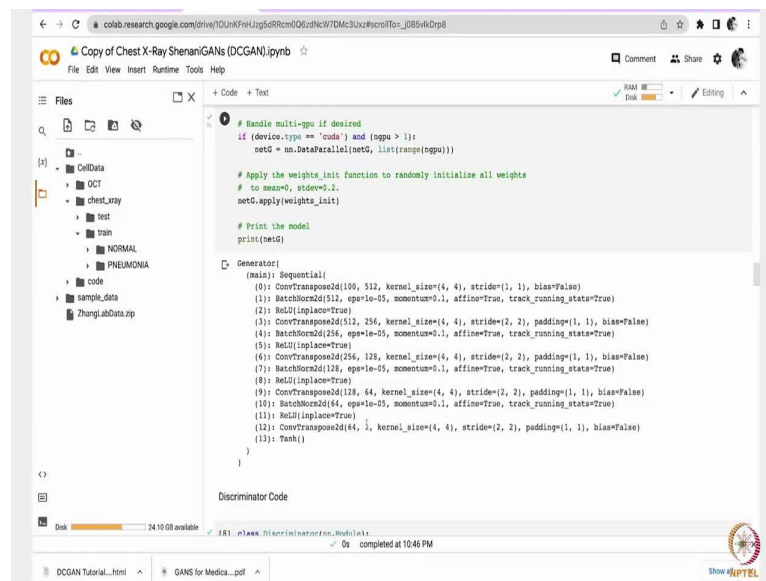
(Refer Slide Time: 59:56)

```
class Discriminator(nn.Module):
    def __init__(self, ngpu):
        super(Discriminator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is (nc) x 64 x 64
            nn.Conv2d(nc, ndf, kernel_size=(5, 5), stride=(1, 1), bias=False),
            nn.BatchNorm2d(ndf, eps=0.2, inplace=True),
            # state size. (ndf) x 32 x 32
            nn.Conv2d(ndf, ndf, kernel_size=(5, 5), stride=(1, 1), bias=False),
            nn.BatchNorm2d(ndf, eps=0.2, inplace=True),
            # state size. (ndf) x 16 x 16
            nn.Conv2d(ndf, ndf, kernel_size=(5, 5), stride=(1, 1), bias=False),
            nn.BatchNorm2d(ndf, eps=0.2, inplace=True),
            # state size. (ndf) x 8 x 8
            nn.Conv2d(ndf, ndf, kernel_size=(5, 5), stride=(1, 1), bias=False),
            nn.BatchNorm2d(ndf, eps=0.2, inplace=True),
            # state size. (ndf) x 4 x 4
            nn.Conv2d(ndf, ndf, kernel_size=(5, 5), stride=(1, 1), bias=False),
            nn.Sigmoid()
        )

    def forward(self, input):
        return self.main(input)
```

Now coming back to discriminator block. It has been executed similarly discriminator.

(Refer Slide Time: 60:11)



The screenshot shows a Google Colab notebook titled "Copy of Chest X-Ray ShenanGANs (DCGAN).ipynb". The left sidebar displays a file explorer with folders like "CellData", "OCT", "chest_xray", "test", "train", "NORMAL", "PNEUMONIA", "code", "sample_data", and "ZhanglabData.zip". The main code area contains the following Python code:

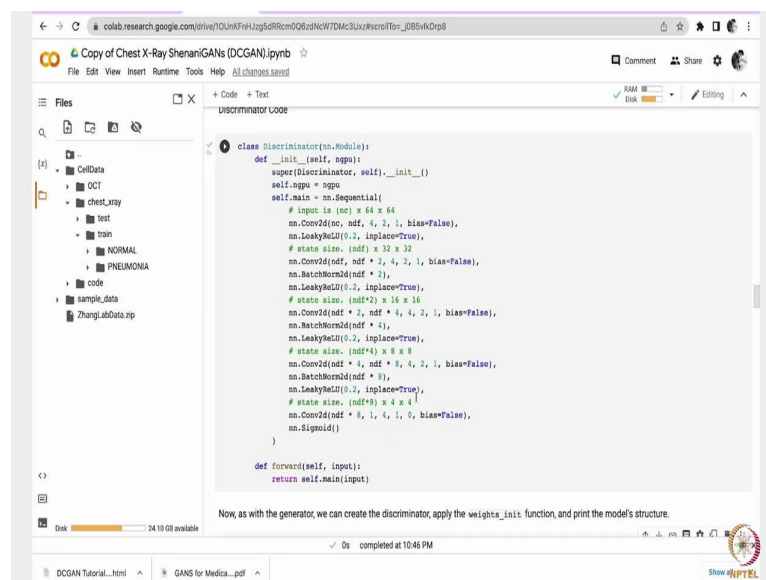
```
# Handle multi-gpu if desired
if (device.type == 'cuda') and (ngpu > 1):
    netG = nn.DataParallel(netG, list(range(ngpu)))

# Apply the weights_init function to randomly initialize all weights
# to mean=0, stdev=0.2.
netG.apply(weights_init)

# Print the model
print(netG)

Generator:
(main): Sequential(
  (0): ConvTranspose2d(100, 512, kernel_size=(4, 4), stride=(1, 1), bias=False)
  (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (2): ReLU(inplace=True)
  (3): ConvTranspose2d(512, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (4): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (5): ReLU(inplace=True)
  (6): ConvTranspose2d(256, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (8): ReLU(inplace=True)
  (9): ConvTranspose2d(128, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (10): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (11): ReLU(inplace=True)
  (12): ConvTranspose2d(64, 3, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (13): Tanh()
)
```

