# Trustworthy AI Systems

## -- Generative Modeling (Part I)

Instructor: Guangjing Wang

guangjingwang@usf.edu

# Last Lecture

- Semantic Segmentation

- Object Detection
  - R-CNN series
  - YOLO series

- Instance Segmentation

# This Lecture

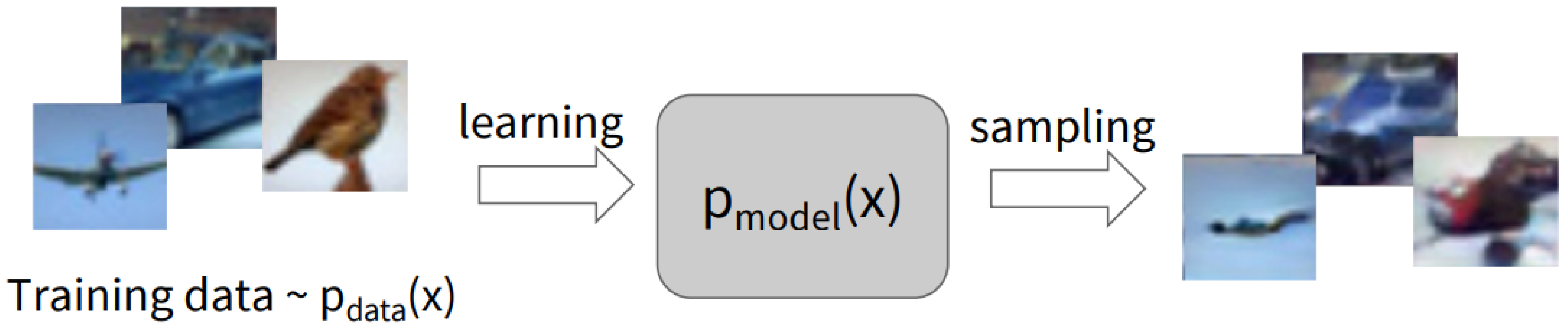
- Generative Modeling

- Generative Adversarial Network
  - DCGAN
  - Conditional GAN
  - CycleGAN

- Neural Style Transfer

# Generative Modeling

Given training data, generate new samples from same distribution




Objectives:

1. Learn $p_{model}(x)$ that approximates $p_{data}(x)$
2. Sampling new x from $p_{model}(x)$

# Learn Data Distributions

- Minimizing some divergence metrics between the training data distribution, and the distribution that the model learns.

- Training models that maximize the expected log likelihood of $p_\theta(x)$

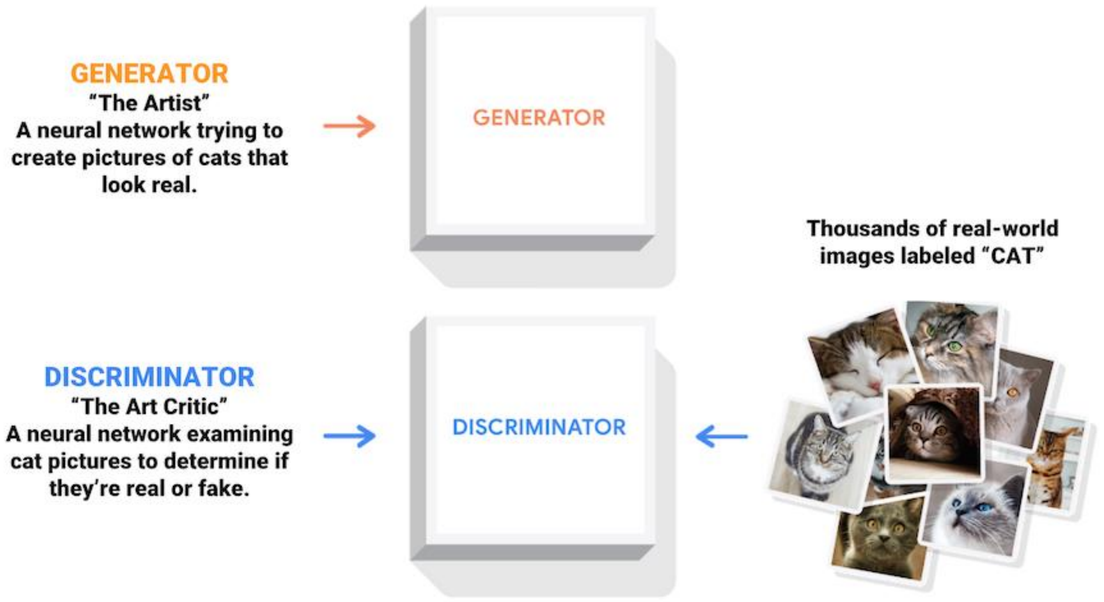
  If I sample from the distribution and get a
  - high likelihood ➔ likely the sample came from the training distribution
  - low likelihood ➔ the sample probably didn't come from the training distribution

