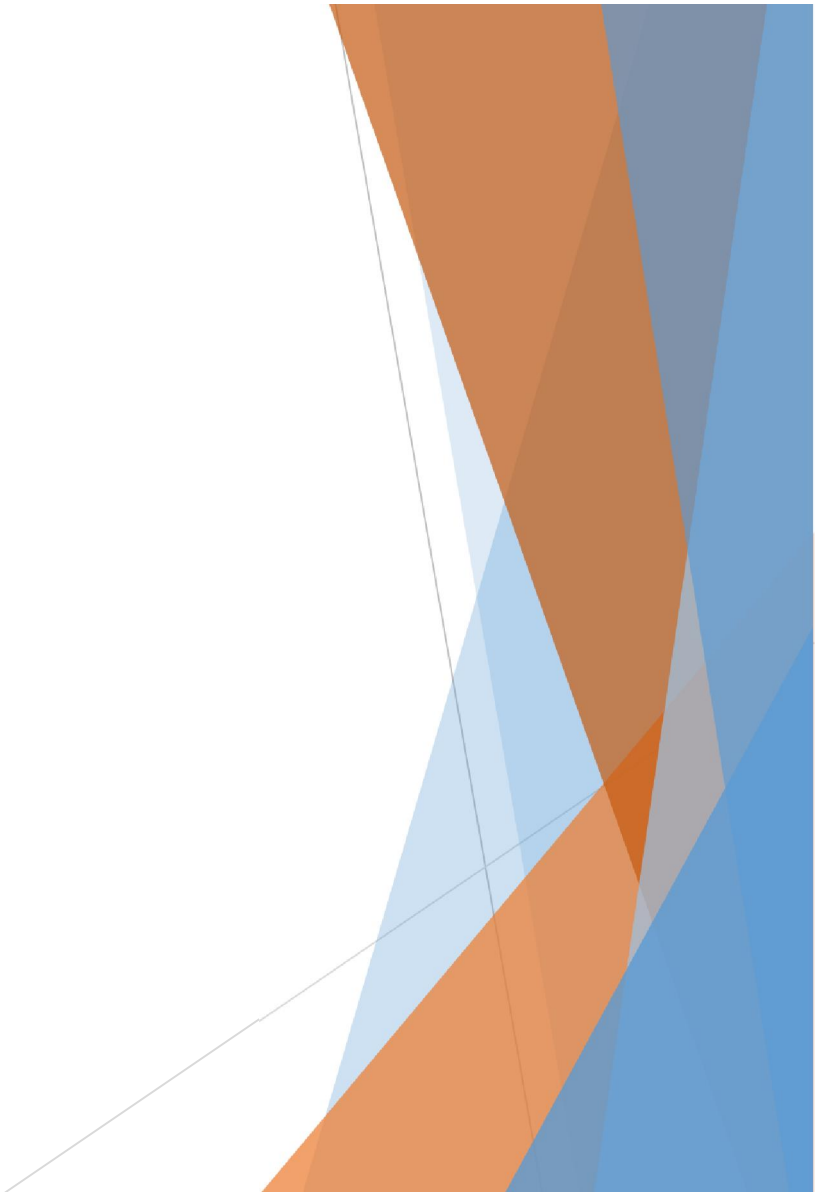
Deep Learning

Generative Adversarial Networks (GANs)

Dr. Ahmar Rashid, FCSE, GIKI

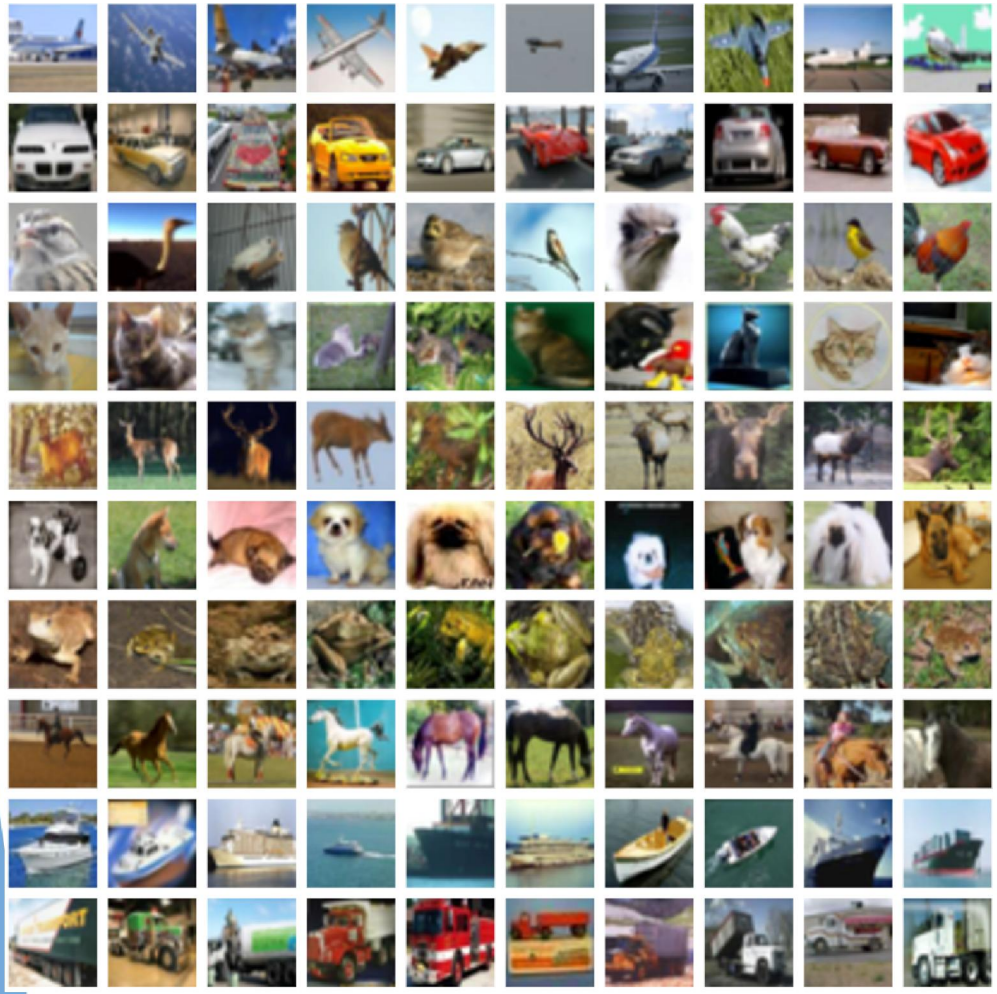
Contents

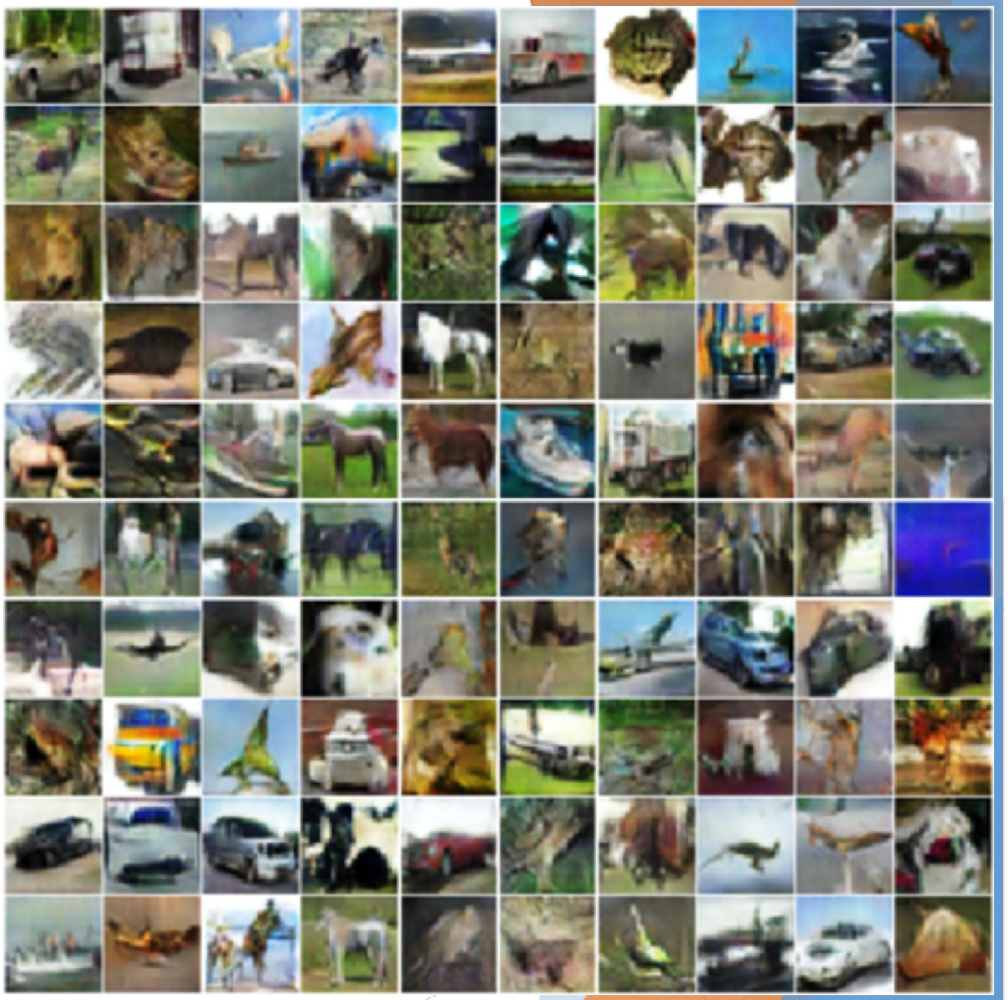


What are GANs

- **Definition**
  - Generative Adversarial Networks (GANs) are a class of machine learning models designed to generate new data samples that resemble a given dataset.
- Introduced by Ian Goodfellow and his colleagues in 2014.
- GANs are part of the broader field of generative modeling.

Generated bedrooms. Source: "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks" https://arxiv.org/abs/1511.06434v2



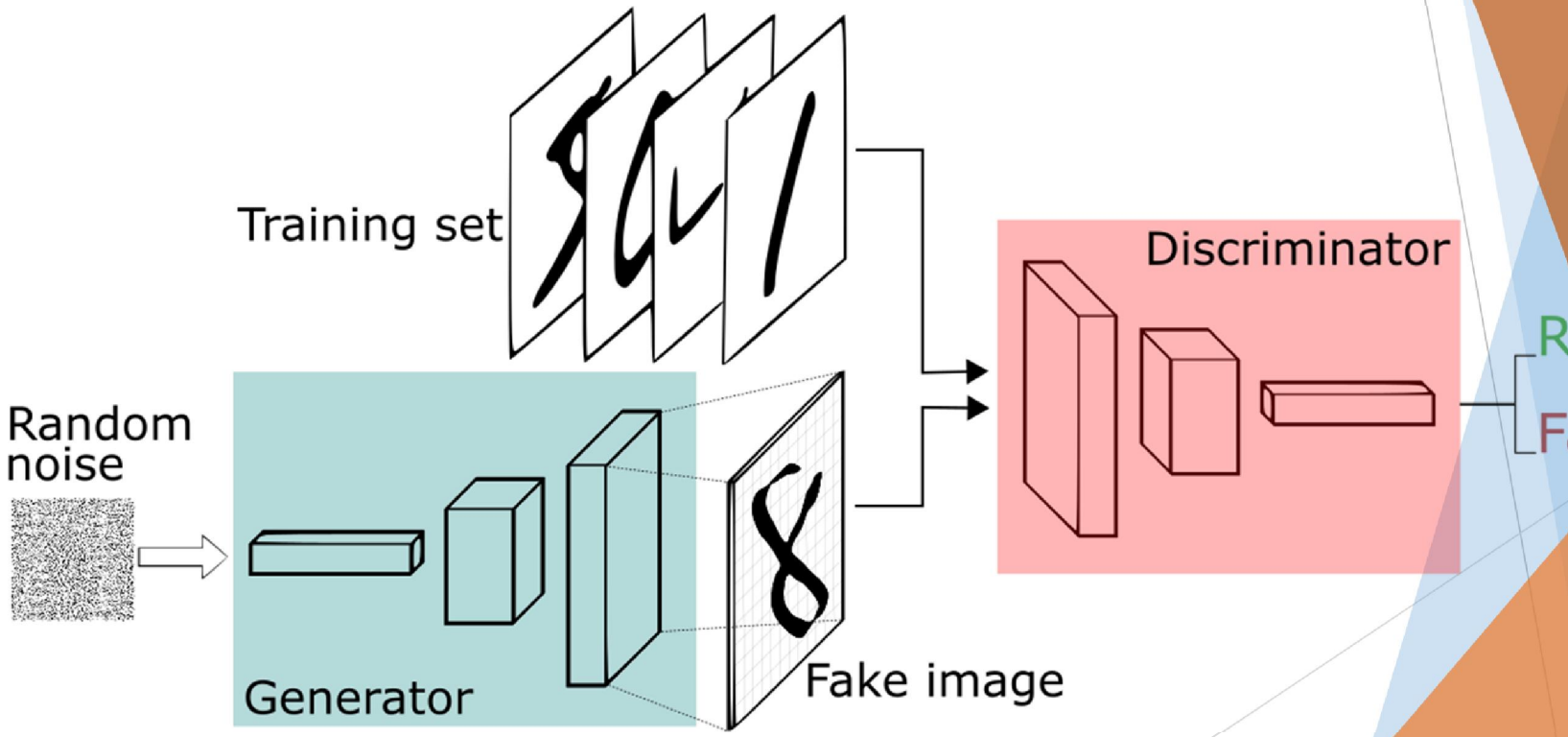


Original CIFAR-10 vs. Generated CIFAR-10 samples
Source: "Improved Techniques for Training GANs" https://arxiv.org/abs/1606.03498

How do GANs Work?