The bottom status bar indicates "181 classes Discriminator from Module" and "0s completed at 10:46 PM".



The screenshot shows the same Google Colab notebook, but now displaying the "Discriminator Code". The code defines a Discriminator class:

```
class Discriminator(nn.Module):
    def __init__(self, ngpu):
        super(Discriminator, self).__init__()
        self.ngpu = ngpu
        self.main = nn.Sequential(
            # input is (nc) x 64 x 64
            nn.Conv2d(nc, ndf, 4, 2, 1, bias=False),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf) x 32 x 32
            nn.Conv2d(ndf, ndf * 2, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*2) x 16 x 16
            nn.Conv2d(ndf * 2, ndf * 4, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*4) x 8 x 8
            nn.Conv2d(ndf * 4, ndf * 8, 4, 2, 1, bias=False),
            nn.BatchNorm2d(ndf * 8),
            nn.LeakyReLU(0.2, inplace=True),
            # state size. (ndf*8) x 4 x 4
            nn.Conv2d(ndf * 8, 1, 4, 1, 0, bias=False),
            nn.Sigmoid()
        )

    def forward(self, input):
        return self.main(input)
```

Below the code, a text note states: "Now, as with the generator, we can create the discriminator, apply the weights_init function, and print the model's structure." The bottom status bar shows "0s completed at 10:46 PM".

Observe here generator this is 100 dimensional input vector and outputting 3 channel image whereas the discriminator it is taking the input as a number of channels cross 64×64 here we have increase around only 64×64 and is giving an output as probability score see here.

(Refer Slide Time: 60:33)

```
# Copy of Chest X-Ray ShenanGANs (DCGAN).ipynb
# to cuda0, device=0.2

netD.apply(weights_init)

# Print the model
print(netD)

Discriminator()
(main): Sequential(
  (0): Conv2d(1, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (1): LeakyReLU(negative_slope=0.2, inplace=True)
  (2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (3): BatchNorm2d(128, eps=0.05, momentum=0.1, affine=True, track_running_stats=True)
  (4): LeakyReLU(negative_slope=0.2, inplace=True)
  (5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (6): BatchNorm2d(256, eps=0.05, momentum=0.1, affine=True, track_running_stats=True)
  (7): LeakyReLU(negative_slope=0.2, inplace=True)
  (8): Conv2d(256, 512, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1), bias=False)
  (9): BatchNorm2d(512, eps=0.05, momentum=0.1, affine=True, track_running_stats=True)
  (10): LeakyReLU(negative_slope=0.2, inplace=True)
  (11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), bias=False)
  (12): Sigmoid()
)

Train the GAN

[ ] # Initialize BCELoss function
criterion = nn.BCELoss()

# Create batch of latent vectors that we will use to visualize
# the progression of the generator
fixed_noise = torch.randn(64, nz, 1, 1, device=device)

# Establish convention for real and fake labels during training
real_label = 1
fake_label = 0

# Setup Adam optimizers for both G and D
optimizerD = optim.Adam(netD.parameters(), lr=0.0002, betas=(betal, 0.999))
optimizerG = optim.Adam(netG.parameters(), lr=0.001, betas=(betal, 0.999))

[ ] # Training Loop

# Lists to keep track of progress
img_list = []
G_losses = []
D_losses = []
iters = 0
num_epochs = 5
print('Starting Training Loop...')
# For each epoch
for epoch in range(num_epochs):
    # For each batch in the dataloader
```

Input as 3 channel image and outputting sigmoid sigmoid output is nothing but 0 to 1 rounded of 2. So, we are considering them as probability scores. Now, discriminator block is done, generator block is done and these models have been sorry both networks have been moved to device GPU devices. Next thing model converting model into GPU device in the sense all the parameters that have been converted into GPU tensors.

