quantiphi

A Primer on understanding GANs

"The coolest idea in deep learning in the last 20 years."
— Yann LeCun, Director of AI Research, Facebook

Overview of the session

- Introduction to generative models
- Explicit prob. density estimation - PixelCNN, PixelRNN, VAE
- Implicit prob. Density estimation - GAN
- GAN theory and code
- Why there are so many variants of GANs (If time permits)
- Research directions in GANs
- Personal experience, tips and tricks with training GANs interspersed in between

ML Paradigms/Settings

- **Supervised**:

  Given bunch of data x and label y, learn a function to map x → y

  Ex - Most of the work we at Quantiphi do: classification, regression, object detection, semantic/instance segmentation, image captioning and so on.
- **Unsupervised**:

  Given bunch of data x, learn some underlying hidden structure of data

  Ex - clustering, dimensionality reduction, feature learning and so on.
- **RL** is a different setting altogether. Not in scope.
- Generative models are an example of unsupervised learning.

Generative Models

- Try to learn the underlying data distribution in an implicit or explicit way.
- That is given $p_{data}(x)$, we want to find a $p_{model}(x)$, where x is the space of the image. That is for a 8-bit 28x28x1 MNIST image, this space is 784 dimensional with 256 possible values along each dimension.
- Such high dimensionality !! You can't expect to learn a simple 784-dim distribution explicitly.
- Unless you condition it on something.
- Well, condition it on past pixels
- What exactly is past pixels?

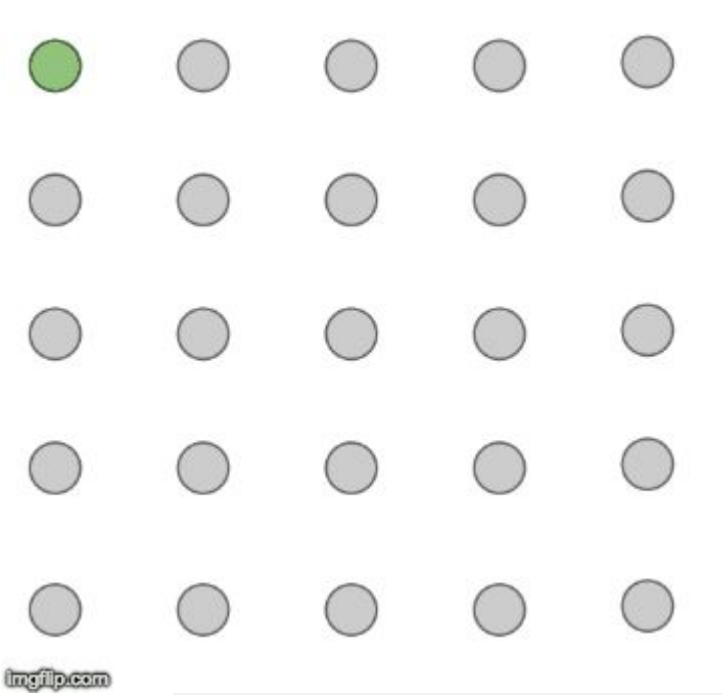
Fully visible belief nets

- We explicitly try to model the probability of an image occuring in the dataset !!

$$p(x) = \prod_{i=1}^{n} p(x_i | x_0, x_1, .., x_{i-1})$$

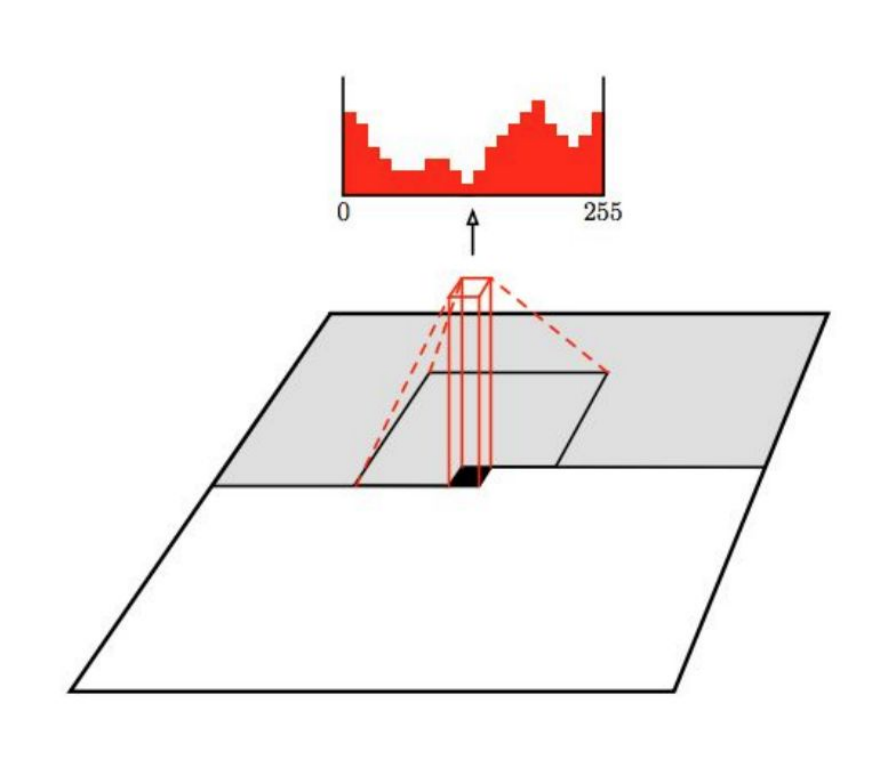
- This is what past pixels mean.
- We model the dependency using an RNN called PixelRNN or a CNN called PixelCNN
- Other than the dependency, the architecture is exactly the same.