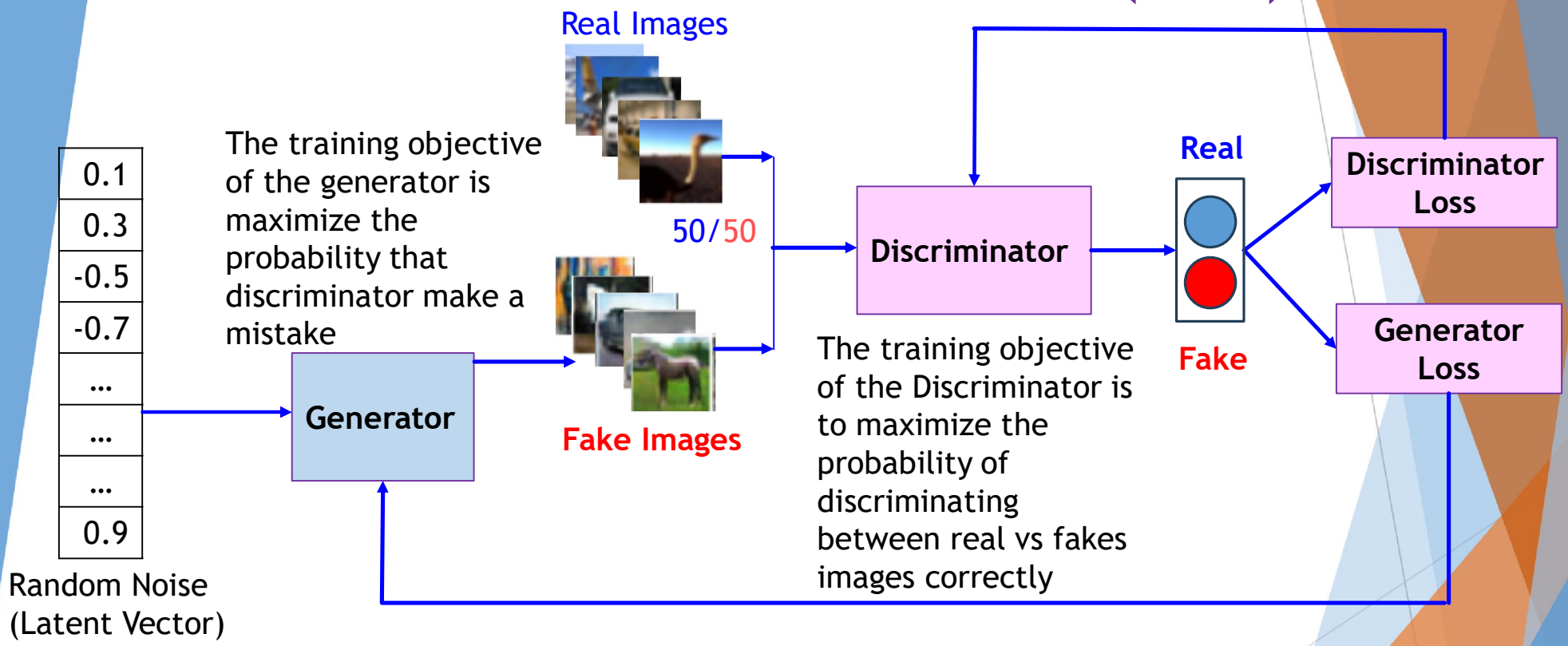
- **GAN Architecture:**
  - Generator (G): $G(z)$ where z is random noise.
  - Discriminator(D): $D(x)$ where x is real or generated data.
- **Objective:**
  - Minimize $\mathcal{L}_D$ and $\mathcal{L}_G$ where $\mathcal{L}_D$ is the discriminator loss and $\mathcal{L}_G$ is the generator loss.

Generative Adversarial Network (GAN)



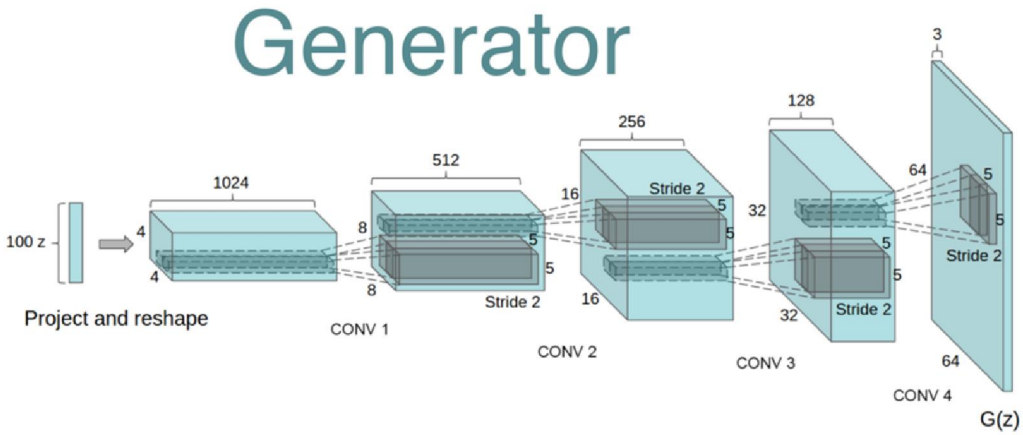
Source: Zhifei Zhang , University of Tennessee, Knoxville

Generative Adversarial Network (GAN)



Source: DigitalSreeni Youtube Channel

Generative Adversarial Network (GAN)

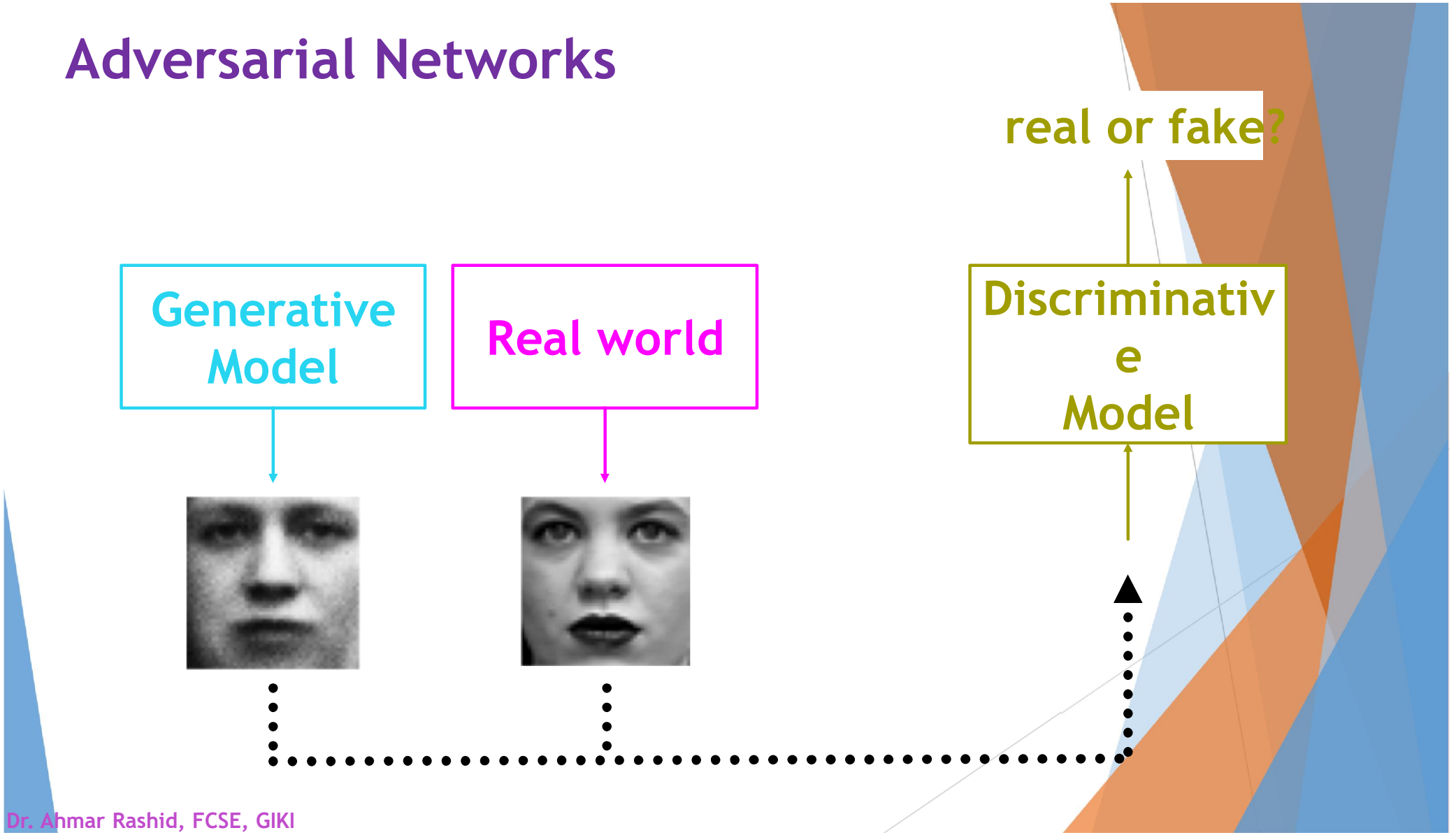


New components:
Transposed convolution,
Batch Normalization

Binary Classifier:
Conv, Leaky ReLU,
FC, Sigmoid

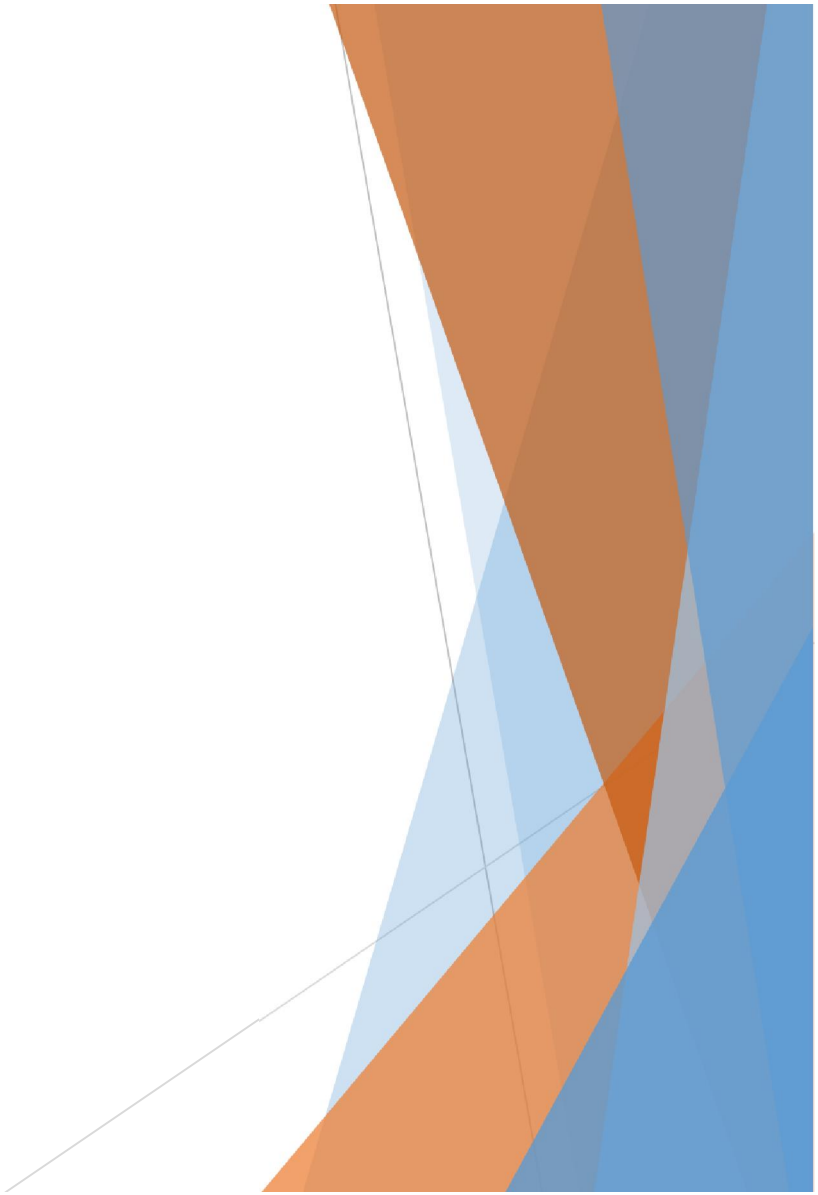
https://github.com/PramodShenoy/GANerations

Adversarial Networks



Components of a GANs

- **Generator:**
  - Input: Random noise (z).
  - Output: Generates synthetic data ($G(z)$).
- **Discriminator:**
  - Input: Real or generated data (x).
  - Output: Classifies input as real or fake ($D(x)$).



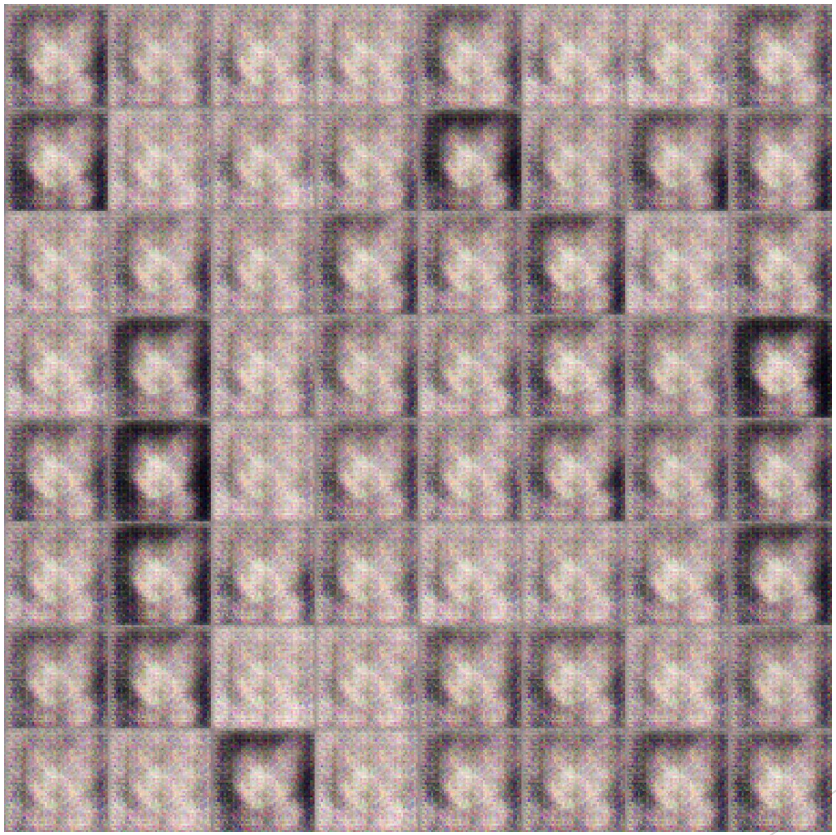
Generating fake figures

You can use the following to start a project (but this is in Chinese):

Source of images: https://zhuanlan.zhihu.com/p/24767059
From Dr. HY Lee's notes.