(Refer Slide Time: 61:04)

```
optimizerD = optim.Adam(netD.parameters(), lr=0.0002, betas=(betal, 0.999))
optimizerG = optim.Adam(netG.parameters(), lr=0.001, betas=(betal, 0.999))

[ ] # Training Loop

# Lists to keep track of progress
img_list = []
G_losses = []
D_losses = []
iters = 0
num_epochs = 5
print('Starting Training Loop...')
# For each epoch
for epoch in range(num_epochs):
    # For each batch in the dataloader
```

Now, coming back to training the GAN, here we are used the lost binary cross entropy loss only it is a noise this is for visualization purposes focusing on generator later on you will understand. The real label is 1 optimizer and discriminator see here, how changes the value to

But here, after doing some modifications or playing around with the values here, I came to know that maybe this value is giving somewhat good understanding of hyper parameter. So I kept this one but you can play around with first, you can start with both 0.0002, 0.0002 and later on you can do some changes to learning rate.

(Refer Slide Time: 61:50)

→ ↻ ↗ colab.research.google.com/share?1tOuhKFrH-Lzrg5SRmCtmQ6dZKWDMc3UxzhScvrlTnVUFUgq5GfDyQ

Colab of Chest X-Ray Shenanigans (DCGAN).ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

RAM ✓ Disk ✓ Editing ✓

Files

- [x] CellData
- [x] OCT
- [x] chest_xray
 - test
 - train
 - NORMAL
 - PNEUMONIA
 - code
- [x] sample_data
- [x] ZhangLi_nData.zip

```
def optimize_network(step):  
    #####  
    # (2) Update G network: maximize log(D(G(z)))  
    #####  
    netc.zero_grad()  
    label.fill_(real_label) # fake labels are real for generator cost  
    # Since we just updated D, perform another forward pass of all-fake batch through D  
    output = netd(fake).view(-1)  
    # Calculate G's loss based on this output  
    errG = criterion(output, label)  
    # Calculate gradients for G  
    errG.backward()  
    D_0_g2 = output.mean().item()  
    # Update G  
    optimizerG.step()  
  
    # Output training state  
    if i % 10 == 0:  
        print('%4d/%4d/%4d/%4d/%4d/%4d/%4d/%4d/%4d/%4d' % (  
            epoch, num_epochs, i, len(dataloader),  
            errD.item(), errG.item(), D_x, D_0_g1, D_0_g2))  
  
    # Save Losses for plotting later  
    G_losses.append(errG.item())  
    D_losses.append(errD.item())  
  