PixelRNN dependency modelling from TL to BR



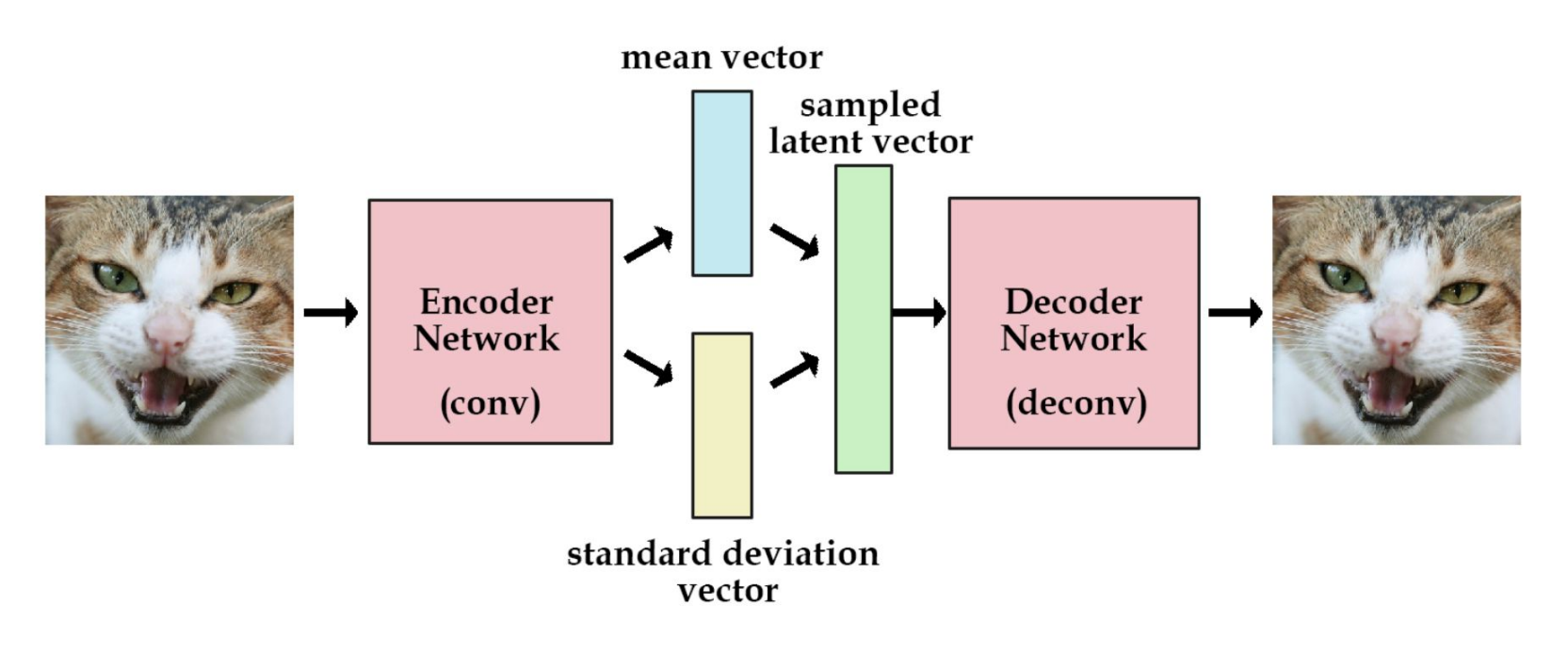
Source: CS231N Lecture Notes, Stanford

PixelCNN contextual dependency modelling



Source: Conditional Image Generation with Pixel CNN Decoders, van der Oord et al., 2016

VAE



Source: Variational Autoencoders Explained, Kevin Frans's blog

VAE

- A dimensionality reduction technique for images. Can be used as a data compressor!!
- Obj - Learn a lower-dimensional feature representation from original image dataset so that original images can be reconstructed.

  Ideal:

$$P(x) = \int p_\theta(z) p_\theta(x|z) dz$$

- Sadly this integral is intractable because we can't compute P(x | z) for every z.
- So we don't optimize P(x) but rather calculate a lower bound of the expression using variational algebra and optimize that instead.

$$log[P_\theta(x|z)] = E_z[log(P_\theta(x|z))] - D_{KL}[q_\phi(z|x)||P_\theta(z)]$$

More on VAE

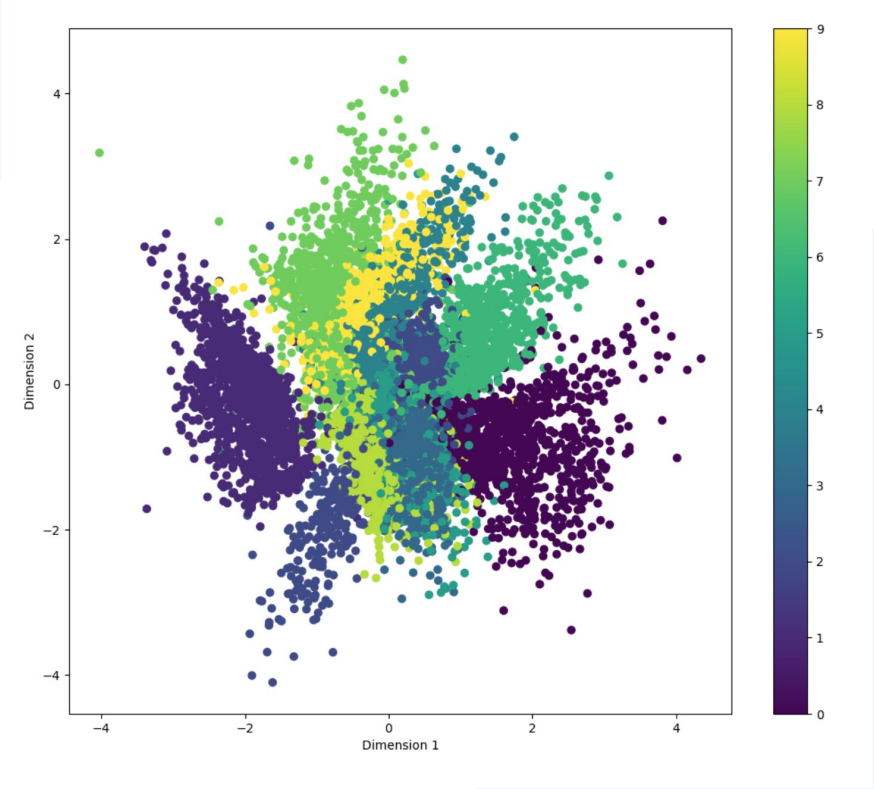
- VAE is an improvement over AE with a “reparametrization trick”
- VAE’s loss is sum of reconstruction loss and KL divergence between Gaussian normal and the distribution of z

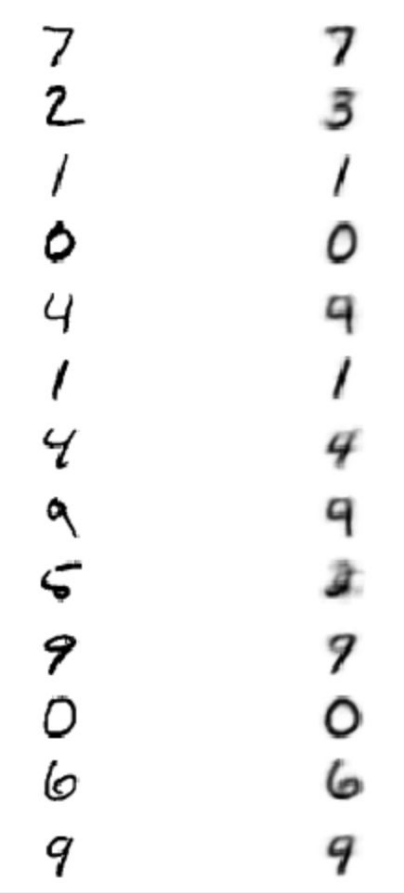
VAE Code Snippet

```
inputs = Input(shape=input_shape, name='encoder_input')
z_mean, z_log_var = encoder(inputs)
z = sample_z(z_mean, z_log_var)
outputs = decoder(z)
vae = Model(inputs, outputs, name='vae_mlp')
vae.compile(loss=vae_loss, optimizer=Adam(lr=1e-3)
vae.fit(x=x_train, y=x_train, validation_data=(x_test, x_test)\
        batch_size=batch_size, epochs=epochs)
```