DCGAN: https://github.com/carpedm20/DCGAN-tensorflow

GAN - generating 2nd element figuresb



100 rounds

This is fast, I think you can use your CPU

GAN – generating 2nd element figures

1000 rounds

GAN – generating 2nd element figures

2000 rounds

GAN – generating 2nd element figures

5000 rounds

GAN – generating 2nd element figures

10,000 rounds

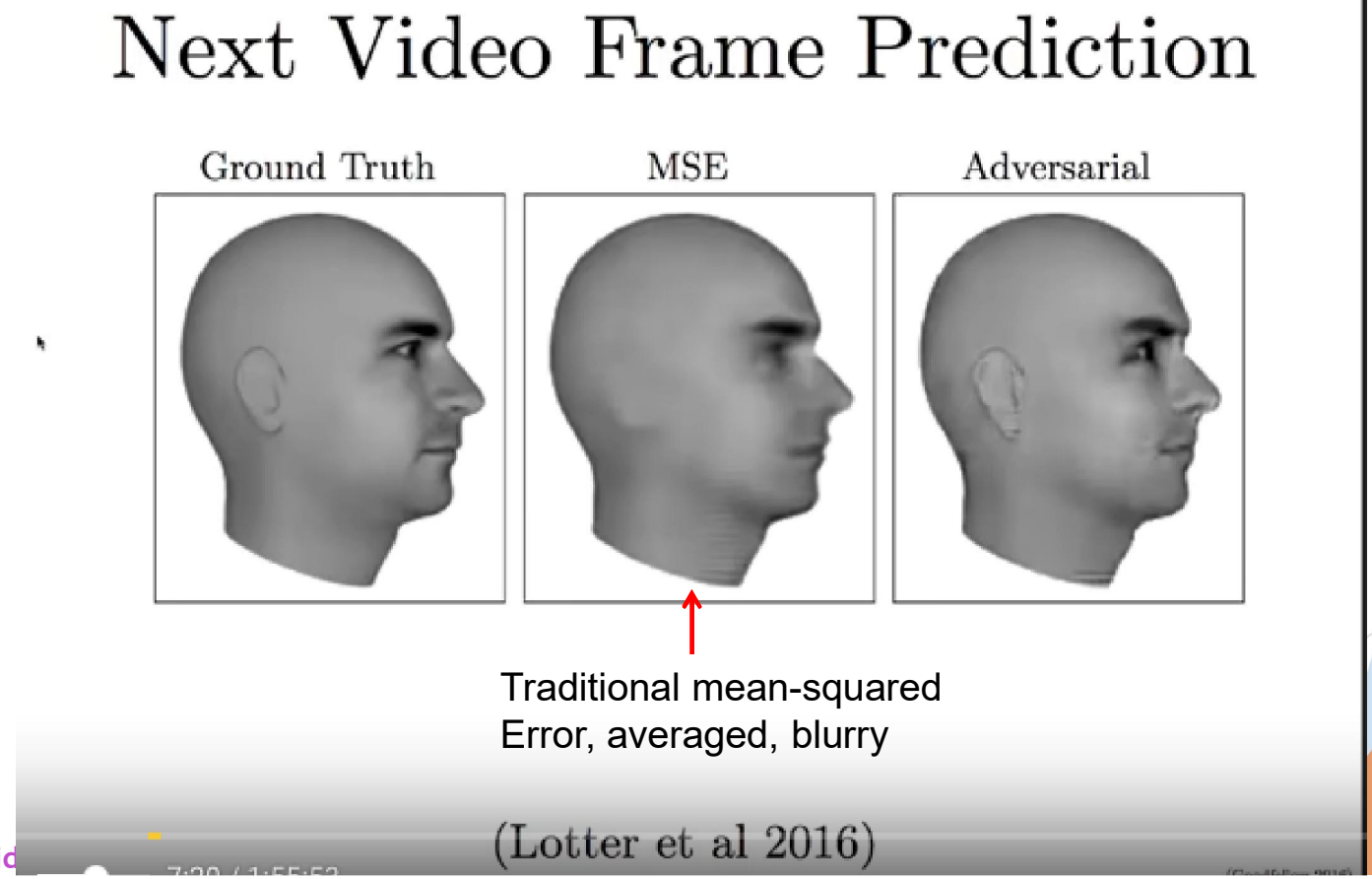
GAN – generating 2nd element figures

20,000 rounds

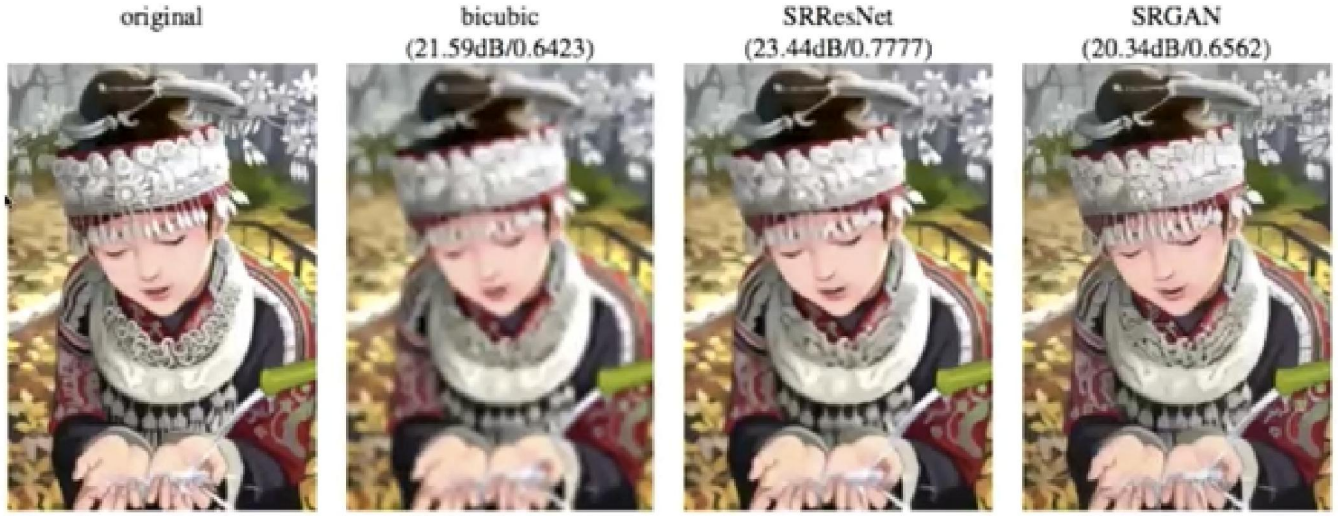
GAN – generating 2nd element figures

50,000 rounds

Next few images from Goodfellow lecture



Single Image Super-Resolution

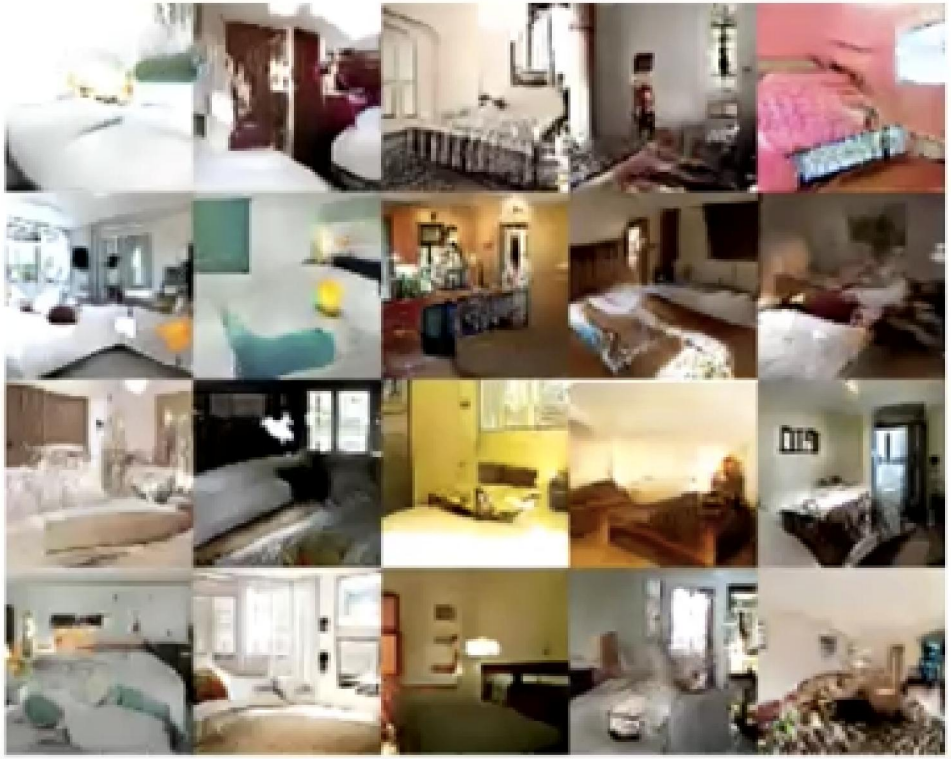


(Ledig et al 2016)

Last 2 are by deep learning approaches.

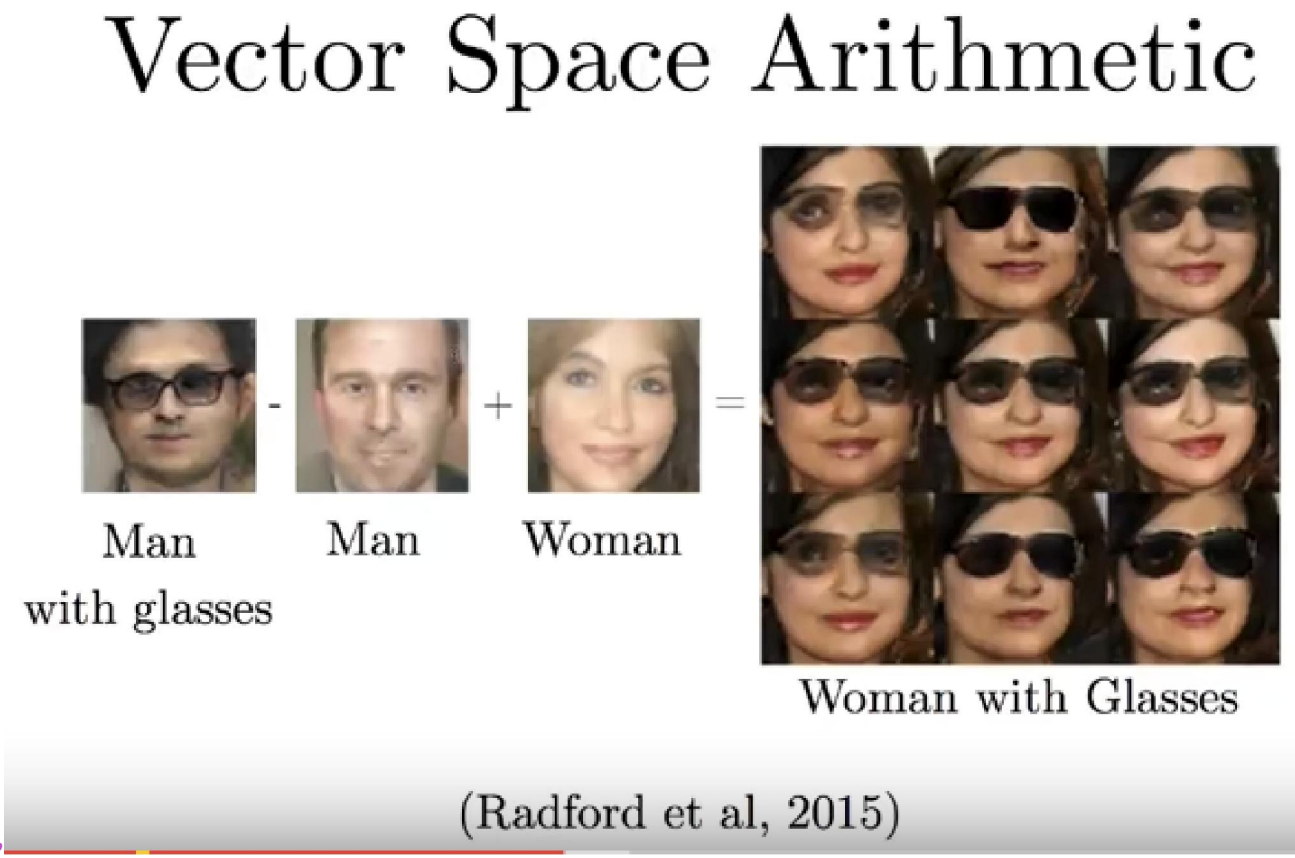
Image to Image Translation

DCGANs for LSUN Bedrooms



(Radford et al 2015)

Similar to word embedding (DCGAN paper)