    # Check how the generator is doing by saving G's output on fixed_noise  
    if (iters % 500 == 0) or ((epoch == num_epochs-1) and (i == len(dataloader)-1)):  
        with torch.no_grad():  
            fake = netG(fixed_noise).detach().cpu()  
            img_list.append(utils.make_grid(fake, padding=2, normalize=True))  
  
    iters += 1
```

loading...

Disk 24.18 GB available

✓ completed at 10:48 PM

DCGAN Tutorial...html * GANS for Medical...pdf

Show help

```

# Copy of Chest X-Ray ShenanIGANs (DCGAN).ipynb
File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

optimizer.step()

# Output training stats
if i % 10 == 0:
    print('[%d/%d] [%d/%d] Loss_D: %.4f, Loss_G: %.4f, D(x): %.4f, D(z): %.4f / %.4f'
          % (epoch, num_epochs, i, len(data_loader),
             err_D.item(), err_G.item(), D_x, D_z_0.2, D_z_0.2))

# Save losses for plotting later
G_losses.append(err_G.item())
D_losses.append(err_D.item())

# Check how the generator is doing by saving G's output on fixed_noise
if (i iterz = 100 == 0) or (epoch == num_epochs - 1 and i == len(data_loader) - 1):
    with torch.no_grad():
        fake = netG(fixed_noise).detach().cpu()
        img_list.append(utils.make_grid(fake, padding=2, normalize=True))

iterz += 1

Starting Training Loop...
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:490: UserWarning: This DataLoader will create 4 worker
spawned, checked)

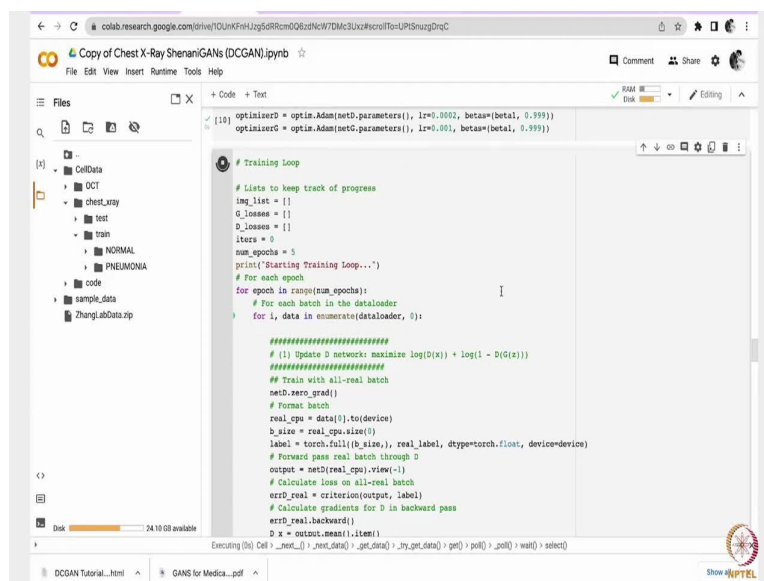
[0/5][0/41] Loss_D: 1.5541 Loss_G: 5.0495 D(x): 0.4314 D(z): 0.5837 / 0.0013
[0/5][10/41] Loss_D: 1.4240 Loss_G: 13.5476 D(x): 0.8480 D(z): 0.6424 / 0.0000
[0/5][20/41] Loss_D: 0.8421 Loss_G: 10.3384 D(x): 0.7503 D(z): 0.0953 / 0.0001
[0/5][30/41] Loss_D: 2.1911 Loss_G: 5.7345 D(x): 0.3036 D(z): 0.2121 / 0.0058
[0/5][40/41] Loss_D: 3.1456 Loss_G: 6.4448 D(x): 0.4644 D(z): 0.8172 / 0.0040
[1/5][0/41] Loss_D: 3.5002 Loss_G: 4.9144 D(x): 0.2064 D(z): 0.4249 / 0.0092
[1/5][10/41] Loss_D: 2.5659 Loss_G: 4.9992 D(x): 0.7006 D(z): 0.5249 / 0.0115
[1/5][20/41] Loss_D: 1.9244 Loss_G: 3.2037 D(x): 0.4124 D(z): 0.4831 / 0.0515
[1/5][30/41] Loss_D: 2.1879 Loss_G: 2.6449 D(x): 0.4013 D(z): 0.4395 / 0.0933
[1/5][40/41] Loss_D: 1.7907 Loss_G: 2.0522 D(x): 0.4194 D(z): 0.4931 / 0.1559
[2/5][0/41] Loss_D: 1.4879 Loss_G: 2.4569 D(x): 0.3304 D(z): 0.4869 / 0.2744
... ..
✓ completed at 10:48 PM
  