Source: https://github.com/piyush-kgp/VAE-MNIST-Keras

Results with VAE in 2-D for MNIST



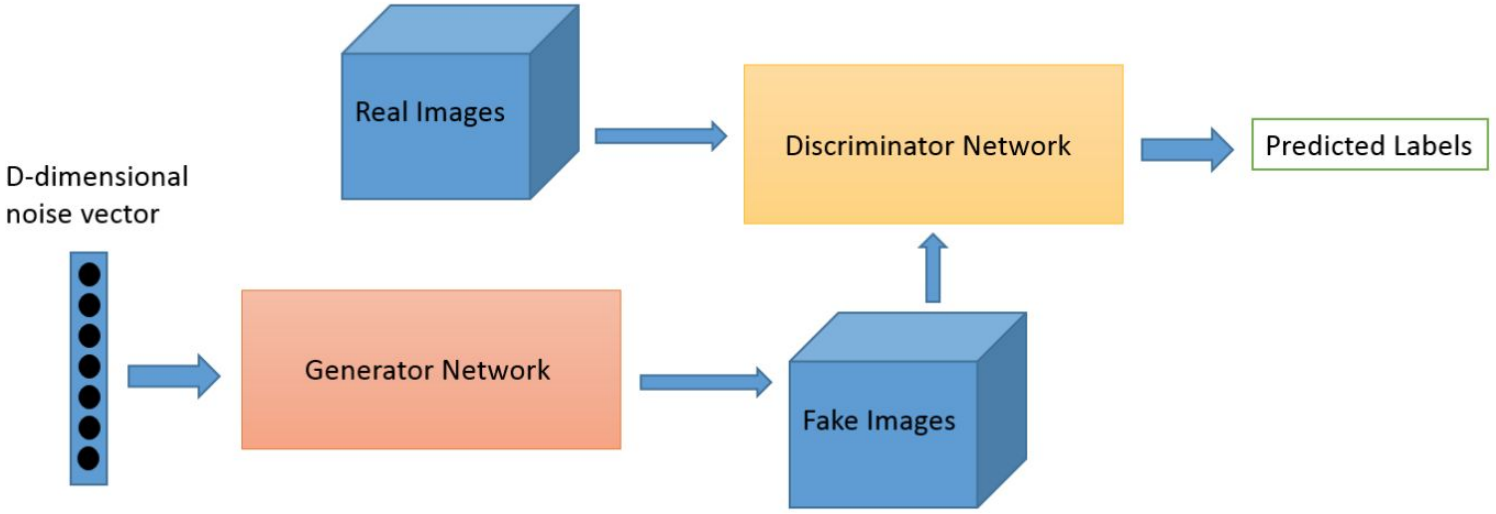


Source: https://github.com/piyush-kgp/VAE-MNIST-Keras

Problems with these/ Why we need GANs

- Deep belief nets take too long to generate a new sample because of its arch.
- VAE-generated samples are blurred because we are optimizing the lower bound of the likelihood since the original integral is intractable.
- GANs take a game-theoretic approach approach which is very different from anything seen so far.
- Learn to generate from training distribution through 2-player game using just random noise.
- Typical analogy drawn is with a counterfeiter and police where with time, the counterfeiter becomes more and more clever in forging currency and police becomes more and more smart in telling fake from real currency.

The GAN framework



A simple flowchart of a GAN

Source: GANs, a modern perspective, Medium.

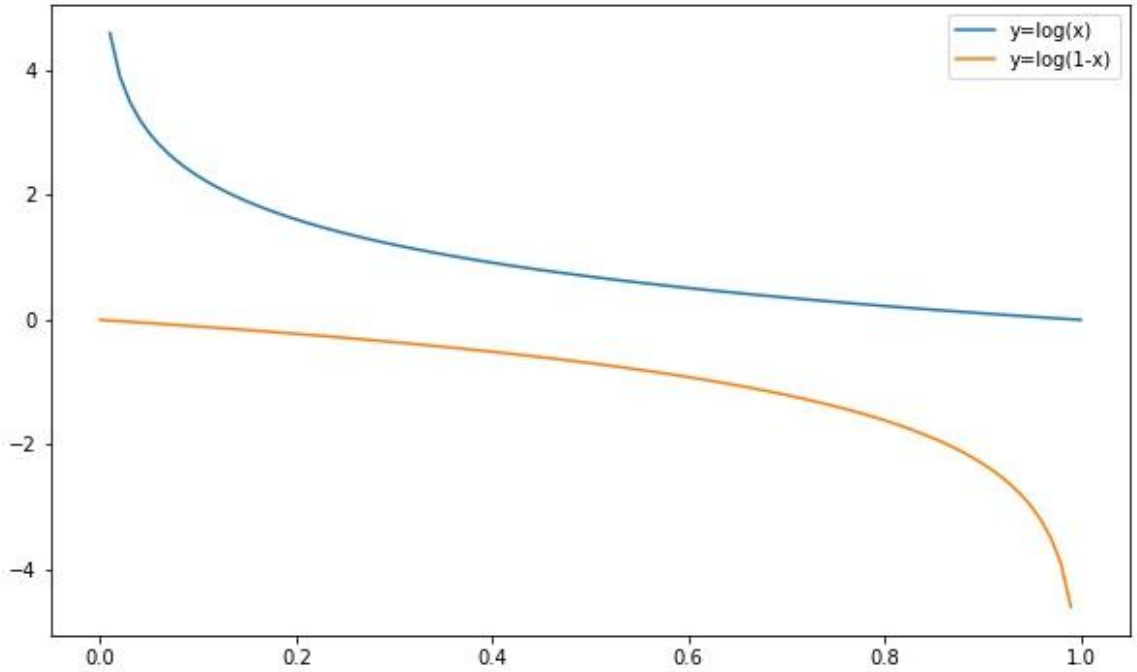
Objective Function

- Discriminator or D tries to label fake (produced by G) as 0 and real (from actual dataset) as 1 and Generator or G tries to fool the discriminator. Hence the game.

$$\min_G \max_D V(D,G) = E_{x\sim X} log(D(x)) + E_{z\sim Z} log(1 - D(G(z)))$$

- Consider D to be parametrized by θ_d and G by θ_g, then we want to obtain θ_d such that V is maximized and θ_g such that V is minimized.
- Notice that if V is maximum; it means it does 100% job on the detection task ie all D(x) are 1's and all D(G(z)) are 0's.
- And that if V is minimum; all D(G(z)) are 1's. This corresponds to 100% forgery success.

Gradient issues



Gradient issues [contd.]

- X-axis denotes D(G(z)) and y-axis denotes the term in value function
- Notice that the orange line which is in our original V function, near x=0, where D is pretty sure that G(z) is fake, the gradient is small.
- This is very bad.
- So we bring in another monotonically decreasing function which has high gradient near 0 and low gradient near 1.
- This new function is slight modification of log(1-D(G(z))) ie log(D(G(z)). This is the blue curve.