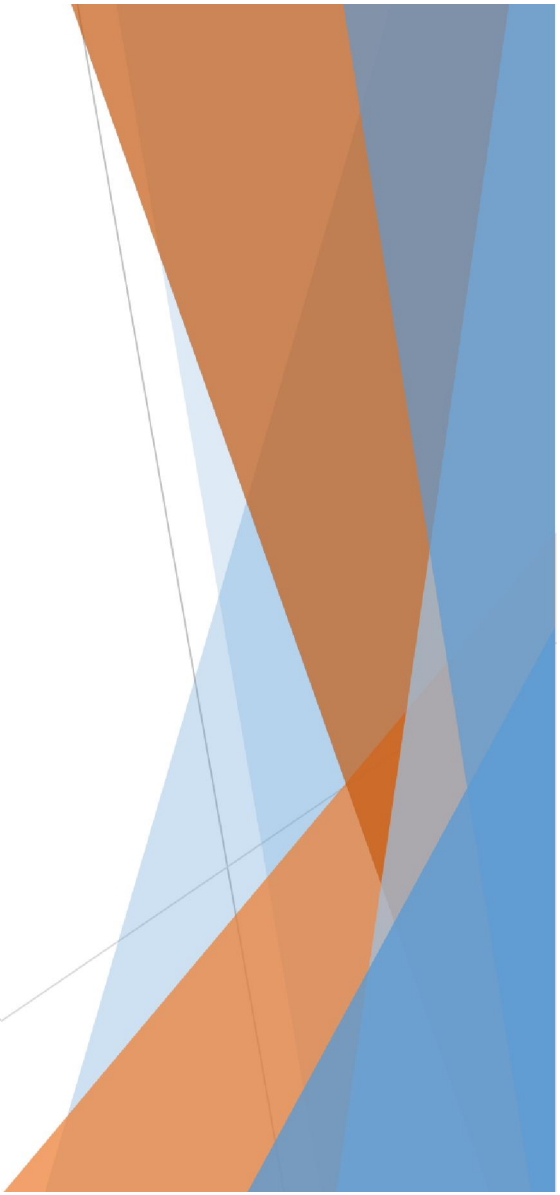


Training GANs

- Iterative Process:
  - Generator generates samples $G(z)$).
  - Discriminator evaluates and provides feedback.
  - Adjustments made to both generator and discriminator.
  - Repeat until convergence.
- Adversarial Loss Function:
  - $\mathcal{L}_D = \frac{1}{2}\mathbb{E}_{x\sim p_{\text{data}}}[-\ln D(x)] + \frac{1}{2}\mathbb{E}_{z\sim p_z}[-\ln\left(1 - D\left(G(z)\right)\right)]$
  - $\mathcal{L}_G = -\mathcal{L}_D = \frac{1}{2}\mathbb{E}_{x\sim p_{\text{data}}}[\ln D(x)] + \frac{1}{2}\mathbb{E}_{z\sim p_z}[\ln\left(1 - D\left(G(z)\right)\right)]$
  - $\mathcal{L}_G = \frac{1}{2}\mathbb{E}_{z\sim p_z}[\ln\left(1 - D\left(G(z)\right)\right)] = \frac{1}{2}\mathbb{E}_{z\sim p_z}[-\ln D\left(G(z)\right)]$
  - $min_G, maxD(\mathcal{L}_D) = \frac{1}{2}\mathbb{E}_{x\sim p_{\text{data}}}[\ln D(x)] + \frac{1}{2}\mathbb{E}_{z\sim p_z}[\ln\left(1 - D\left(G(z)\right)\right)]$

Logistic Regression

Cost function

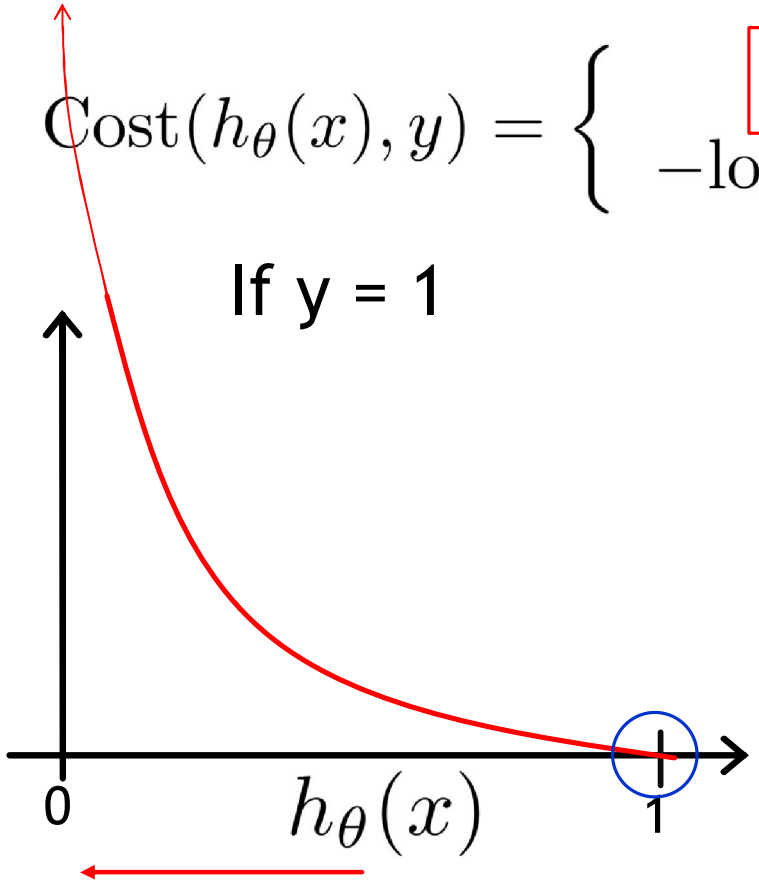


Dr. Ahmar Rashid, FCSE, GIKI

Logistic regression cost function

$$\text{Cost}(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{if } y = 1 \\ -\log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$$

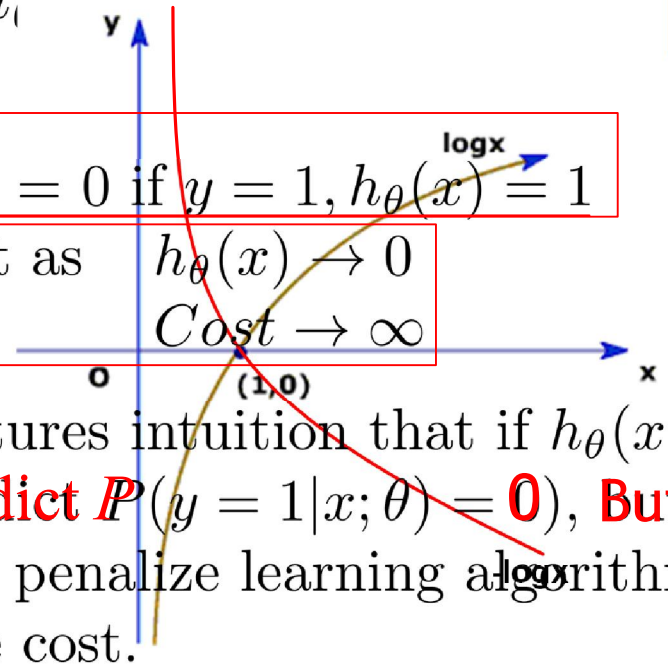
If y = 1



$\text{Cost} = 0$ if $y = 1, h_\theta(x) = 1$

But as $h_\theta(x) \to 0$

$Cost \to \infty$

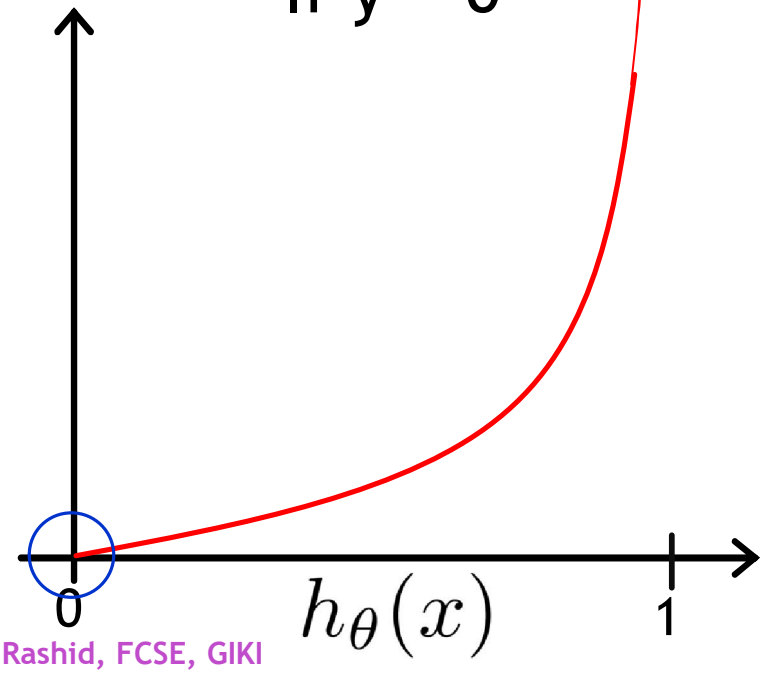


Captures intuition that if $h_\theta(x) = 0$, (predict $P(y = 1|x; \theta) = 0$), But $y = 1$, we'll penalize learning algorithm by a very large cost.

Logistic regression cost function

$$\text{Cost}(h_\theta(x), y) = \begin{cases} -\log(h_\theta(x)) & \text{if } y = 1 \\ -\log(1 - h_\theta(x)) & \text{if } y = 0 \end{cases}$$

If y = 0



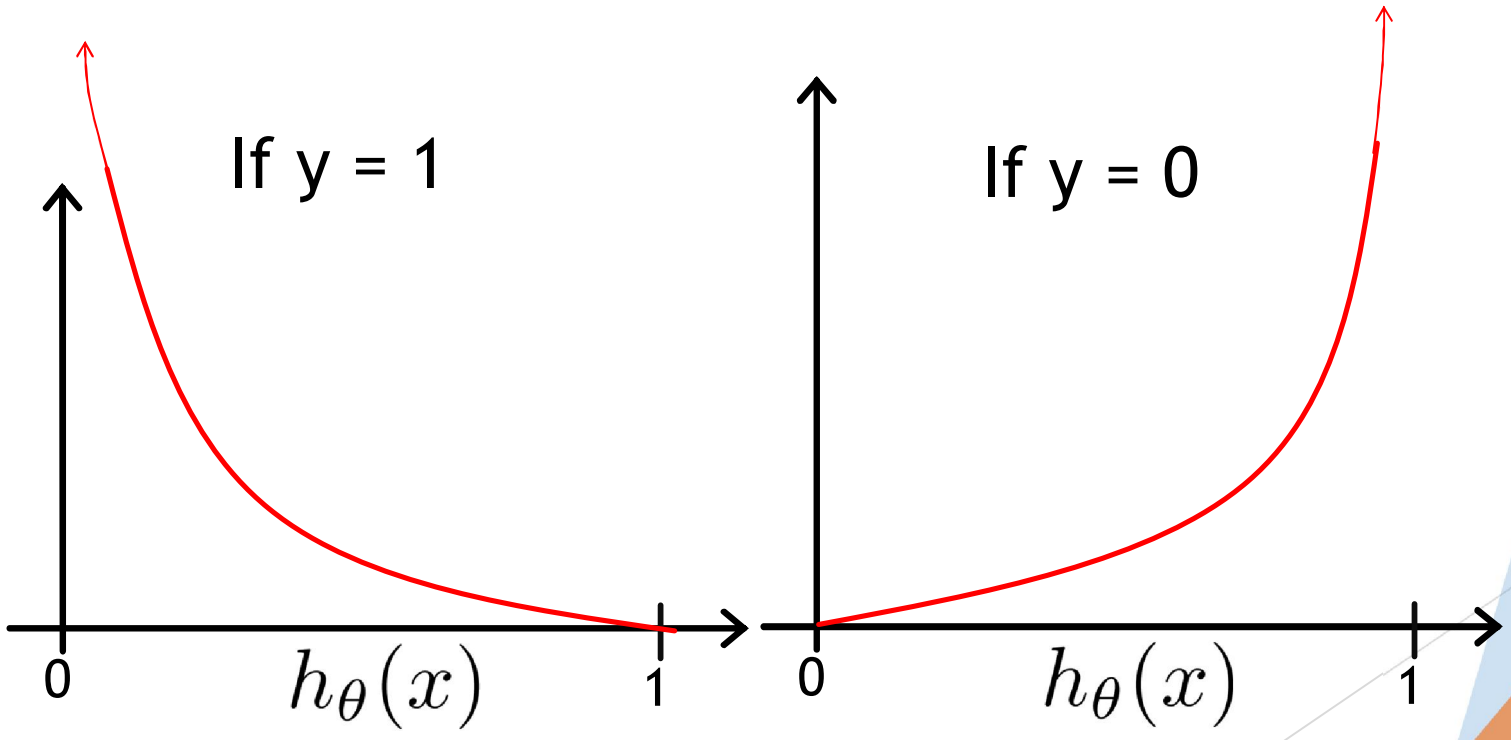
$\text{Cost} = 0$ if $y = 0$, $h_\theta(x) = 0$