# Why Generative Modeling?

- Realistic samples for artwork, super-resolution, colorization, etc.
- Learn useful features for downstream tasks such as classification.
- Getting insights from high-dimensional data (physics, medical imaging, etc.)
- Modeling physical world for simulation and planning (robotics and reinforcement learning applications)
- Many more …

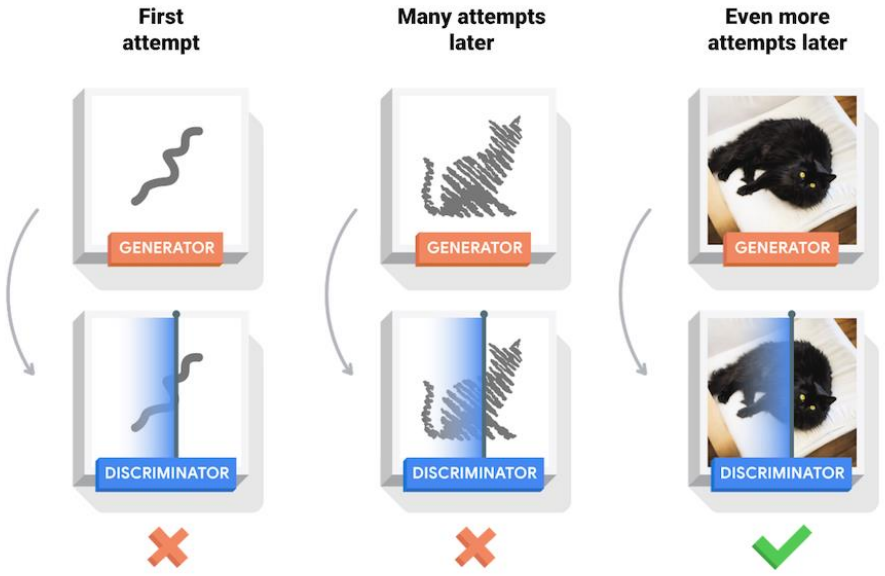
# Generative Adversarial Network (GAN)

- Two models are trained simultaneously by an adversarial process.
  - A generator ("the artist") learns to create images that look real
  - A discriminator ("the art critic") learns to tell real images apart from fakes.



# The idea of GAN

- During training, the generator progressively becomes better at creating images that look real, while the discriminator becomes better at telling them apart.
- The process reaches equilibrium when the *discriminator* can no longer distinguish real images from fakes.



# Deep Convolutional GAN (DCGAN)

- Generator: Upsampling layers (???) to produce an image from a seed (**random noise**)

```
def make_generator_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(7*7*256, use_bias=False, input_shape=(100,)))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Reshape((7, 7, 256)))
    assert model.output_shape == (None, 7, 7, 256)  # Note: None is the batch size

    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', use_bias=False))
    assert model.output_shape == (None, 7, 7, 128)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use_bias=False))
    assert model.output_shape == (None, 14, 14, 64)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())

    model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same', use_bias=False, activation=
    assert model.output_shape == (None, 28, 28, 1)

    return model
```

# Deep Convolutional GAN (DCGAN)

- Discriminator: a classifier

```
def make_discriminator_model():
    model = tf.keras.Sequential()
    model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',
                                     input_shape=[28, 28, 1]))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))

    model.add(layers.Flatten())
    model.add(layers.Dense(1))

    return model
```

# Deep Convolutional GAN (DCGAN)

- Loss function: optimization goal
  - Discriminator loss: how well the discriminator is able to distinguish real images from fakes
  - Generator loss: how well it was able to trick the discriminator

```
def discriminator_loss(real_output, fake_output):
    real_loss = cross_entropy(tf.ones_like(real_output), real_output)
    fake_loss = cross_entropy(tf.zeros_like(fake_output), fake_output)
    total_loss = real_loss + fake_loss
    return total_loss

def generator_loss(fake_output):
    return cross_entropy(tf.ones_like(fake_output), fake_output)
```