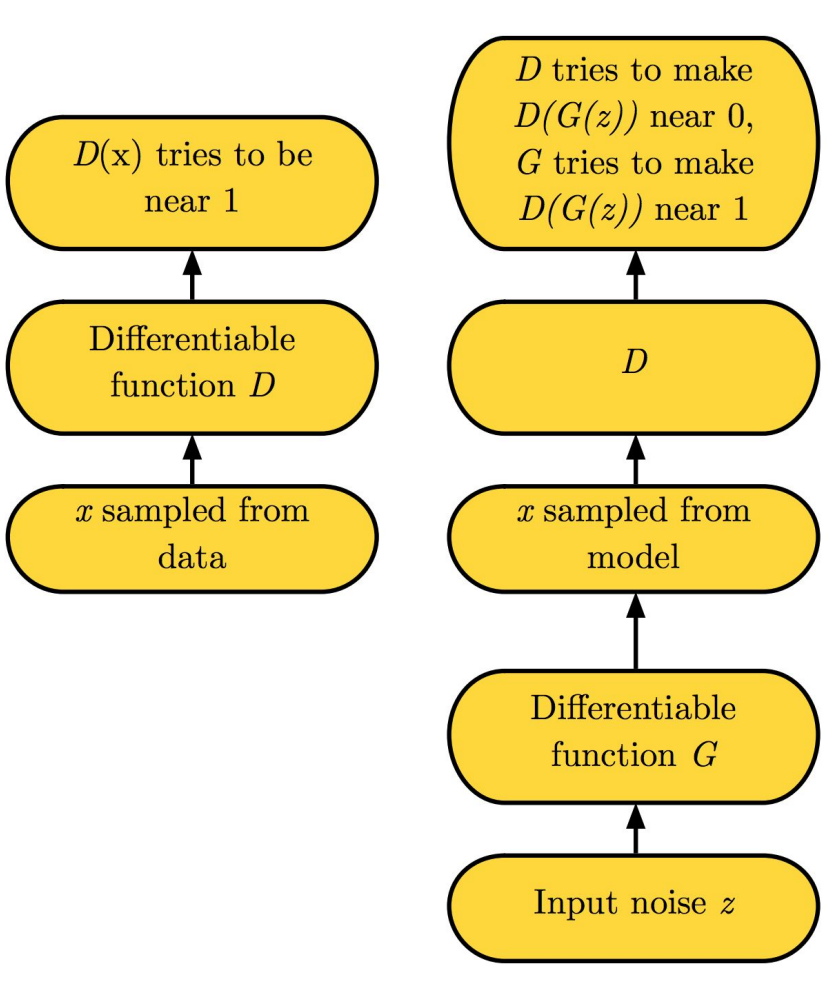
Modified objective function

- In terms of the game, instead of doing good on correctly labelling fake images as fake, we want to do bad on incorrectly labelling fake images as real. [Code]
- And in terms of minmax game, instead of maximizing likelihood of discriminator being correct, now minimize likelihood of discriminator being wrong.
- Thus we maximize two value functions, 1st called DM, 2nd called AM.

$$\max_D V(D,G) = E_{x\sim X} log(D(x)) + E_{z\sim Z} log(1 - D(G(z)))$$

$$\max_G V(D,G) = E_{z\sim Z} log(D(G(z)))$$

The algorithm



Source: NIPS 2016 Tutorial: Generative Adversarial Networks, Ian Goodfellow

Code - Imports

```
from tensorflow.keras.models  import Sequential, Model
from tensorflow.keras.layers import *
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.datasets import mnist
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import imageio
import glob
```

Code[contd] - Constructor

```
class DCGAN:
    def __init__(self):
        (self.data, _), (_, _) = mnist.load_data()
        self.latent_dim = 100
        self.input_shape = self.data.shape[1:] + (1,)
        self.batch_size = 32
        self.num_steps = 4000
        self.save_dir = 'plots/'
        self.save_interval=100
        self.preprocess_data()
        self.make_gan()
```

Code[contd] - Reading real images

```
def preprocess_data(self):
    # Scale in -1 to 1
    self.data = self.data.astype('float32')/127.5-1
    self.data = np.expand_dims(self.data, axis = -1)

def get_real_images_batch(self):
    idx = np.random.randint(0, self.data.shape[0], \
          self.batch_size)
    return self.data[idx]
```

Code[contd] - Generator

```
def make_generator_model(self):
    model = Sequential()
    model.add(Dense(7*7*256, use_bias=False, input_shape=\
    (self.latent_dim,)))
    model.add(BatchNormalization(momentum=0.8))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Reshape((7, 7, 256)))
    model.add(Conv2DTranspose(128, (5, 5), strides=(1, 1), \
    padding='same', use_bias=False))
    model.add(BatchNormalization(momentum=0.8))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Conv2DTranspose(64, (5, 5), strides=(2, 2), \
    padding='same', use_bias=False))
    model.add(BatchNormalization(momentum=0.8))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Conv2DTranspose(1, (5, 5), strides=(2, 2), \
    padding='same', use_bias=False, activation='tanh'))
    return model
```

G model summary

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 12544)             1254400
_________________________________________________________________
batch_normalization_3 (Batch (None, 12544)             50176
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 12544)             0
_________________________________________________________________
reshape (Reshape)            (None, 7, 7, 256)         0
_________________________________________________________________
conv2d_transpose (Conv2DTran (None, 7, 7, 128)         819200
_________________________________________________________________
batch_normalization_4 (Batch (None, 7, 7, 128)         512
_________________________________________________________________
leaky_re_lu_5 (LeakyReLU)    (None, 7, 7, 128)         0
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 14, 14, 64)        204800
_________________________________________________________________
batch_normalization_5 (Batch (None, 14, 14, 64)        256
_________________________________________________________________
leaky_re_lu_6 (LeakyReLU)    (None, 14, 14, 64)        0
_________________________________________________________________
conv2d_transpose_2 (Conv2DTr (None, 28, 28, 1)         1600
=================================================================
```

Code[contd] - Discriminator

```
def make_discriminator_model(self):
    model = Sequential()
    model.add(Conv2D(32, kernel_size=3, strides=2, input_shape=\
    self.input_shape, padding="same"))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.25))
    model.add(Conv2D(64, kernel_size=3, strides=2, padding="same"))
    model.add(ZeroPadding2D(padding=((0,1),(0,1))))
    model.add(BatchNormalization(momentum=0.8))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.25))
    model.add(Conv2D(128, kernel_size=3, strides=2, padding="same"))
    model.add(BatchNormalization(momentum=0.8))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.25))
    model.add(Conv2D(256, kernel_size=3, strides=1, padding="same"))
    model.add(BatchNormalization(momentum=0.8))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.25))
    model.add(Flatten())
    model.add(Dense(1, activation='sigmoid'))
```

D model summary

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 14, 14, 32)        320
_________________________________________________________________
leaky_re_lu (LeakyReLU)      (None, 14, 14, 32)        0
_________________________________________________________________
dropout (Dropout)            (None, 14, 14, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 7, 7, 64)          18496
_________________________________________________________________
zero_padding2d (ZeroPadding2 (None, 8, 8, 64)          0
_________________________________________________________________
batch_normalization (BatchNo (None, 8, 8, 64)          256
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 8, 8, 64)          0
_________________________________________________________________
dropout_1 (Dropout)          (None, 8, 8, 64)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 4, 4, 128)         73856
_________________________________________________________________
batch_normalization_1 (Batch (None, 4, 4, 128)         512
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 4, 4, 128)         0
_________________________________________________________________
dropout_2 (Dropout)          (None, 4, 4, 128)         0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 4, 4, 256)         295168
_________________________________________________________________
batch_normalization_2 (Batch (None, 4, 4, 256)         1024
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 4, 4, 256)         0
_________________________________________________________________
dropout_3 (Dropout)          (None, 4, 4, 256)         0
_________________________________________________________________
flatten (Flatten)            (None, 4096)              0
_________________________________________________________________
dense (Dense)                (None, 1)                 4097
=================================================================
```

Code[contd] - Make GAN

```
def make_gan(self):
    optimizer = Adam(0.0002, 0.5)
    self.D = self.make_discriminator_model()
    self.D.compile(loss='binary_crossentropy',
        optimizer=optimizer, metrics=['accuracy'])
    self.G = self.make_generator_model()
    z = Input(shape=(self.latent_dim, ))
    img = self.G(z)
    self.D.trainable = False
    validity = self.D(img)
    self.AM = Model(z, validity)
    self.AM.summary()
    self.AM.compile(loss='binary_crossentropy',
        optimizer=optimizer)
```

AM model summary

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 100)               0
_________________________________________________________________
model_1 (Model)              (None, 28, 28, 1)         2330944
_________________________________________________________________
model (Model)                (None, 1)                 393729
=================================================================
```

Code[contd] - Training Loop