But as $h_\theta(x) \to 1$

$Cost \to \infty$

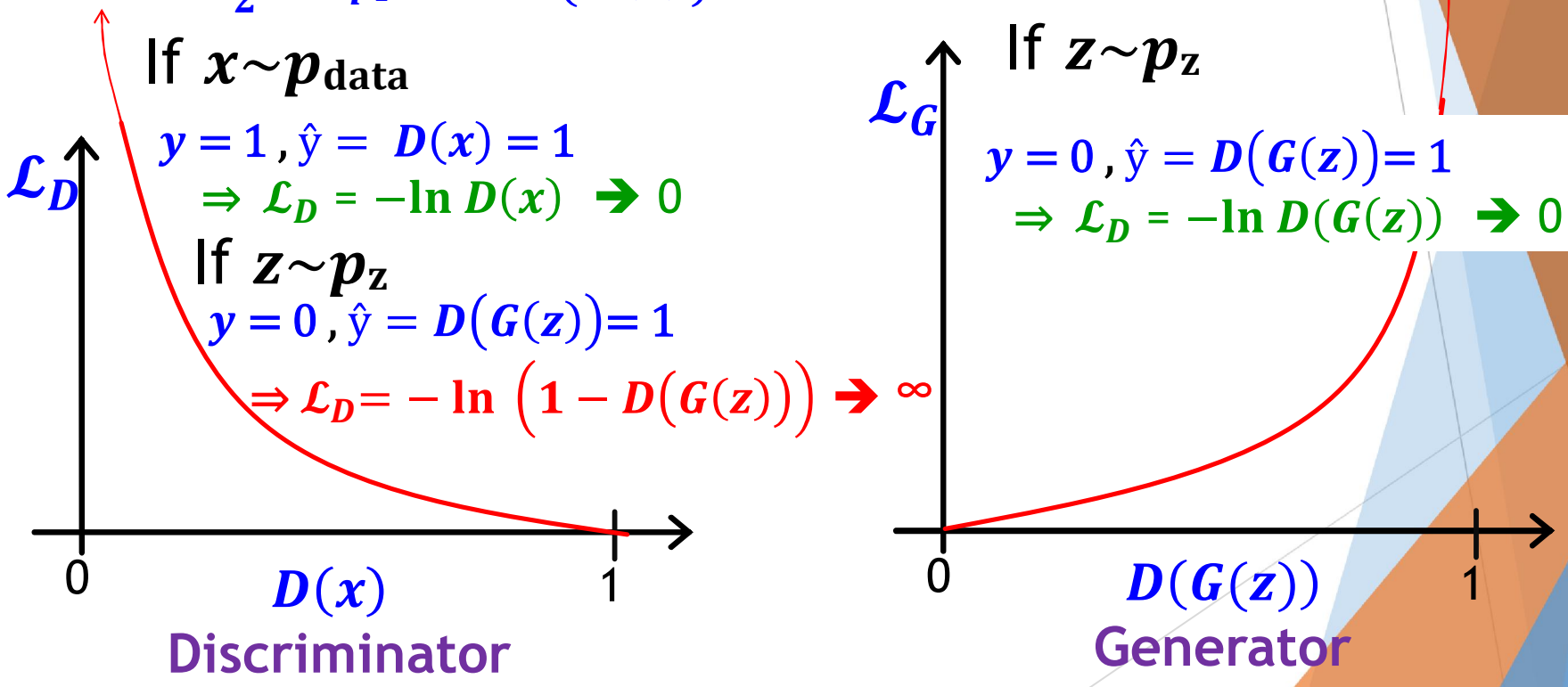
Logistic regression consolidated cost function

$$Cost(h_\theta(x), y) = -y \log(h_\theta(x)) - (1-y) \log(1- h_\theta(x))$$



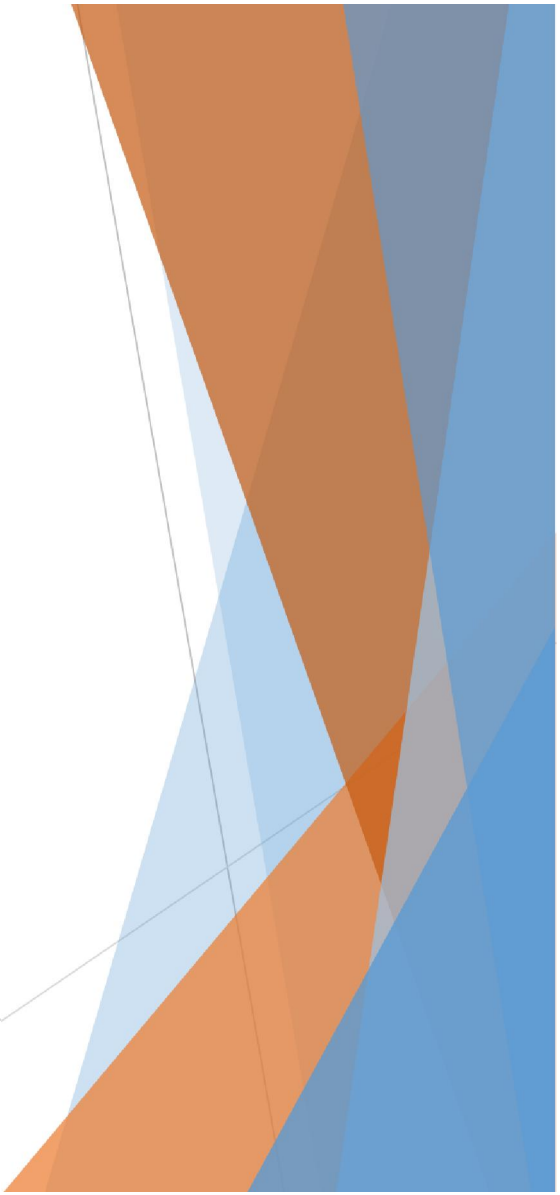
GAN consolidated cost function

- $\mathcal{L}_D = -\frac{1}{2}\mathbb{E}_{x\sim p_{\text{data}}} \ln D(x) - \frac{1}{2}\mathbb{E}_{z\sim p_z} \ln\left(1 - D(G(z))\right)$
- $\mathcal{L}_G = \frac{1}{2}\mathbb{E}_{z\sim p_z}[-\ln D(G(z))]$



GAN Applications

- **Image Generation:**
  - Visual: Comparison between real and generated images.
- **Data Augmentation:**
  - Visual: Original vs. augmented dataset.
- **Image-to-Image Translation:**
  - Visual: Satellite image to map conversion.
- **Super Resolution:**
  - Visual: Low resolution vs. high resolution image.

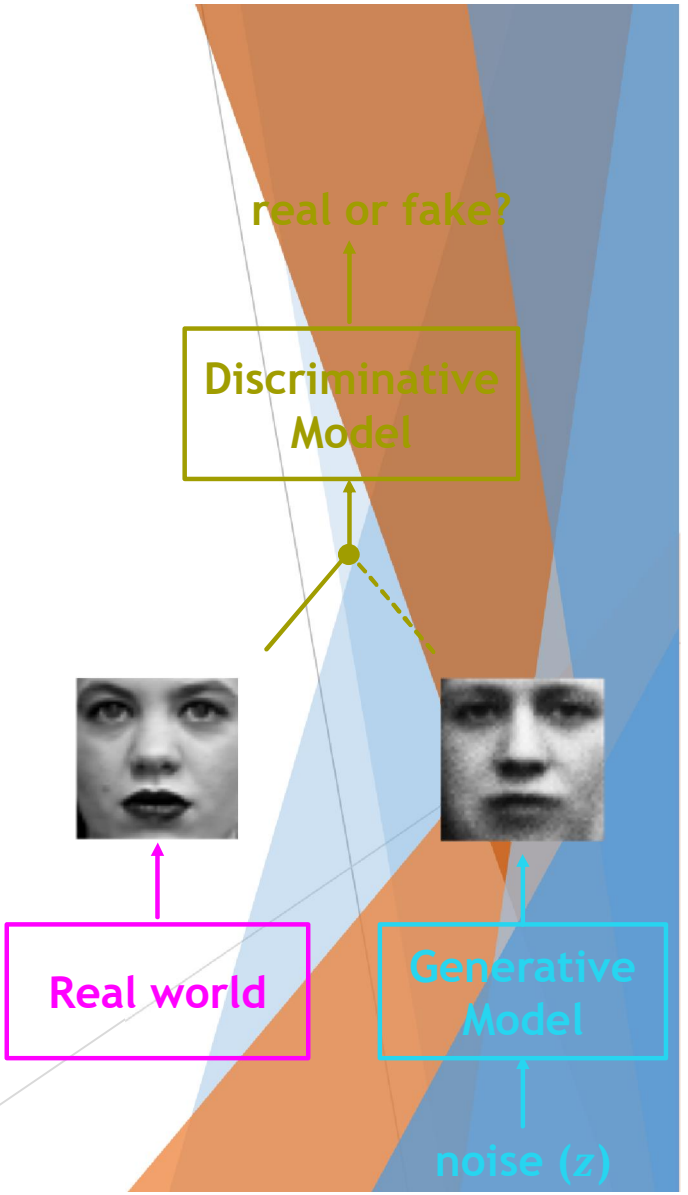


GAN Challenges and Future Directions

- **Training Stability:**
  - Visual: Illustration of mode collapse.
- **Evaluation Metrics:**
  - Visual: Difficulty in evaluating generated samples.
- **Ethical Considerations:**
  - Visual: Potential biases in generated data.
- **Future Directions:**
  - Visual: Concepts for improved GAN architectures.

Training Procedure

- Train both models simultaneously via stochastic gradient descent using minibatches consisting of
  - some generated samples
  - some real-world samples
- Training of D is straightforward
- Error for G comes via back propagation through D
  - Two ways to think about training
    - (1) freeze D weights and propagate $\mathcal{L}_G$ through D to determine $\partial \mathcal{L}_G / \partial x$
    - (2) Compute $\partial \mathcal{L}_D / \partial x$ and then $\partial \mathcal{L}_G / \partial x = -\partial \mathcal{L}_D / \partial x$
- D can be trained without altering G, and vice versa
  - May want multiple training epochs of just D so it can stay ahead
  - May want multiple training epochs of just G because it has a harder task



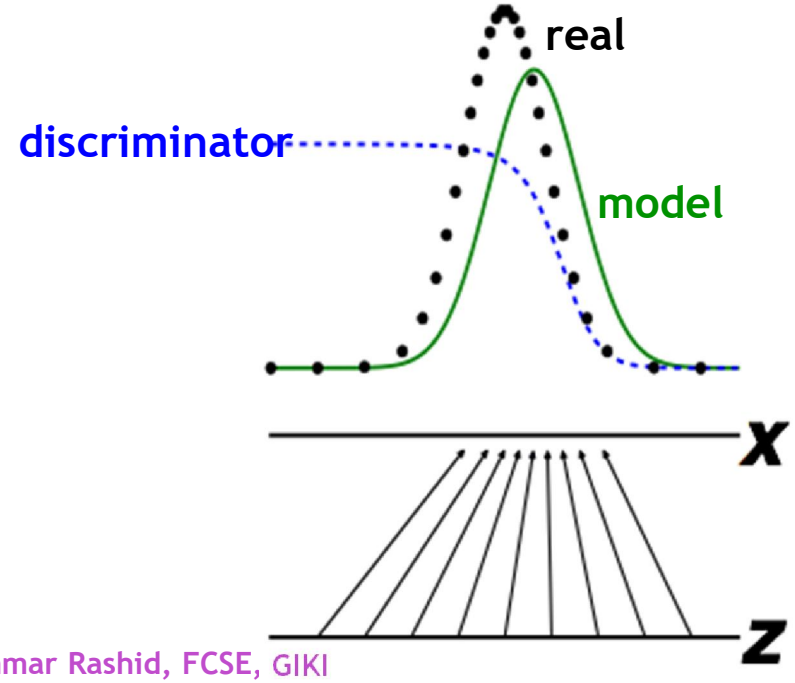
Training Procedure

- **for** each training iteration **do**
  - **for** k steps **do**
    - Sample ***m*** noise samples $\{z_1, \ldots, z_m\}$ and transform with Generator
    - Sample ***m*** real samples $\{x_1, \ldots, x_m\}$ from real data
    - Update the Discriminator by ascending the gradient.
    - $\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\ln D(x^{(i)}) + \ln \left(1 - D\left(G(z^{(i)})\right) \right) \right]$
  - end **for**
  - Sample ***m*** noise samples $\{z_1, \ldots, z_m\}$ and transform with Generator
  - Update the Generator by descending the gradient:
    - $\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \left[\ln D(x^{(i)}) + \ln \left(1 - D\left(G(z^{(i)})\right) \right) \right]$
    - $\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \left[\ln \left(1 - D\left(G(z^{(i)})\right) \right) \right] \rightarrow \nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \left[-\ln \left(D\left(G(z^{(i)})\right) \right) \right]$

Understand the Math and Theory of GANs in ~ 10 mins
https://www.youtube.com/watch?v=J1aG12dLo4I

The Discriminator Has a Straightforward Task

- D has learned when
  - $$D(x) = Pr(\text{real}|x) = \frac{Pr(x|\text{real})}{Pr(x|\text{real}) + Pr(x|\text{synthesized})}$$



Goodfellow (2017)

The Math Behind Generative Adversarial Networks Clearly Explained!
https://www.youtube.com/watch?v=Gib_kiXgnvA

Coding GANs

https://www.geeksforgeeks.org/generative-adversarial-network-gan/

Importing Libraries

```
import os
import numpy as np
import matplotlib.pyplot as plt
from tqdm import tqdm
from numpy import zeros
from numpy import ones
from numpy.random import randn
from numpy.random import randint
from keras.datasets.cifar10 import load_data
from keras.optimizers import Adam
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers import Reshape
from keras.layers import Flatten
from keras.layers import Conv2D
from keras.layers import Conv2DTranspose, BatchNormalization
from keras.layers import LeakyReLU, ReLU
from keras.layers import Dropout
from keras.layers import Embedding
from keras.layers import Concatenate
import matplotlib.pyplot as plt
```


https://www.datacamp.com/tutorial/generative-adversarial-networks

Copy/Split the Dataset into Train & Test Folders

```
from google.colab import drive
    drive.mount('/content/drive')
from sklearn.model_selection import train_test_split
import os
import shutil

dataset_folder = '/content/drive/MyDrive/tea_sickness_subset'

# Get the list of image filenames in the dataset folder
image_files = [os.path.join(dataset_folder, file)
for file in os.listdir(dataset_folder)
if file.endswith('.jpg') or file.endswith('.png')]

#Define the ratio for splitting the dataset into training and
testing sets
test_size = 0.2

# Split the dataset into training and testing sets
train_files, test_files = train_test_split(image_files,
test_size=test_size, random_state=42)

# Print the number of images in each set
print("Number of images in training set:", len(train_files))
print("Number of images in testing set:", len(test_files))
```

```
# Optionally, you can move the files to separate training and
testing folders
train_folder = '/content/train_dataset'
test_folder = '/content/test_dataset'

# Create the training and testing folders if they don't exist
if not os.path.exists(train_folder):
    os.makedirs(train_folder)
if not os.path.exists(test_folder):
    os.makedirs(test_folder)

# Move training images to the train folder
for file in train_files:
    filename = os.path.basename(file)
    dest_path = os.path.join(train_folder, filename)
    shutil.copyfile(file, dest_path)

# Move testing images to the test folder
for file in test_files:
    filename = os.path.basename(file)
    dest_path = os.path.join(test_folder, filename)
    shutil.copyfile(file, dest_path)

print("Dataset split and saved into training and testing
folders.")
```


https://www.datacamp.com/tutorial/generative-adversarial-networks

Load the Split Dataset into x_train & y_train np Arrays

```python
from PIL import Image
import os
import numpy as np

# Define the paths to the training and testing folders
train_folder = '/content/train_dataset'
test_folder = '/content/test_dataset'

# Initialize lists to store the image data
x_train = []
x_test = []

# Resize function
def resize_image(img, size=(512, 512)):
    return img.resize(size)
```

```python
# Read images from the training folder
for filename in os.listdir(train_folder):
    img_path = os.path.join(train_folder, filename)
    img = Image.open(img_path)
    img_resized = resize_image(img)  # Resize the image to 256x256
    img_data = np.array(img_resized)  # Convert image to numpy array
    x_train.append(img_data)

# Read images from the testing folder
for filename in os.listdir(test_folder):
    img_path = os.path.join(test_folder, filename)
    img = Image.open(img_path)
    img_resized = resize_image(img)  # Resize the image to 256x256
    img_data = np.array(img_resized)  # Convert image to numpy array
    x_test.append(img_data)

# Convert lists to numpy arrays
x_train = np.array(x_train)
x_test = np.array(x_test)

# Print the shapes of the arrays
print("Shape of x_train:", x_train.shape)
print("Shape of x_test:", x_test.shape)
```