```

Let me run this one. Now, the final thing training loop, generator updatation and discriminator updatation, you can see here, as I told you earlier, we have only 41 batches, each batch contains of some 128 images out of this 41 also and in each and every epoch I have 41 batches. I just

want to show the, I want to see the loss value at after every 10 iterations that is why I have kept an $i/10 = 0$.

So, if you want to see out of these 41 iterations you may want to see for every 5 iterations just keep a 5 value here. Suppose 20, just keep a 20 value here. So, we are observing the discriminator loss, generator loss, output of the discriminator, at the same time output of the discriminator run fake image is given and at the same time real images is given, this is how the entire thing goes on.

(Refer Slide Time: 62:56)



```
optmizerD = optim.Adam(netD.parameters(), lr=0.0002, betas=(0.999, 0.999))
optmizerG = optim.Adam(netG.parameters(), lr=0.0001, betas=(0.999, 0.999))

# Training Loop

# Lists to keep track of progress
img_list = []
G_losses = []
D_losses = []
iters = 0
num_epochs = 5
print('Starting Training Loop...')
# For each epoch
for epoch in range(num_epochs):
    # For each batch in the dataloader
    for i, data in enumerate(dataloader, 0):

        #####
        # (1) Update D network: maximize log(D(x)) + log(1 - D(G(z)))
        #####
        ## Train with all-real batch
        netD.zero_grad()
        # Format batch
        real_cpu = data[0].to(device)
        b_size = real_cpu.size(0)
        label = torch.full((b_size, 1), 1, device=device)
        # Forward pass real batch through D
        output = netD(real_cpu).view(-1)
        # Calculate loss on all-real batch
        errD_real = criterion(output, label)
        # Calculate gradients for D in backward pass
        errD_real.backward()
        D_x = output.mean().item()
        D_x = netD(data[0].to(device)).get_device()

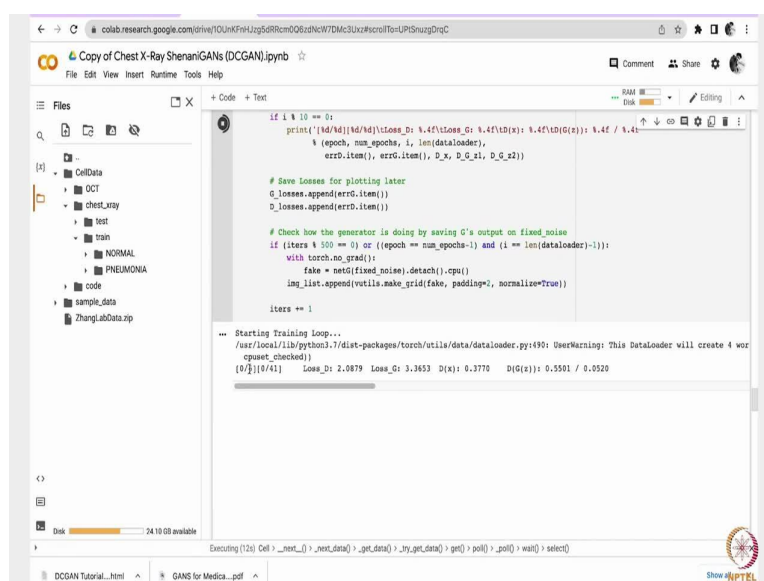
        #####
        # (2) Update G network: minimize D(G(z))
        #####
        # Train with all-fake batch
        netG.zero_grad()
        # Format batch
        fake_cpu = data[1].to(device)
        b_size = fake_cpu.size(0)
        label = torch.full((b_size, 1), 0, device=device)
        # Forward pass fake batch through D
        output = netD(fake_cpu).view(-1)
        # Calculate loss on all-fake batch
        errD_fake = criterion(output, label)
        # Calculate gradients for G in backward pass
        errD_fake.backward()
        G_z = output.mean().item()
        G_z = netG(data[1].to(device)).get_device()

        # Save losses for plotting later
        G_losses.append(errG.item())
        D_losses.append(errD.item())

        # Check how the generator is doing by saving G's output on fixed_noise
        if (iters % 50 == 0) or ((epoch == num_epochs - 1) and (i == len(dataloader) - 1)):
            with torch.no_grad():
                fake = netG(fixed_noise).detach().cpu()
            img_list.append(utils.make_grid(fake, padding=2, normalize=True))

            iters += 1

    # Starting Training Loop...