```
def train(self):
    valids = np.ones((self.batch_size, 1))
    fakes = np.zeros((self.batch_size, 1))
    for step in range(self.num_steps):
        images_real = self.get_real_images_batch()
        d_loss_real = self.D.train_on_batch(images_real, valids)
        noise = np.random.normal(0, 1, size=\
                [self.batch_size, self.latent_dim])
        images_fake = self.G.predict(noise)
        d_loss_fake = self.D.train_on_batch(images_fake, fakes)
        d_loss, d_acc = 0.5*np.add(d_loss_real, d_loss_fake)
        g_loss = self.AM.train_on_batch(noise, valids)
        if step%self.save_interval==0:
            self.plot_images(step)
```

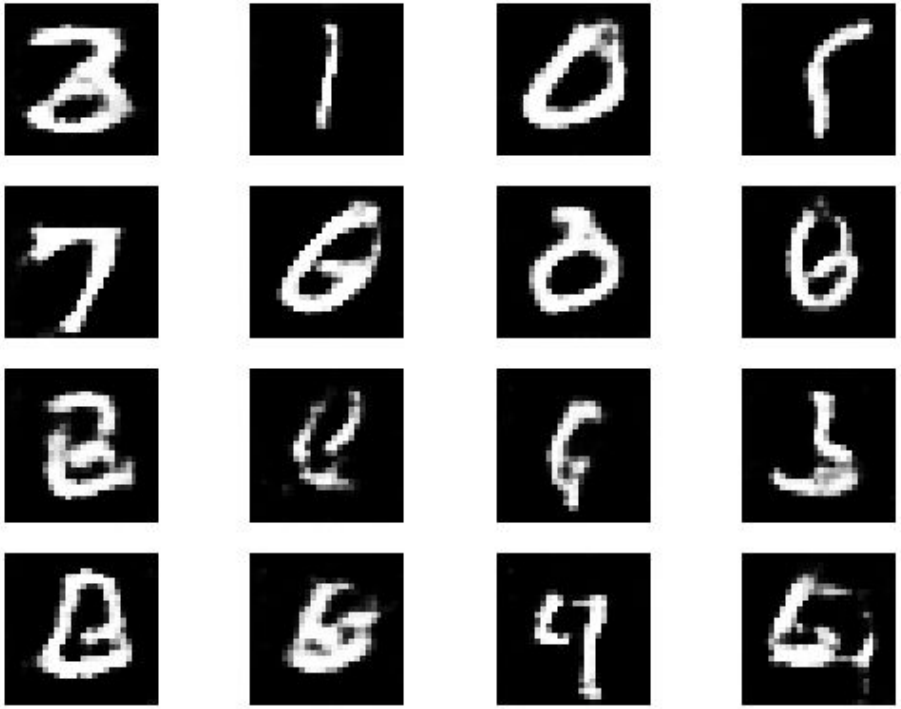
Code[contd] - Plot generated images

```
def plot_images(self, step, num_images = 16):
    noise = np.random.normal(0, 1, size=[num_images,
            self.latent_dim])
    fake_images = self.G.predict(noise)
    fake_images = fake_images*0.5 + 0.5 # rescale to (0, 1)
    fig, axes = plt.subplots(4, 4)
    for i, img in enumerate(fake_images):
        r, c = i//4, i%4
        axes[r, c].imshow(img[:,:,0], cmap='gray')
        axes[r, c].axis('off')
    fig.savefig(self.save_dir+'step_%s.png' %step)
```

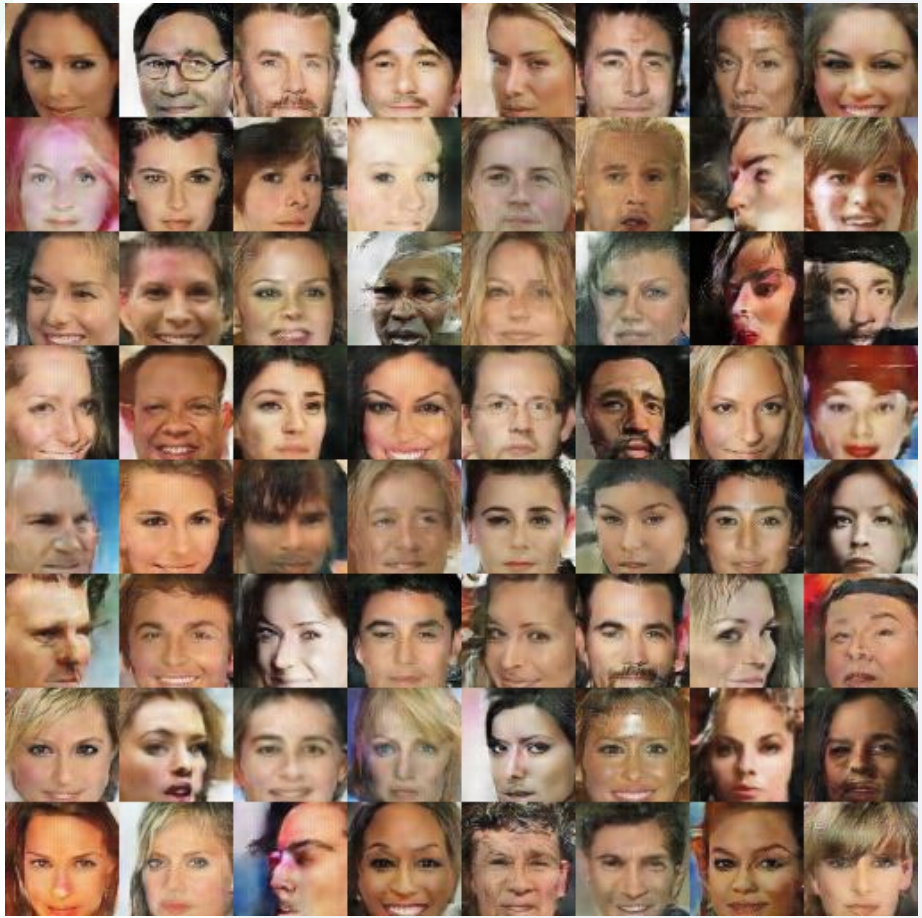
Code[contd] - Run everything