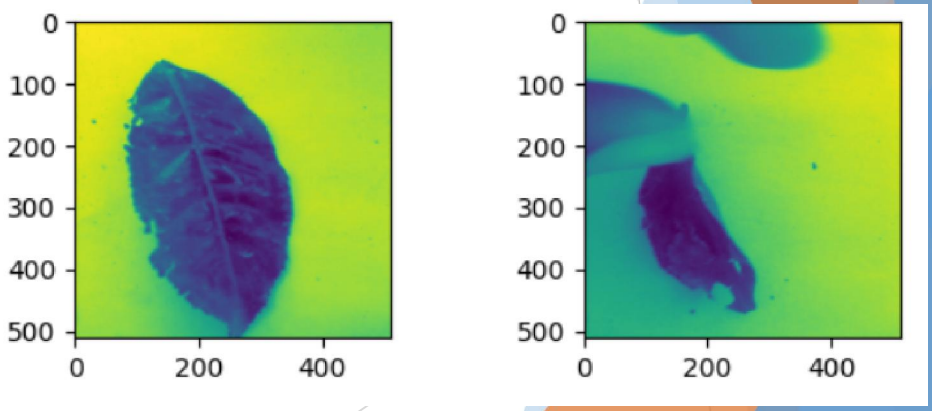

https://www.datacamp.com/tutorial/generative-adversarial-networks

Convert Data into Float and Normalize

```
x_train = np.array(x_train)
x_test = np.array(x_test)
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train,x_test=x_train/255.0,x_test/255.0
print(x_train.shape,x_test.shape)
img_width = x_train.shape[1]
img_height = x_train.shape[2]
num_channels = 3
x_train = x_train.reshape(x_train.shape[0], img_height, img_width, num_channels)
x_test = x_test.reshape(x_test.shape[0], img_height, img_width, num_channels)
# Print the shapes of the arrays
print("Shape of x_train:", x_train.shape)
print("Shape of x_test:", x_test.shape)
input_shape = (img_height, img_width, num_channels)
plt.figure(1)

plt.subplot(221)
plt.imshow(x_train[300][:,:,0])

plt.subplot(222)
plt.imshow(x_test[10][:,:,0])
```



https://www.datacamp.com/tutorial/generative-adversarial-networks

Us the Adam Optimizer

```
# You will use the Adam optimizer
def get_optimizer():
    return Adam(lr=0.0002, beta_1=0.5)
```


https://www.datacamp.com/tutorial/generative-adversarial-networks

Build up the Generator

```
def define_generator(latent_dim):
    # Image generator input
    in_lat = Input(shape=(latent_dim,))  # 
    # Foundation for 32x32 image
    n_nodes = 32 * 32 * 256
    gen = Dense(n_nodes)(in_lat)  # shape = 262,144
    gen = Reshape((32, 32, 256))(gen)  # Shape=32x32x256
    gen = BatchNormalization()(gen)
    gen = ReLU()(gen)
    # Upsample to 64x64x128
    gen = Conv2DTranspose(128, (4, 4), strides=(2, 2), padding='same')(gen)
    gen = BatchNormalization()(gen)
    gen = ReLU()(gen)
    # Upsample to 128x128x64
    gen = Conv2DTranspose(32, (4, 4), strides=(2, 2), padding='same')(gen)
    gen = BatchNormalization()(gen)
    gen = ReLU()(gen)
    # Upsample to 256x256x32
    gen = Conv2DTranspose(16, (4, 4), strides=(2, 2), padding='same')(gen)
    gen = BatchNormalization()(gen)
    gen = ReLU()(gen)
    # Output layer ➔ 256x256x3
    out_layer = Conv2D(3, (8, 8), activation='tanh', padding='same')(gen)
    # Define model
    model = Model(in_lat, out_layer)
    return model  # Model not compiled as it is not directly trained like the discriminator.
```

The input the first layer is a variable called **in_lat**, of dimension = latent_dim (e.g., 128)

- This **generator network** starts with **(01)** block featuring **dense layer**, followed by **Reshape(32, 32, 256)**, **BacthNormalizaton** and **ReLU** activation function

- unsampling blocks featuring **2D transpose convolution** layers.
- Each **Conv2DTranspose** layer is followed by a **BacthNormalizaton** and **ReLU** activation function
- Image is reshaped to **64x64x128**, **128x128x64** and **256x256x32** sizes, respectively by these blocks

image of size = **256x256x3**

Create a functional model for the generator, with **in_lat** as model input and **last layer output** as model output

https://www.geek

Build up the Generator

```
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 128)]             0

 dense_1 (Dense)             (None, 262144)            33816576

 reshape (Reshape)           (None, 32, 32, 256)       0

 batch_normalization (Batch  (None, 32, 32, 256)       1024
 Normalization)

 re_lu (ReLU)                (None, 32, 32, 256)       0

 conv2d_transpose (Conv2DTr  (None, 64, 64, 128)       524416
 anspose)

 batch_normalization_1 (Bat  (None, 64, 64, 128)       512
 chNormalization)

 re_lu_1 (ReLU)              (None, 64, 64, 128)       0

 conv2d_transpose_1 (Conv2D  (None, 128, 128, 64)      131136
 Transpose)

 batch_normalization_2 (Bat  (None, 128, 128, 64)      256
 chNormalization)

 re_lu_2 (ReLU)              (None, 128, 128, 64)      0

 conv2d_transpose_2 (Conv2D  (None, 256, 256, 32)      32800
 Transpose)

 batch_normalization_3 (Bat  (None, 256, 256, 32)      128
 chNormalization)

 re_lu_3 (ReLU)              (None, 256, 256, 32)      0

 conv2d_3 (Conv2D)           (None, 256, 256, 3)       6147

=================================================================
Total params: 34512995 (131.66 MB)
Trainable params: 34512035 (131.65 MB)
Non-trainable params: 960 (3.75 KB)
_________________________________________________________________
```

Build up the Discriminator

```
def define_discriminator(image_shape = (256,256,3)):
    # Input layer for real or generated image
    in_image = Input(shape=image_shape)
    # Downsample to 128x128x64
    d = Conv2D(64, (4, 4), strides=(2, 2), padding='same')(in_image)
    d = LeakyReLU(alpha=0.2)(d)
    d = Dropout(0.25)(d)
    # Downsample to 64x64x128
    d = Conv2D(128, (4, 4), strides=(2, 2), padding='same')(d)
    d = LeakyReLU(alpha=0.2)(d)
    d = Dropout(0.25)(d)
    # Downsample to 32x32x256
    d = Conv2D(256, (4, 4), strides=(2, 2), paddi
    d = LeakyReLU(alpha=0.2)(d)
    d = Dropout(0.25)(d)
    # Flatten and output a single classification
    d = Flatten()(d)
    out_layer = Dense(1, activation='sigmoid')(d)
    # Define model
    model = Model(in_image, out_layer)
    opt = Adam(lr=0.0002, beta_1=0.5)
    model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
    return model
```

The input dimension of the first layer is a is **256x256x3**, which is the same as the output dimension of the generator's last layer.

- The descriminator network features three **(03)** downsampling blocks featuring **2D convolution** layers.
- Each **Conv2D** layer is followed by **LeakyReLU** activation function

- Create a functional model for the descriminator, with **in_image** as model input and **last layer output** as model output
- The descriminator is compiled with **binary cross-entropy** loss and the optimizer passed as an argument.

https://www.geeksforgeeks.org/generative-adversarial-network-gan/

Build up the Discrimnator

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 256, 256, 3)]	0
conv2d (Conv2D)	(None, 128, 128, 64)	3136
leaky_re_lu (LeakyReLU)	(None, 128, 128, 64)	0
dropout (Dropout)	(None, 128, 128, 64)	0
conv2d_1 (Conv2D)	(None, 64, 64, 128)	131200
leaky_re_lu_1 (LeakyReLU)	(None, 64, 64, 128)	0
dropout_1 (Dropout)	(None, 64, 64, 128)	0
conv2d_2 (Conv2D)	(None, 32, 32, 256)	524544
leaky_re_lu_2 (LeakyReLU)	(None, 32, 32, 256)	0
dropout_2 (Dropout)	(None, 32, 32, 256)	0
flatten (Flatten)	(None, 262144)	0
dense (Dense)	(None, 1)	262145

Total params: 921025 (3.51 MB)
Trainable params: 921025 (3.51 MB)
Non-trainable params: 0 (0.00 Byte)

Building the Generative Adversarial Network

```
def define_gan(g_model, d_model):
    d_m                    Fals          s trained separately.
    # S                    nabl
    # c                    and discriminator...
    # Get noise and label inputs from generator model
    gen_noise = g_model.input
    # Get image output from the generator model
    gen_output = g_model.output  #256x256x3
    # generator image output is inputs to discriminator
    gan_output = d_model(gen_output)
    # Define a gan model as taking in noise, and outputti
    model = Model(gen_noise, gan_output)
    # Compile model
    opt = Adam(lr=0.0002, beta_1=0.5)
    model.compile(loss='binary_crossentropy', optimizer=opt)
    return model
```

A pre-trained generator model

Aa pre-trained discriminator model

Get noise and label inputs from generator model

Get image output (256x256x3)from the generator

generator image output is inputs to discriminator

Define a gan model as taking in noise, and outputting a classification

GAN is compiled with **binary cross-entropy** loss and the optimizer passed as an argument.

https://www.geeksforgeeks.org/generative-adversarial-network-gan/

Generate Real Samples

```
def generate_real_samples(dataset, n_samples):
  # split into images and labels
  images, labels = dataset
  # choose random instances
  ix = randint(0, images.shape[0], n_samples)
  # select images and labels
  X, labels = images[ix], labels[ix]
  # generate class labels and assign to y (don't confuse this with the above labels that
  # correspond to cifar labels)
  y = ones((n_samples, 1))  #Label=1 indicating they are real
  return [X, labels], y
```


https://www.geeksforgeeks.org/generative-adversarial-network-gan/

Generate points in latent space as input for the generator

```
def generate_latent_points(latent_dim, n_samples, n_classes=10):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  z_input = x_input.reshape(n_samples, latent_dim)
  # generate labels
  labels = randint(0, n_classes, n_samples)
  return [z_input, labels]
```

https://www.geeksforgeeks.org/generative-adversarial-network-gan/

Generate Fake Samples

```
def generate_fake_samples(generator, latent_dim, n_samples):
  # generate points in latent space
  z_input, labels_input = generate_latent_points(latent_dim, n_samples)
  # predict outputs
  images = generator.predict([z_input, labels_input])
  # create class labels
  y = zeros((n_samples, 1))  #Label=0 indicating they are fake
  return [images, labels_input], y
```