# Deep Convolutional GAN (DCGAN)

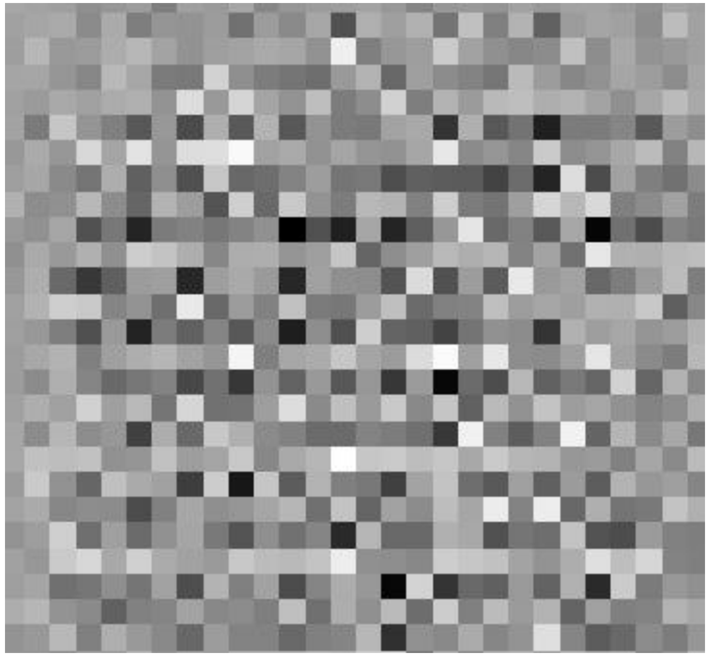
Architecture guidelines for stable Deep Convolutional GANs

- 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.

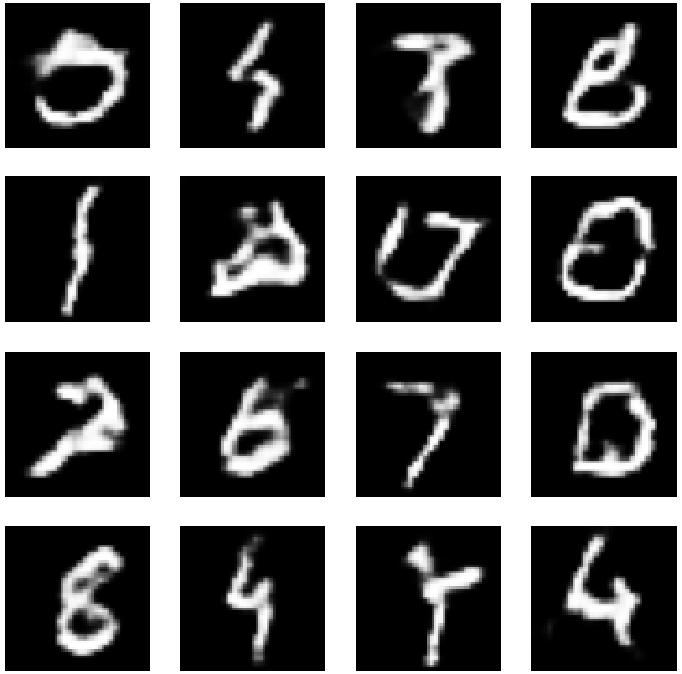
The GAN Zoo: https://github.com/hindupuravinash/the-gan-zoo
Tricks to make GAN better: https://github.com/soumith/ganhacks

Radford et al, "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks", ICLR 2016