```
if __name__=='__main__':
    tf.logging.set_verbosity(tf.logging.ERROR)
    dcgan = DCGAN()
    dcgan.train()
```

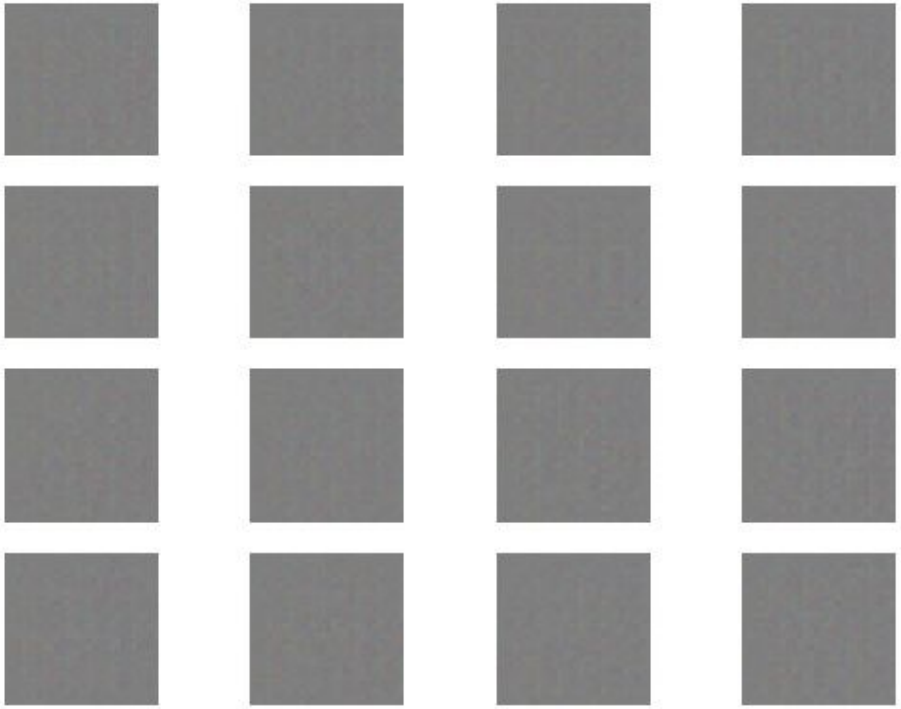
Results from MNIST



Results from CelebA



Generator epoch-wise Progression



How is DCGAN different from vanilla GAN?

- Replace all max pooling with convolutional stride
- Use transposed convolution for upsampling.
- Eliminate fully connected layers.
- Use Batch normalization except the output layer for the generator and the input layer of the discriminator.
- Use ReLU in the generator except for the output which uses tanh.
- Use LeakyReLU in the discriminator.
- Notice that these are really minor improvement tricks rather than big changes.

What do G and D learn in GANs

- D is trained as an unsupervised estimator that gives us a low value for data and higher output for everything else.
- D develops a good internal representation of the data
- Can be used as a feature extractor for a classifier, for example.
- G: Parametrizes the complicated surface of real data.
- Arithmetic on faces in the Z vector space: [man with glasses] - [man without glasses] + [woman without glasses] = [woman with glasses].
- This is similar to vector arithmetic on words in word2vec like [King]-[Man]+[Woman] = Queen.

Vector arithmetic on faces from GANs

Source: CS231N Lecture Notes

Towards better GANs

- Convergence is still the most common woe.
- Improvements can be made by:
- Experiments with network design - CGAN, InfoGAN, SGAN, Prog. Gro. of GAN, SAGAN
- Choosing a better loss function - LSGAN, WGAN, WGAN-GP, EBGAN, BEGAN, RSGAN, RaGAN, DRAGAN
- Training for a different task like Image-to-image translation. (Another major chunk of papers) - CycleGAN, StyleGAN, text-to-image - StackGAN
- We'll look at some of these GANs in brief today. The idea is to understand how different it is from the code we've seen.

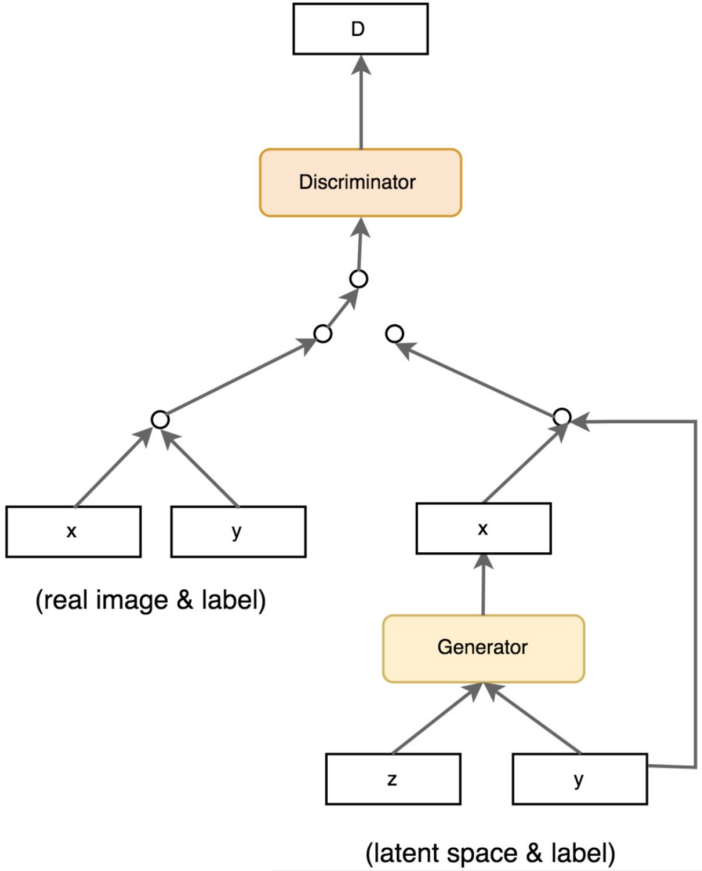
CGAN or Class-conditioned GAN

- We add class label to the information:

$$\min_G \max_D V(D,G) = E_{x\sim X} log(D(x|y)) + E_{z\sim Z} log(1 - D(G(z|y)))$$

- Nothing else changes.
- InfoGAN is a slight variation on CGAN. Out of scope for today.

CGAN Architecture



CGAN Code - Generator

```
def build_generator(self):
    model = Sequential()
    model.add(Dense(256, input_dim=self.latent_dim))
    ...
    model.add(Reshape(self.img_shape))
    noise = Input(shape=(self.latent_dim,))
    label = Input(shape=(1,), dtype='int32')
    label_embedding = Flatten()(Embedding(self.num_classes, \
                        self.latent_dim)(label))
    model_input = multiply([noise, label_embedding])
    img = model(model_input)
    return Model([noise, label], img)
```

CGAN Code[contd] - Discriminator

```
def build_discriminator(self):
    model = Sequential()
    model.add(Dense(512, input_dim=np.prod(self.img_shape)))
    ...
    model.add(Dense(1, activation='sigmoid'))
    img = Input(shape=self.img_shape)
    label = Input(shape=(1,), dtype='int32')
    label_embedding = Flatten()(Embedding(self.num_classes, \
                                np.prod(self.img_shape))(label))
    flat_img = Flatten()(img)
    model_input = multiply([flat_img, label_embedding])
    validity = model(model_input)
    return Model([img, label], validity)
```

CGAN Code - Make GAN

```
def make_gan(self):
    self.D = self.make_discriminator_model()
    self.D.compile(...)
    self.G = self.make_generator_model()
    z = Input(shape=(self.latent_dim, ))
    label = Input(shape=(1,))
    img = self.G([z, label])
    self.D.trainable = False
    validity = self.D([img, label])
    self.AM = Model(z, validity)
    self.AM.compile(...)
```

CGAN Code - Training loop