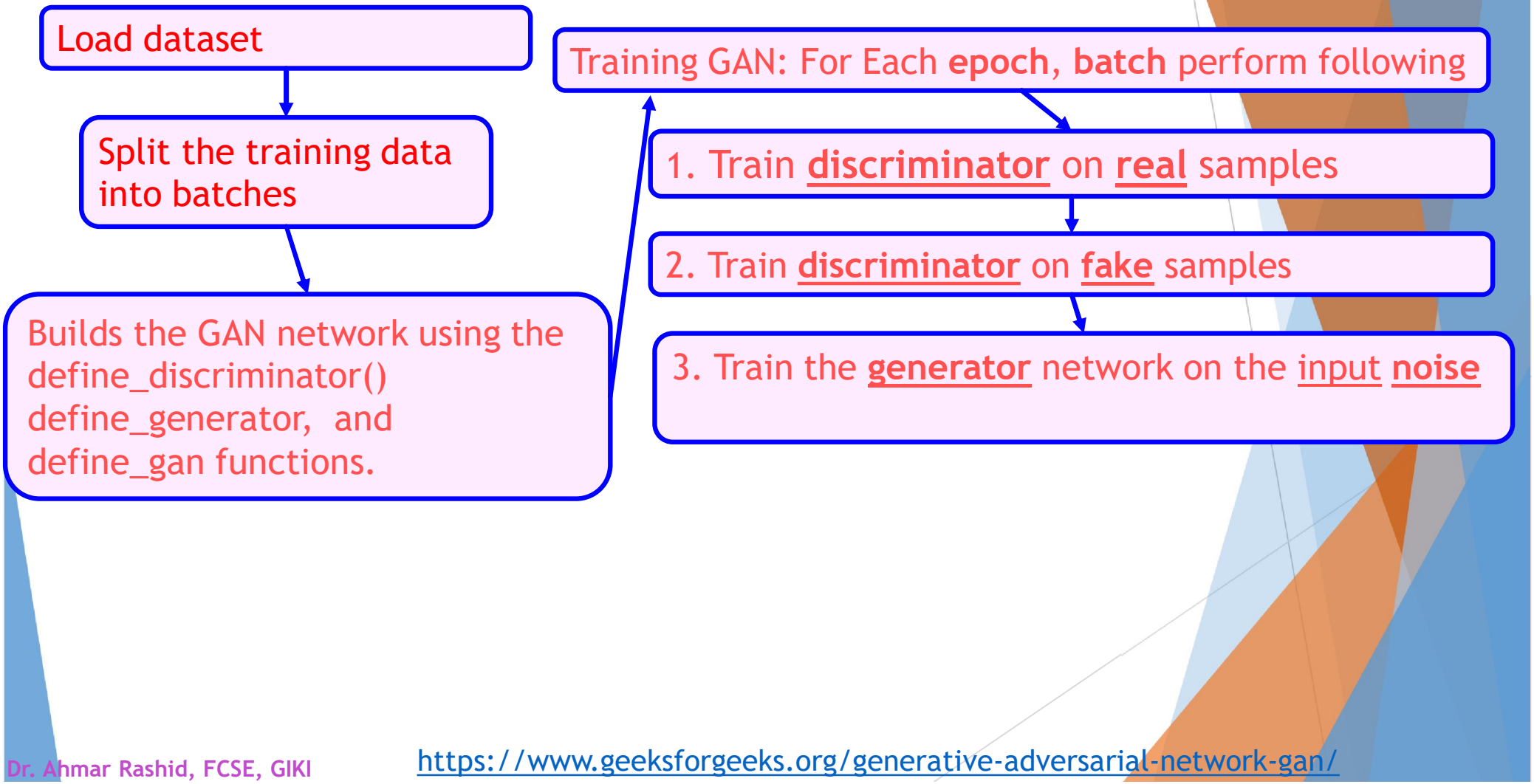

https://www.geeksforgeeks.org/generative-adversarial-network-gan/

Training Procedure

- **for** each training iteration **do**
  - **for** k steps **do**
    - Sample ***m*** real samples $\{x_1, \ldots, x_m\}$ from real data
    - Sample ***m*** noise samples $\{z_1, \ldots, z_m\}$ and transform with Generator
    - Update the Discriminator by ascending the gradient.
    - $\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\ln D(x^{(i)}) + \ln \left(1 - D\left(G(z^{(i)})\right) \right) \right]$
  - end **for**
  - Sample ***m*** noise samples $\{z_1, \ldots, z_m\}$ and transform with Generator
  - Update the Generator by descending the gradient:
    - $\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \left[\ln D(x^{(i)}) + \ln \left(1 - D\left(G(z^{(i)})\right) \right) \right]$
    - $\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \left[\ln \left(1 - D\left(G(z^{(i)})\right) \right) \right] \rightarrow \nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \left[-\ln \left(D\left(G(z^{(i)})\right) \right) \right]$

Understand the Math and Theory of GANs in ~ 10 mins
https://www.youtube.com/watch?v=J1aG12dLo4I

Training the Generative Adversarial Network



https://www.geeksforgeeks.org/generative-adversarial-network-gan/

Training the Generative Adversarial Network

```
# size of the latent space
latent_dim = 128
# create the discriminator
d_model = define_discriminator()
# create the generator
g_model = define_generator(latent_dim)
# create the gan model
gan_model = define_gan(g_model, d_model)
# load image data
dataset = load_real_samples()
# train the GAN mode model
train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=10, n_batch=32)
```

Create the discriminator model

Create the generator model

Create GAN

Load Image Data

Train the GAN Model

https://www.geeksforgeeks.org/generative-adversarial-network-gan/

Training the Generative Adversarial Network

```
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=100,
n_batch=128, subset_size=None):
  if subset_size is not None:
    dataset_subset=(dataset[0][:subset_size], dataset[1][:subset_size])
  else:
    dataset_subset = dataset
  bat_per_epo = int(dataset_subset[0].shape[0] / n_batch)
  half_batch = int(n_batch / 2)
```

- `# The discriminator model is updated for a half batch of real samples`
- `# and a half batch of fake samples, combined a single batch.`

https://www.geeksforgeeks.org/generative-adversarial-network-gan/

Training the Generative Adversarial Network

```
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=100, n_batch=128, subset_size=None):
    ---
for i in range(n_epochs):
    for j in range(batch_per_epoch):
        # Train discriminator on real sampl
        X_real, y_real = generate_real_samples(dataset_subset, half_batch)
        d_loss_fake, _ = d_model.train_on_batch(X_fake, y_fake)
        # Train generator
        z_input = generate_latent_points(latent_dim, n_batch)
        y_gan = np.ones((n_batch, 1))
        g_loss = gan_model.train_on_batch(z_input, y_gan)
        # Print losses on this batch
        print('Epoch#%d,Batch#%d/%d,d_loss_real=%.3f,d_loss_fake=%.3f,g_loss = %.3f'%
            (i+1, j+1, batch_per_epoch, d_loss_real, d_loss_fake, g_loss))
```

1. Train **discriminator** on **real** samples

2. Train **discriminator** on **fake** samples

- Train the discriminator on real and fake images, separately (half batch each)
- Research showed that separate training is more effective.

Print losses in this batch

https://www.geeksforgeeks.org/generative-adversarial-network-gan/

Load the trained model and generate a few images

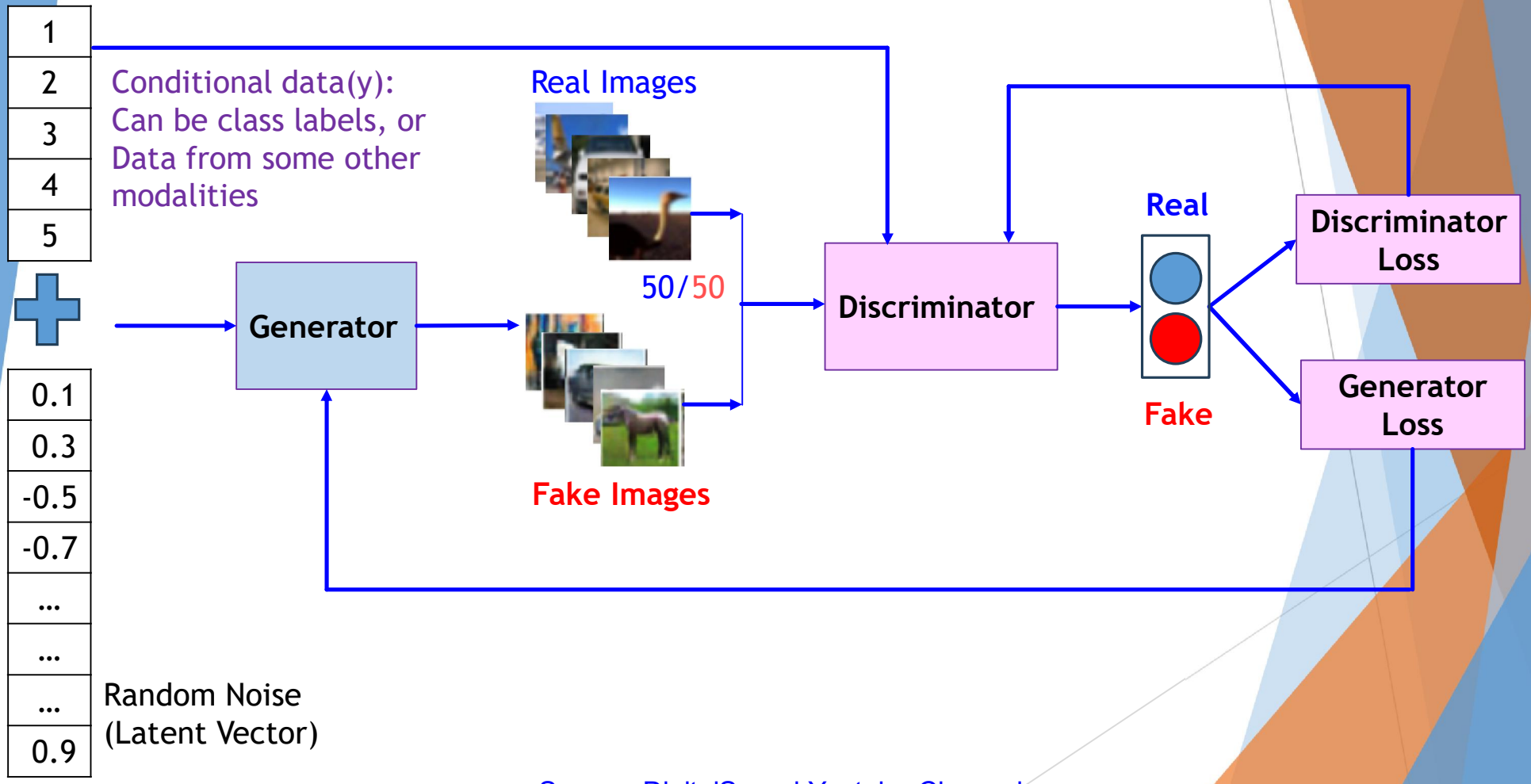
```
google_drive_path = '/content/drive/My Drive/Colab Notebooks/DeepFake/'
model = load_model(google_drive_path + 'GAN_Tea_Disease.h5')
# generate multiple images
latent_points = generate_latent_points(128, 128)
# specify labels - generate 10 sets of labels each gping from 0 to 9
# generate images
X  = model.predict([latent_points])
# scale from [-1,1] to [0,1]
X = (X + 1) / 2.0
X = (X*255).astype(np.uint8)
# plot the result (10 sets of images, all images in a column should be of same
# class in the plot)
# Plot generated images
def show_plot(examples, n):
  for i in range(n * n):
    plt.subplot(n, n, 1 + i)
    plt.axis('off')
    plt.imshow(examples[i, :, :, :])
  plt.show()
show_plot(X, 10)
```

https://www.geeksforgeeks.org/generative-adversarial-network-gan/

Conditional GANs

https://www.geeksforgeeks.org/generative-adversarial-network-gan/

Generative Adversarial Network (GAN)



Source: DigitalSreeni Youtube Channel

Coding Conditional GANs

https://www.geeksforgeeks.org/generative-adversarial-network-gan/

Defining a Utility Class to Build the Generator

```
def define_generator(latent_dim, n_classes=10
  in_label = Input(shape=(1,))  #Input of dimens
  li = Embedding(n_classes, 50)(in_label) #Shap
  n_nodes = 32 * 32  #Linear multiplication, to
  concatenation later in this step.

  li = Dense(n_nodes)(li)    # Dense layer of size  1 x
  li = Reshape((32, 32, 1))(li) # (32x32x1) reshape to add
  in_lat = Input(shape=(latent_dim,))  # image generator
```

Label **Input** of dimension 1

- **Embedding** for categorical input
- Each **label** (total 10 classes for cifar), will be represented by a vector of **size 50**.

- **Linear multiplication**
- To match the dimensions for concatenation later in this step.

Create a **dense layer** of size **1 x 1024**

Reshape to **additional channel**

Image **generator input**
Input of dimension **latent_dim** (e.g., **100**)

Source: DigitalSreeni Youtube Channel

Build up the Generator

- This part is usually **same** as **unconditional GAN** until the output layer.

```
def define_generator(latent_dim, n_classes = 10):
    # Image generator input
    in_lat = Input(shape=(latent_dim,))  # image generator Input of dimension 100

    # Foundation for 32x32 image
    n_nodes = 32 * 32 * 256
    gen = Dense(n_nodes)(in_lat)  # shape = 262,144
    gen = Reshape((32, 32, 256))(gen)  # Shape=32x32x2
    gen = BatchNormalization()(gen)
    gen = ReLU()(gen)

    # merge image gen and label input
    merge = Concatenate()([gen, li])  #Shape=32x32x257 (Extra channel corresponds to the label)
    # Upsample to 64x64x128
    gen = Conv2DTranspose(128, (4, 4), strides=(2, 2), padding='same')(merge)
    gen = BatchNormalization()(gen)
    gen = ReLU()(gen)

    # Upsample to 128x128x64
    gen = Conv2DTranspose(32, (4, 4), strides=(2, 2), padding='same')(gen)
    gen = BatchNormalization()(gen)
    gen = ReLU()(gen)

    # Upsample to 256x256x32
    gen = Conv2DTranspose(16, (4, 4), strides=(2, 2), padding='same')(gen)
    gen = BatchNormalization()(gen)
    gen = ReLU()(gen)

    # Output layer ➔ 256x256x3
    out_layer = Conv2D(3, (8, 8), activation='tanh', pad

    # Define model
    model = Model([in_lat, in_label], out_layer)

    return model  # Model not compiled as it is not directly trained like the discriminator.
```

- The difference is that generated Image **gen** is merged with its label input **li**, represented as an extra channel
- The merged image is then passed on to the next layer

- While defining model **inputs** we will **combine input label** and the **latent input**.

https://www.geeksforgeeks.org/generative-adversarial-network-gan/

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	[(None, 128)]	0
dense_1 (Dense)	(None, 262144)	33816576
reshape (Reshape)	(None, 32, 32, 256)	0
batch_normalization (BatchNormalization)	(None, 32, 32, 256)	1024
re_lu (ReLU)	(None, 32, 32, 256)	0
conv2d_transpose (Conv2DTranspose)	(None, 64, 64, 128)	524416
batch_normalization_1 (BatchNormalization)	(None, 64, 64, 128)	512
re_lu_1 (ReLU)	(None, 64, 64, 128)	0
conv2d_transpose_1 (Conv2DTranspose)	(None, 128, 128, 64)	131136
batch_normalization_2 (BatchNormalization)	(None, 128, 128, 64)	256
re_lu_2 (ReLU)	(None, 128, 128, 64)	0
conv2d_transpose_2 (Conv2DTranspose)	(None, 256, 256, 32)	32800
batch_normalization_3 (BatchNormalization)	(None, 256, 256, 32)	128
re_lu_3 (ReLU)	(None, 256, 256, 32)	0
conv2d_3 (Conv2D)	(None, 256, 256, 3)	6147

Total params: 34512995 (131.66 MB)
Trainable params: 34512035 (131.65 MB)
Non-trainable params: 960 (3.75 KB)