    /usr/local/lib/python3.7/dist-packages/torch/autograd/variable.py:190: UserWarning: This DataLoader will create 4 worker
    queues (checked)
    (0/2) [0/41] Loss_D: 2.0879 Loss_G: 3.3653 D(x): 0.3770 D(G(z)): 0.5501 / 0.0520
```



```
if i % 10 == 0:
    print(f'[{epoch}/{num_epochs}] Loss_D: {D_loss:.4f} Loss_G: {G_loss:.4f} D(x): {D_x:.4f} D(G(z)): {D_z:.4f} / {D_z:.4f}')
    (epoch, num_epochs, i, len(dataloader),
    errD.item(), errG.item(), D_x, D_G_z1, D_G_z2)

# Save losses for plotting later
G_losses.append(errG.item())
D_losses.append(errD.item())

# Check how the generator is doing by saving G's output on fixed_noise
if (iters % 50 == 0) or ((epoch == num_epochs - 1) and (i == len(dataloader) - 1)):
    with torch.no_grad():
        fake = netG(fixed_noise).detach().cpu()
    img_list.append(utils.make_grid(fake, padding=2, normalize=True))

    iters += 1

Starting Training Loop...
/usr/local/lib/python3.7/dist-packages/torch/autograd/variable.py:190: UserWarning: This DataLoader will create 4 worker
queues (checked)
(0/2) [0/41] Loss_D: 2.0879 Loss_G: 3.3653 D(x): 0.3770 D(G(z)): 0.5501 / 0.0520
```



```

G_losses.append(errG.item())
D_losses.append(errD.item())

# Check how the generator is doing by saving G's output on fixed_noise
if (iters % 50 == 0) or ((epoch == num_epochs-1) and (i == len(data_loader)-1)):
    with torch.no_grad():
        fake = netG(fixed_noise).detach().cpu()
        img_list.append(vutils.make_grid(fake, padding=2, normalize=True))

    iters += 1

--- Starting Training Loop...
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:490: UserWarning: This DataLoader will create 4 worker
processes (checked)
[0/5][0/41] Loss_D: 2.0879 Loss_G: 3.3653 D(x): 0.3770 D(G(z)): 0.5501 / 0.0520
[0/5][10/41] Loss_D: 1.5543 Loss_G: 13.6661 D(x): 0.8025 D(G(z)): 0.6351 / 0.0000
[0/5][20/41] Loss_D: 0.9879 Loss_G: 11.2471 D(x): 0.7037 D(G(z)): 0.1853 / 0.0000
[0/5][30/41] Loss_D: 0.4930 Loss_G: 12.7184 D(x): 0.3494 D(G(z)): 0.3995 / 0.0000
[0/5][40/41] Loss_D: 3.3006 Loss_G: 7.0687 D(x): 0.6459 D(G(z)): 0.9150 / 0.0049
[1/5][0/41] Loss_D: 2.4812 Loss_G: 3.2583 D(x): 0.2484 D(G(z)): 0.3105 / 0.0834
[1/5][10/41] Loss_D: 2.2799 Loss_G: 2.6442 D(x): 0.3611 D(G(z)): 0.5151 / 0.0915
[1/5][20/41] Loss_D: 2.4812 Loss_G: 2.8048 D(x): 0.3044 D(G(z)): 0.5691 / 0.0884
[1/5][30/41] Loss_D: 2.7775 Loss_G: 2.5564 D(x): 0.2579 D(G(z)): 0.5939 / 0.1097
[1/5][40/41] Loss_D: 2.3683 Loss_G: 2.0795 D(x): 0.3285 D(G(z)): 0.5604 / 0.1761
[2/5][0/41] Loss_D: 2.4184 Loss_G: 2.7372 D(x): 0.3965 D(G(z)): 0.6679 / 0.1067
[2/5][10/41] Loss_D: 2.4835 Loss_G: 2.0629 D(x): 0.3807 D(G(z)): 0.6201 / 0.1593
[2/5][20/41] Loss_D: 2.4592 Loss_G: 2.7858 D(x): 0.4465 D(G(z)): 0.7141 / 0.0844
[2/5][30/41] Loss_D: 1.7367 Loss_G: 1.5997 D(x): 0.4188 D(G(z)): 0.4708 / 0.2322
[2/5][40/41] Loss_D: 2.0760 Loss_G: 2.0569 D(x): 0.4271 D(G(z)): 0.6124 / 0.1722
[3/5][0/41] Loss_D: 1.9626 Loss_G: 2.4509 D(x): 0.4614 D(G(z)): 0.5957 / 0.1307
[3/5][10/41] Loss_D: 1.8413 Loss_G: 2.1124 D(x): 0.4787 D(G(z)): 0.5657 / 0.1453
[3/5][20/41] Loss_D: 2.4000 Loss_G: 1.4404 D(x): 0.3090 D(G(z)): 0.5742 / 0.2784
[3/5][30/41] Loss_D: 1.6932 Loss_G: 1.6974 D(x): 0.4264 D(G(z)): 0.4845 / 0.2168
[3/5][40/41]