```
def train(self):
    valids = np.ones((self.batch_size, 1))
    fakes = np.zeros((self.batch_size, 1))
    for step in range(self.num_steps):
        x, y = self.get_real_batch()
        d_loss_real = self.D.train_on_batch([x, y], valids)
        noise = np.random.normal(0, 1, \
                size=[self.batch_size, self.latent_dim])
        gen_imgs = self.G.predict([noise, y])
        d_loss_fake = self.D.train_on_batch(gen_imgs, fakes)
        d_loss, d_acc = 0.5*np.add(d_loss_real, d_loss_fake)
        sampled_labels = np.random.randint(0, 10, batch_size)
        g_loss = self.AM.train_on_batch([noise, \
                 sampled_labels], valids)
```

Progressive growing of GANs by NVIDIA- Best results

Progressive growing of GANs

- Most beautiful results; coming from Nvidia.
- Trains the network in multiple phases.
- In phase 1, it takes in a latent feature z and uses two convolution layers to generate 4×4 images.
- Then, we train the discriminator with the generated images and the 4×4 real images.
- How do you get 4x4 real images? By downsampling (think cv2.resize)
- Once the training stables, we add 2 more convolution layers to upsampling the image to 8×8 and 2 more convolution layers to downsampling images in the discriminator.
- We keep going till 9 phases in total, which then generate 1024 × 1024 images.

Excerpt from paper

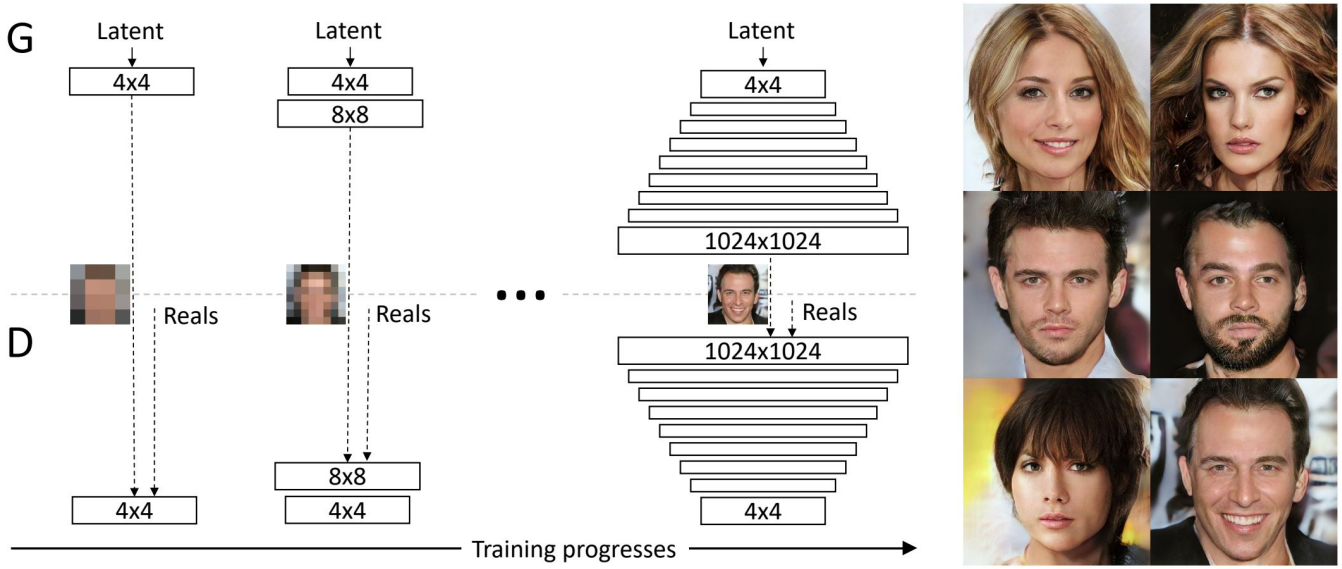


Figure 1: Our training starts with both the generator (G) and discriminator (D) having a low spatial resolution of 4×4 pixels. As the training advances, we incrementally add layers to G and D, thus increasing the spatial resolution of the generated images. All existing layers remain trainable throughout the process. Here $\boxed{N \times N}$ refers to convolutional layers operating on $N \times N$ spatial resolution. This allows stable synthesis in high resolutions and also speeds up training considerably. One the right we show six example images generated using progressive growing at 1024×1024.

Source: Progressive growing of GANs for improved quality, stability and variation, Karras et al., arxiv

Improvements based on loss function - LSGAN

- LSGAN uses MSE instead of cross entropy loss. Other than that everything remains same:

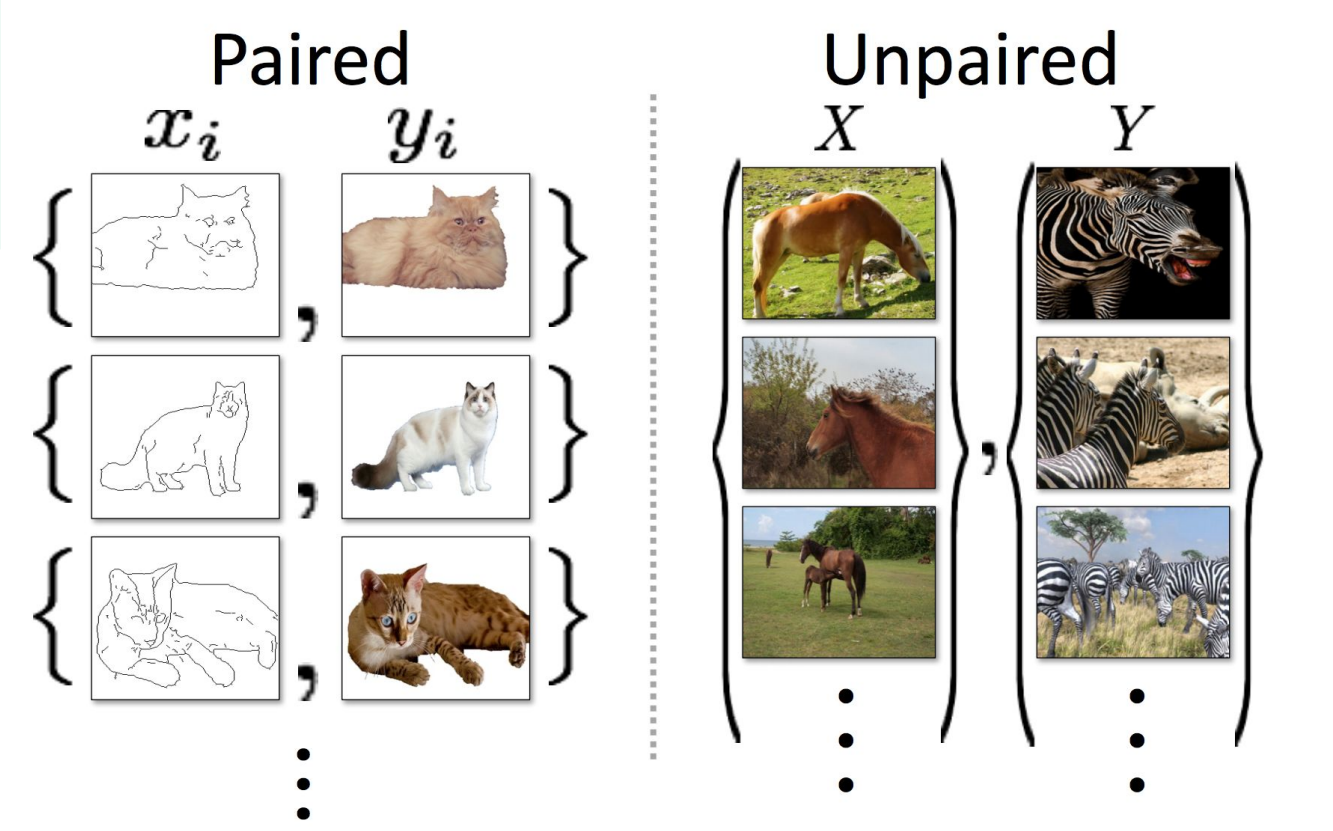
$$\min_D V(D,G) = \frac{1}{2}E_{x\sim X}(D(x)-1)^2 + \frac{1}{2}E_{z\sim Z}(D(G(z)))^2$$

$$\min_G V(D,G) = \frac{1}{2}E_{z\sim Z}(D(G(z))-1)^2$$

LSGAN in code - Make GAN method