Layer (type)	Output Shape	Param #	Connected to
input_42 (InputLayer)	[(None, 128)]	0	[]
input_41 (InputLayer)	[(None, 1)]	0	[]
dense_41 (Dense)	(None, 262144)	33816576	['input_42[0][0]']
embedding_20 (Embedding)	(None, 1, 50)	500	['input_41[0][0]']
leaky_re_lu_36 (LeakyReLU)	(None, 262144)	0	['dense_41[0][0]']
dense_40 (Dense)	(None, 1, 1024)	52224	['embedding_20[0][0]']
reshape_32 (Reshape)	(None, 32, 32, 256)	0	['leaky_re_lu_36[0][0]']
reshape_31 (Reshape)	(None, 32, 32, 1)	0	['dense_40[0][0]']
concatenate_19 (Concatenate)	(None, 32, 32, 257)	0	['reshape_32[0][0]', 'reshape_31[0][0]']
conv2d_transpose_20 (Conv2DTranspose)	(None, 64, 64, 128)	526464	['concatenate_19[0][0]']
batch_normalization_15 (BatchNormalization)	(None, 64, 64, 128)	512	['conv2d_transpose_20[0][0]']
re_lu_13 (ReLU)	(None, 64, 64, 128)	0	['batch_normalization_15[0][0]']
conv2d_transpose_21 (Conv2DTranspose)	(None, 128, 128, 64)	131136	['re_lu_13[0][0]']
batch_normalization_16 (BatchNormalization)	(None, 128, 128, 64)	256	['conv2d_transpose_21[0][0]']
re_lu_14 (ReLU)	(None, 128, 128, 64)	0	['batch_normalization_16[0][0]']
conv2d_transpose_22 (Conv2DTranspose)	(None, 256, 256, 32)	32800	['re_lu_14[0][0]']
batch_normalization_17 (BatchNormalization)	(None, 256, 256, 32)	128	['conv2d_transpose_22[0][0]']
re_lu_15 (ReLU)	(None, 256, 256, 32)	0	['batch_normalization_17[0][0]']
conv2d_28 (Conv2D)	(None, 256, 256, 3)	6147	['re_lu_15[0][0]']

Total params: 34566743 (131.86 MB)
Trainable params: 34566295 (131.86 MB)
Non-trainable params: 448 (1.75 KB)

None

Defining a Utility Class to Build the Discriminator

```
def define_discriminator(in_shape=(32,32,3), n_classes
in_label = Input(shape=(1,))  #label input of Shape 1
li = Embedding(n_classes, 50)(in_label) #Shape
n_nodes = in_shape[0] * in_shape[1]  #256x256 = 10
li = Dense(n_nodes)(li)  #Shape = 1, 65,536
li = Reshape((in_shape[0], in_shape[1], 1)
# image input
in_image = Input(shape=in_shape) #256x256x3
# concat label as a channel
merge = Concatenate()([in_image, li]) # 256x256x4 (4 channels, 3 for
                                      # image and the other for labels)
```

Label **Input** of shape 1

- **li = Embedding** for categorical **input**
- Each **label** (total 10 classes for cifar), will be represented by a vector of **size 50**, which in turn will be **learned by the discriminator**

Scale up to image dimensions with linear activation

Reshape to additional channel

The **input dimension** of the first layer is a is **256x256x3**, which is the **same** as the **output dimension** of the **generator's** last layer.

- Concatenate label with image as a channel
- Merged image = **256x256x4** (4 channels, 3 for image and the other for labels)

Source: DigitalSreeni Yo

Build up the Discriminator

```
def define_discriminator(in_shape=(32,32,3), n_classes=10):
    # Input layer for real or generated image
    in_image = Input(shape=image_shape)
    # concat label as a channel
    merge = Concatenate()([in_image, li]) #32x32x4 (4 channels, 3 for in
    # Downsample to 128x128x64
    d = Conv2D(64, (4, 4), strides=(2, 2), padding='same')(merge)
    d = LeakyReLU(alpha=0.2)(d)
    d = Dropout(0.25)(d)
    # Downsample to 64x64x128
    d = Conv2D(128, (4, 4), strides=(2, 2), paddin
    d = LeakyReLU(alpha=0.2)(d)
    d = Dropout(0.25)(d)
    # Downsample to 32x32x256
    d = Conv2D(256, (4, 4), strides=(2, 2), padding='same')(d)
    d = LeakyReLU(alpha=0.2)(d)
    d = Dropout(0.25)(d)
    # Flatten and output a single classification value
    d = Flatten()(d)
    out_layer = Dense(1, activation='sigmoid')(d)

    # Define model
    model = Model([in_image, in_label], out_layer)
    opt = Adam(lr=0.0002, beta_1=0.5)
    model.compile(loss='binary_crossentropy', optimizer=opt, metrics=['accuracy'])
    return model
```

- This part is usually **same** as **unconditional GAN** until the output layer.

The input dimension of the first layer is image shape **256x256x3**, and **number of classes**

- The difference is that **input Image** is merged with its label input **li**, represented as an extra channel
- The merged image is then passed on to the next layer

- While defining model **inputs** we will **combine input label** `in_label` with the **input image** `in_image`

https://www.geeksforgeeks.org/generative-adversarial-network-gan/

Build up the Discrimnator

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 256, 256, 3)]	0
conv2d (Conv2D)	(None, 128, 128, 64)	3136
leaky_re_lu (LeakyReLU)	(None, 128, 128, 64)	0
dropout (Dropout)	(None, 128, 128, 64)	0
conv2d_1 (Conv2D)	(None, 64, 64, 128)	131200
leaky_re_lu_1 (LeakyReLU)	(None, 64, 64, 128)	0
dropout_1 (Dropout)	(None, 64, 64, 128)	0
conv2d_2 (Conv2D)	(None, 32, 32, 256)	524544
leaky_re_lu_2 (LeakyReLU)	(None, 32, 32, 256)	0
dropout_2 (Dropout)	(None, 32, 32, 256)	0
flatten (Flatten)	(None, 262144)	0
dense (Dense)	(None, 1)	262145

Total params: 921025 (3.51 MB)
Trainable params: 921025 (3.51 MB)
Non-trainable params: 0 (0.00 Byte)

Layer (type)	Output Shape	Param #	Connected to
input_47 (InputLayer)	[(None, 1)]	0	[]
embedding_23 (Embedding)	(None, 1, 50)	500	['input_47[0][0]']
dense_46 (Dense)	(None, 1, 65536)	3342336	['embedding_23[0][0]']
input_48 (InputLayer)	[(None, 256, 256, 3)]	0	[]
reshape_35 (Reshape)	(None, 256, 256, 1)	0	['dense_46[0][0]']
concatenate_22 (Concatenate)	(None, 256, 256, 4)	0	['input_48[0][0]', 'reshape_35[0][0]']
conv2d_35 (Conv2D)	(None, 128, 128, 64)	4160	['concatenate_22[0][0]']
leaky_re_lu_43 (LeakyReLU)	(None, 128, 128, 64)	0	['conv2d_35[0][0]']
dropout_24 (Dropout)	(None, 128, 128, 64)	0	['leaky_re_lu_43[0][0]']
conv2d_36 (Conv2D)	(None, 64, 64, 128)	73856	['dropout_24[0][0]']
leaky_re_lu_44 (LeakyReLU)	(None, 64, 64, 128)	0	['conv2d_36[0][0]']
dropout_25 (Dropout)	(None, 64, 64, 128)	0	['leaky_re_lu_44[0][0]']
conv2d_37 (Conv2D)	(None, 32, 32, 256)	295168	['dropout_25[0][0]']
leaky_re_lu_45 (LeakyReLU)	(None, 32, 32, 256)	0	['conv2d_37[0][0]']
dropout_26 (Dropout)	(None, 32, 32, 256)	0	['leaky_re_lu_45[0][0]']
flatten_11 (Flatten)	(None, 262144)	0	['dropout_26[0][0]']
dense_47 (Dense)	(None, 1)	262145	['flatten_11[0][0]']

Total params: 3978165 (15.18 MB)
Trainable params: 3978165 (15.18 MB)
Non-trainable params: 0 (0.00 Byte)

Building the Generative Adversarial Network

```
def define_gan(g_model, d_model):
    d_model.trainable = False  # Discriminator is trained separately. So set to not trainable.
    # first, get noise and label inputs from generator model
    gen_noise, gen_label = g_model.input  #Latent vector size and label size
    # get image output from the generator model
    gen_output = g_model.output  #32x32x3
    # generator image output and corresponding input label are inputs to discriminator
    gan_output = d_model([gen_output, gen_label])
    # define gan model as taking noise and label and outputting a classification
    model = Model([gen_noise, gen_label], gan_output)
    # compile model
    opt = Adam(lr=0.0002, beta_1=0.5)
    model.compile(loss='binary_crossentropy', optimizer=opt)
    return model
```

A pre-trained generator model

Aa pre-trained discriminator model

https://www.geeksforgeeks.org/generative-adversarial-network-gan/

Building the Generative Adversarial Network

```
def define_gan(g_model, d_model):
    d_model.trainable = False  #Discriminator is trained separately. So set to not trainable.
    # First, get noise and label inputs from generator model
    gen_noise, gen_label = g_model.input  #Latent vector size and label size
    # Get image output from the generator model
    gen_output = g_model.output  #32x32x3
    # Generator image output and corresponding input label are inputs
    gan_output = d_model([gen_output, gen_label])
    # Define gan model as taking noise and label and outputting a classification
    model = Model([gen_noise, gen_label], gan_output)
    # Compile model
    opt = Adam(lr=0.0002, beta_1=0.5)
    model.compile(loss='binary_crossentropy', optimizer=opt)
    return model
```

A pre-trained generator model

Aa pre-trained discriminator model

Get noise and label inputs from generator model

Get image output from the generator model

Generator image output and corresponding input label are inputs

Define gan model as taking noise and label and outputting a classification

Compile model

https://www.geeksforgeeks.org/generative-adversarial-network-gan/

Building the Generative Adversarial Network

```
def define_gan(g_model, d_model):
  d_model.trainable = False  #Discriminator is trained separately. So set to not trainable.
  ## connect generator and discriminator...
  # first, get noise and label inputs from generator model
  gen_noise, gen_label = g_model.input  #Latent vector size and label size
  # get image output from the generator model
  gen_output = g_model.output  #32x32x3
  # generator image output and corresponding input label are inputs to discriminator
  gan_output = d_model([gen_output, gen_label])
  # define gan model as taking noise and label and outputting a classification
  model = Model([gen_noise, gen_label], gan_output)
  # compile model
  opt = Adam(lr=0.0002, beta_1=0.5)
  model.compile(loss='binary_crossentropy', optimizer=opt)
  return model
```

Differences are highlighted in blue color

https://www.geeksforgeeks.org/generative-adversarial-network-gan/

Generate Real Samples

```
def generate_real_samples(dataset, n_samples):
  # split into images and labels
  images, labels = dataset
  # choose random instances
  ix = randint(0, images.shape[0], n_samples)
  # select images and labels
  X, labels = images[ix], labels[ix]
  # generate class labels and assign to y (don't confuse this with the above labels that
  # correspond to cifar labels)
  y = ones((n_samples, 1))  #Label=1 indicating they are real
  return [X, labels], y
```


https://www.geeksforgeeks.org/generative-adversarial-network-gan/

Generate points in latent space as input for the generator

```
def generate_latent_points(latent_dim, n_samples, n_classes=10):
  # generate points in the latent space
  x_input = randn(latent_dim * n_samples)
  # reshape into a batch of inputs for the network
  z_input = x_input.reshape(n_samples, latent_dim)
  # generate labels
  labels = randint(0, n_classes, n_samples)
  return [z_input, labels]
```


https://www.geeksforgeeks.org/generative-adversarial-network-gan/

Generate Fake Samples

```
def generate_fake_samples(generator, latent_dim, n_samples):
  # generate points in latent space
  z_input, labels_input = generate_latent_points(latent_dim, n_samples)
  # predict outputs
  images = generator.predict([z_input, labels_input])
  # create class labels
  y = zeros((n_samples, 1))  #Label=0 indicating they are fake
  return [images, labels_input], y
```


https://www.geeksforgeeks.org/generative-adversarial-network-gan/

Training the Generative Adversarial Network

```
def train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=100,
n_batch=128, subset_size=None):
    if subset_size is not None:
        dataset_subset = (dataset[0][:subset_size], dataset[1][:subset_size])
    else:
        dataset_subset = dataset
    bat_per_epo = int(dataset_subset[0].shape[0] / n_batch)
    half_batch = int(n_batch / 2)
```

- # The discriminator model is updated for a half batch of real samples
- # and a half batch of fake samples, combined a single batch.

https://www.geeksforgeeks.org/generative-adversarial-network-gan/

Training the Generative Adversarial Network

```
def train(g_model, d_model, gan_
subset_size=None):
    ---
    for i in range(n_epochs):
        for j in range(batch_per_epoch):
            [X_real, labels_real], y_real = generate_re
            d_loss_fake, _ = d_model.train_on_batch([X_fake, labels], y_fake)
            [z_input, labels_input] = generate_latent_points(latent_dim, n_batch)
            y_gan = np.ones((n_batch, 1))
            g_loss = gan_model.train_on_batch([z_input, labels_input], y_gan)
            print('Epoch>%d, Batch%d/%d, d1=%.3f, d2=%.3f g=%.3f' %
                  (i+1, j+1, bat_per_epo, d_loss_real, d_loss_fake, g_loss))
```

Training GAN: For each **poch**, **batch** perform following

1.Generate a set of **input** real and fake and images

2.1.Train discriminator on real samples

2.2.Train discriminator on fake samples

- Train the discriminator on real and fake images, separately (half batch each)
- Research showed that separate training is more effective.

Print losses in this batch

https://www.geeksforgeeks.org/generative-adversarial-network-gan/

Training the Generative Adversarial Network

```
# size of the latent space
latent_dim = 100
# create the discriminator
d_model = define_discriminator()
# create the generator
g_model = define_generator(latent_dim)
# create the gan model
gan_model = define_gan(g_model, d_model)
# load image data
dataset = load_real_samples()
# train the GAN mode model
train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=5, n_batch=512)
#train(g_model, d_model, gan_model, dataset, latent_dim, n_epochs=2, n_batch=512, subset_size=5120)
```

Create the discriminator model

Create the generator model

Create GAN

Load Image Data

Train the GAN Model

https://www.geeksforgeeks.org/generative-adversarial-network-gan/