```

Now, let us run this value. So, let us run this training loop, let us see how the training continues. See here. This is the first step out of 5 epochs and inside that 5 epochs, sorry, inside that first epoch only, we are observing the first iteration out of 41 iteration see here.

This is the 10th iteration. These are the values of loss values, discriminator loss values and the generator. By observing these loss values we just have to observe these values and make a decision, 4th epoch is going now, here given number of epoch is 5 epochs only. Now, the training has been done.

(Refer Slide Time: 68:13)

```

[3/5][20/41] Loss_D: 2.4000 Loss_G: 1.4404 D(x): 0.3090 D(G(z)): 0.5742 / 0.2784
[3/5][30/41] Loss_D: 1.6932 Loss_G: 1.6974 D(x): 0.4264 D(G(z)): 0.4845 / 0.2168
[3/5][40/41] Loss_D: 1.9234 Loss_G: 2.6491 D(x): 0.5491 D(G(z)): 0.4709 / 0.0958
[4/5][0/41] Loss_D: 2.3091 Loss_G: 0.5790 D(x): 0.2135 D(G(z)): 0.2863 / 0.5912
[4/5][10/41] Loss_D: 1.7113 Loss_G: 1.2059 D(x): 0.3974 D(G(z)): 0.4220 / 0.3167
[4/5][20/41] Loss_D: 1.5496 Loss_G: 1.5779 D(x): 0.5092 D(G(z)): 0.5102 / 0.2471
[4/5][30/41] Loss_D: 1.8272 Loss_G: 1.5273 D(x): 0.4661 D(G(z)): 0.5748 / 0.2616
[4/5][40/41] Loss_D: 1.6057 Loss_G: 1.3839 D(x): 0.4912 D(G(z)): 0.5313 / 0.2872

--- Results

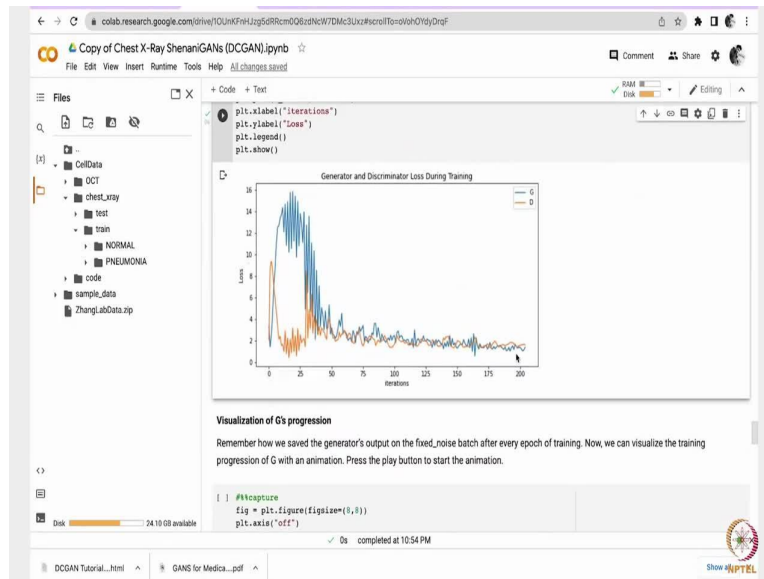
Finally, lets check out how we did. Here, we will look at three different results. First, we will see how D and G's losses changed during training.
Second, we will visualize G's output on the fixed_noise batch for every epoch. And third, we will look at a batch of real data next to a batch of
fake data from G.

Loss versus training iteration

Below is a plot of D & G's losses versus training iterations.

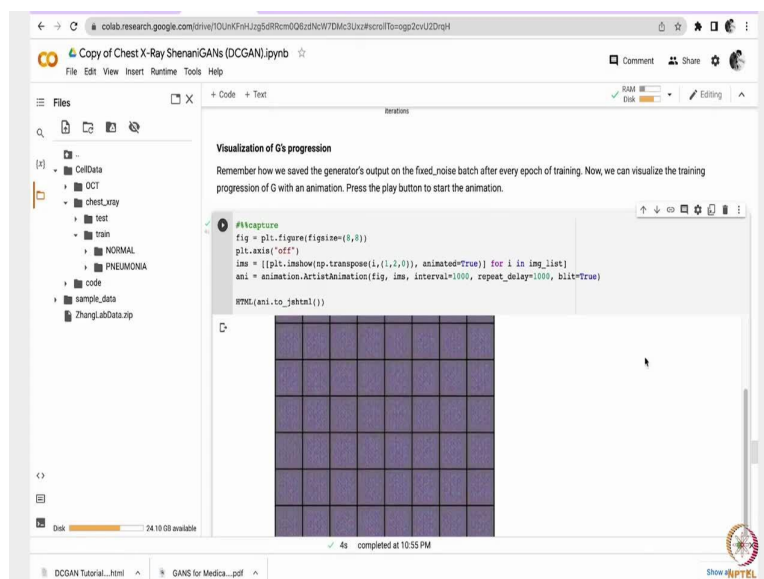
plt.figure(figsize=(10,5))
plt.title("Discriminator and Generator Loss During Training")
plt.plot(G_losses,label="G")
plt.plot(D_losses,label="D")
plt.xlabel("iterations")
plt.ylabel("Loss")
plt.legend()
plt.show()

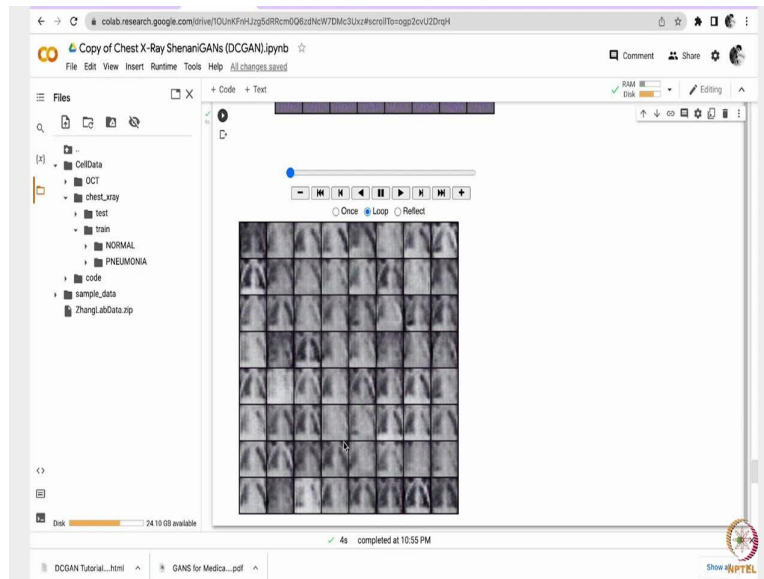
```



Now, let us see how the loss function, loss cause will be, see here both the losses are going down. But here you can see a sudden jump of generator from the discriminator loss value. But as the days pass by as the number of epochs increasing since I have used it very very less number of epochs you can go and change that to later on you can play around let us see.