```
def make_gan(self):
    ...
    self.D.compile(loss='mse',
        ...
        )
    ...
    self.AM.compile(loss='mse',
        ...
        )
```

Paired vs unpaired image domain translation

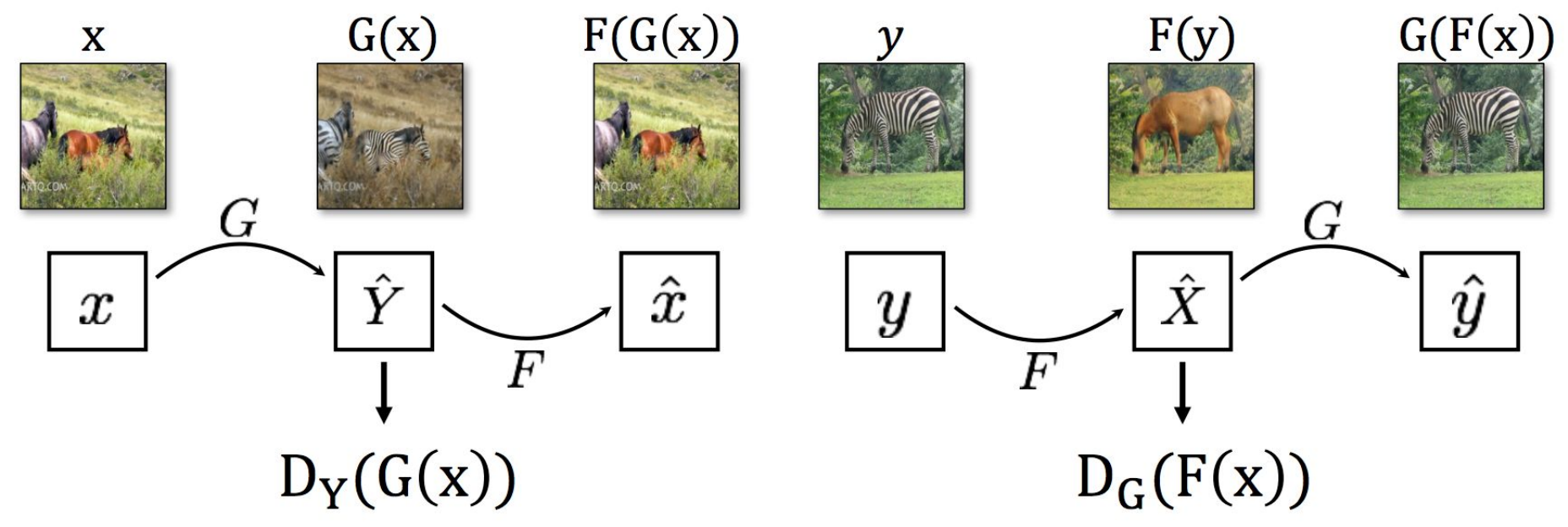


Source: Unpaired Image-to-Image Translation with CycleGAN, CVPR Tutorial, 2018

CycleGAN

- CycleGAN is one of the major **unpaired** image domain translation networks.
- Unpaired image domain translation is preferred over paired because often times obtaining pairs might be costly or pairs may not be available for example which horse corresponds to a zebra?!
- Image domain translation is a very hot research field and several developments have been done post CycleGAN; however CycleGAN remains the baseline.
- Here, we shall restrict ourselves to CycleGAN.

CycleGAN basic arch.



Source: Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks, Zhu et al. 2017

CycleGAN Loss

- Total loss:

$$\mathcal{L}(G, F, D_X, D_Y) = \mathcal{L}_{GAN}(G, D_Y, X, Y) + \mathcal{L}_{GAN}(F, D_X, Y, X) + \lambda \mathcal{L}_{cyc}(G, F)$$

- 1st and 2nd terms are regular GAN losses in X → Y and Y → X respectively and 3rd term is cycle consistency loss for X → Y → X and Y → X → Y
- 3rd term ie cycle consistency loss:

$$\mathcal{L}_{cyc}(G, F) = E_{x \sim X} ||F(G(x)) - x||_1 + E_{y \sim Y} ||G(F(y)) - y||_1$$

- The GAN losses are still computed in the same way.

CycleGAN Code-constructor

```
class CycleGAN:
    def __init__(self, x_img_dir, y_img_dir):
        ...
        self.data_X = self.preprocess_data(x_img_dir)
        self.data_Y = self.preprocess_data(y_img_dir)
        ...
```

CycleGAN Code-Make GAN

```
def make_gan(self):
    self.D_A, self.D_B = ...
    self.D_A.compile(...); self.D_B.compile(...)
    self.G_AB, self.G_BA = ...
    self.G_AB.compile(...); self.G_BA.compile(...)
    img_A = Input(shape=self.img_shape)
    img_B = Input(shape=self.img_shape)
    fake_B = self.G_AB(img_A)
    fake_A = self.G_BA(img_B)
    reconstr_A = self.G_BA(fake_B)
    reconstr_B = self.G_AB(fake_A)
    self.D_A.trainable = False
    self.D_B.trainable = False
    valid_A = self.D_A(fake_A)
    valid_B = self.D_B(fake_B)
    self.AM = Model(inputs=[img_A, img_B],
              outputs=[valid_A, valid_B,
                       reconstr_A, reconstr_B])
    self.AM.compile(loss=['mse', 'mse',
                          'mae', 'mae'],
    )
```

CycleGAN Code - Training

```
def train(self):
    valid = np.ones(...)
    fakes = np.zeros(...)
    for steps in range(self.num_steps):
        imgs_A, imgs_B = self.get_batch_data()
        fake_B = self.G_AB.predict(imgs_A)
        fake_A = self.G_BA.predict(imgs_B)
        D_A_loss_real = self.D_A.train_on_batch(imgs_A, valid)
        D_A_loss_fake = self.D_A.train_on_batch(fake_A, fake)
        D_B_loss_real = self.D_B.train_on_batch(imgs_B, valid)
        D_B_loss_fake = self.D_B.train_on_batch(fake_B, fake)
        total_disc_loss = 0.5 * np.add(D_A_loss_real\
                D_A_loss_fake, D_B_loss_real, D_B_loss_fake)
        G_loss = self.AM.train_on_batch([imgs_A, imgs_B],
                                        [valid, valid,
                                        imgs_A, imgs_B])
        if step%10==0:
            self.save_gen_imgs(...)
```

If you read recent GAN papers, they mostly deal with

- Confronting the non-convergence problem.
- Mode Collapse - Lacking variation in generated images.
- Problems with counting, perspective, global structure.
- Evaluation of generative models is another challenge altogether.
- Most current papers have a reconstruction loss combined with a custom loss.

Tips for implementing GAN papers

- Convergence issues - As a beginner, this is the first thing you'll face as I did. Don't lose hope ! Ask for help on reddit / stackoverflow.
- If you have to make changes, do small changes first, viz. a new loss function or an optimizer change.
- Nobody knows whether Adam works best or RMSProp does or SGD with Nesterov momentum. So try them all but mostly Adam works.
- Normally refrain from making changes to the suggested generator and discriminator networks from as suggested in the paper.
- The good thing is these days, they publish all the hyperparameters in the paper, so you don't have to look for what works.

References

- Generative Models, CS231N, Staford; http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture13.pdf
- NIPS 2016 Tutorial: Generative Adversarial Networks, Goodfellow; https://arxiv.org/pdf/1701.00160.pdf
- GAN Series (from the beginning to the end), Jonathan Hui;
  https://medium.com/@jonathan_hui/gan-gan-series-2d279f906e7b
  Quickly glazes over multiple topics of GANs, variants and mathematical analysis.
- How to Train a GAN? Tips and tricks to make GANs work; https://github.com/soumith/ganhacks
  My personal favorite guide for things to try to make GAN converge.
- Up-sampling with Transposed Convolution, Naoki Shibuya;
  https://towardsdatascience.com/up-sampling-with-transposed-convolution-9ae4f2df52d0
  Helpful to understand the “Deconvolution” operation
- Deconvolution and Checkerboard Artifacts - Chris Olah, https://distill.pub/2016/deconv-checkerboard/
  This explains why most GANs have checkerboard artifacts.
- GAN-Lab, https://github.com/piyush-kgp/GAN-Lab/

Thank You!!
Happy Holi.