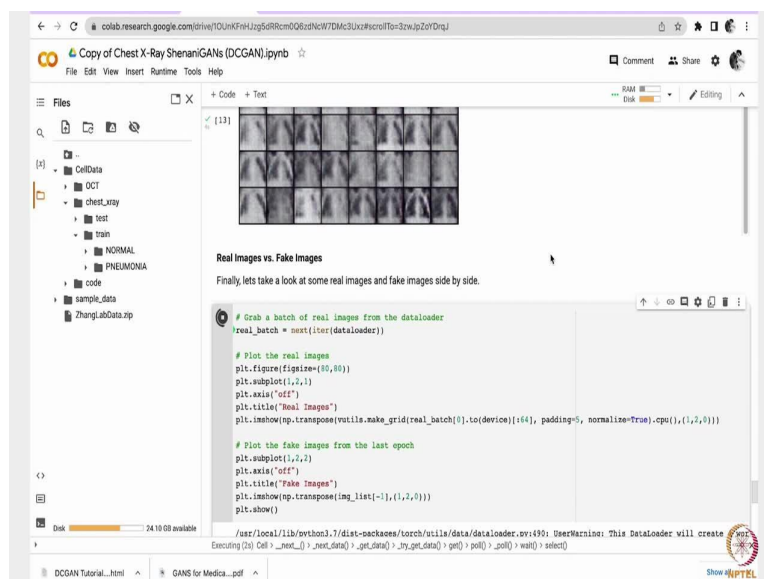
(Refer Slide Time: 68:44)

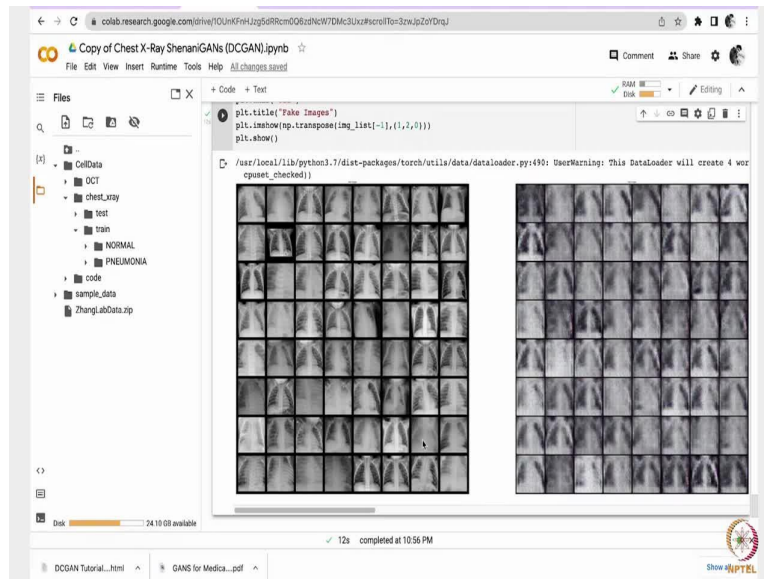




This progression I mean here in each and every loop we have the noise vectors it is corresponding, see here you can see the images are similar to chest X ray images whatever, but not almost close to them, but you can see those replications somehow you can identify the.

(Refer Slide Time: 69:08)

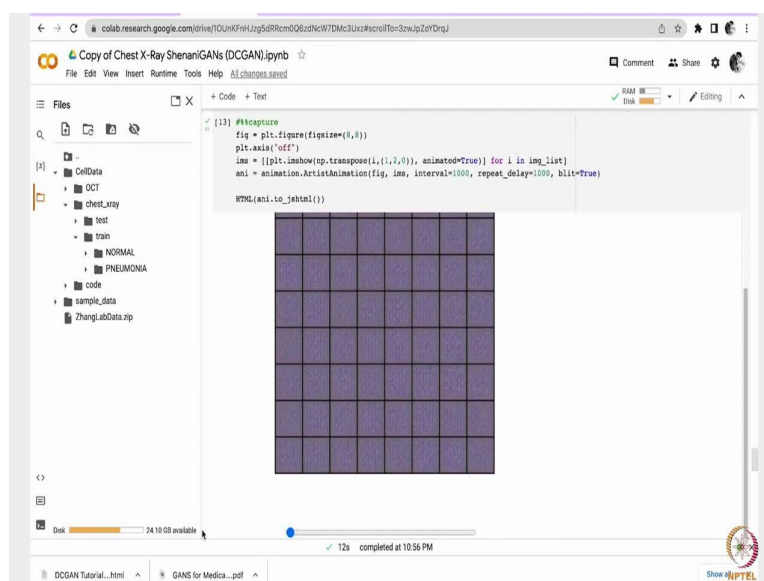




Let us see first 64 images were exactly the output has been taken out, see here these are all left side you can see all the real images whereas the right side you have only fake images. So, now the entire task is done.

Now, our turn is to modify hyper parameters. So, that you should get this distribution or this chest X ray images almost similar to these values. Then we can say that generator has been trained very well and we can save that particular model. You can now randomly give a random noise vector with a particular generator saved model and you will definitely get a chest X ray image out of it. So, the entire problem has been done.

(Refer Slide Time: 70:20)



Now, as a summary note I can say that it is completely for any model hyper parameter tuning is very important that is where your skills matter. Now, coming back to suppose if you are, you should have using chest X ray data, you may have some other data and keep the directories like this inside the chest X ray go to train data inside this suppose you have some N number of classes, keep all the N number of classes here and this is how the directory should be present. So, I can see that the task has been done, thank you.