# Effect of DCGAN



Start from: Random Noise



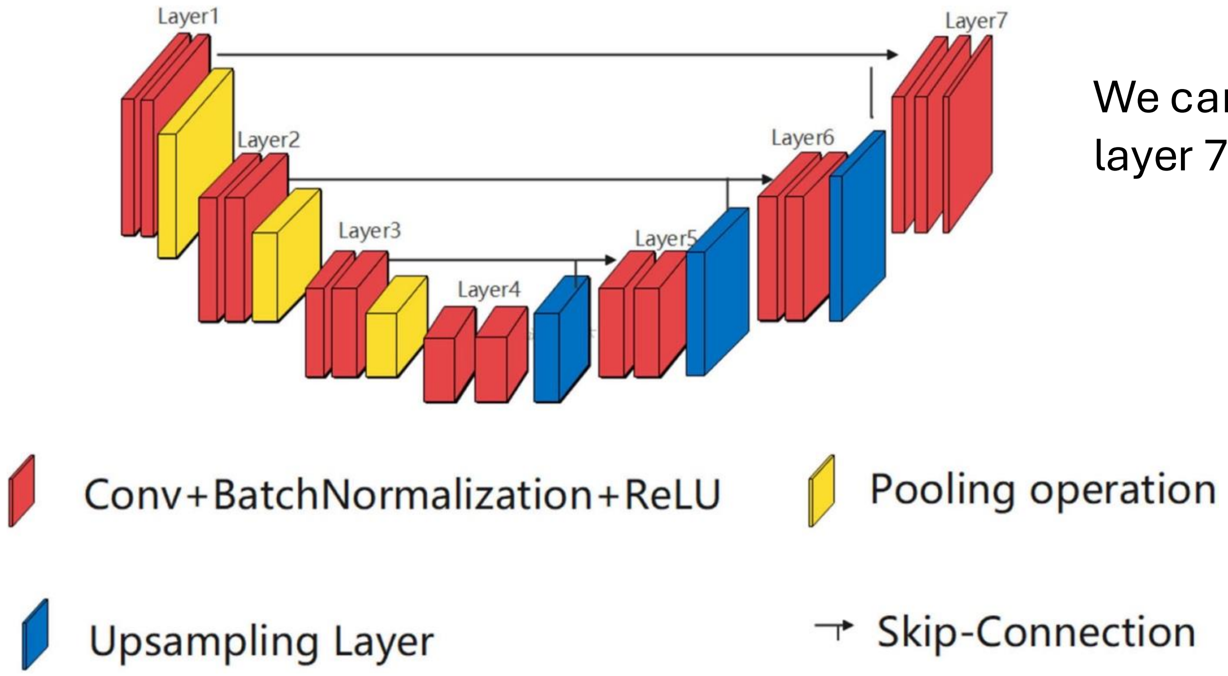
Synthesized Image

# Conditional GAN (cGAN)

- Pix2pix: Learns a mapping from input images to output images
- cGAN: Condition on input images and generate corresponding output images

# Conditional GAN (cGAN)

- Generator (UNet): an encoder (downsampler) and decoder (upsampler)

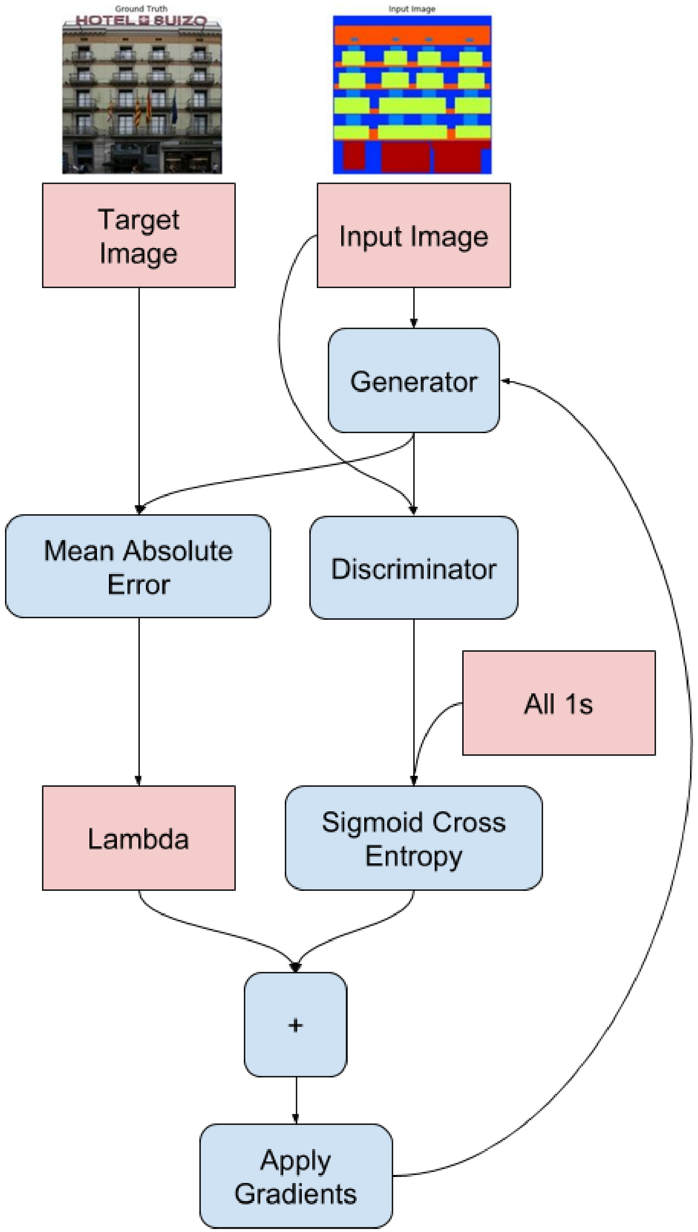



We can set the size of layer 7 the same as input

https://www.frontiersin.org/journals/aging-neuroscience/articles/10.3389/fnagi.2022.841297/full

# Training of Generator

```
def generator_loss(disc_generated_output, gen_output, target):
  gan_loss = loss_object(tf.ones_like(disc_generated_output), disc_generated_output)

  # Mean absolute error
  l1_loss = tf.reduce_mean(tf.abs(target - gen_output))

  total_gen_loss = gan_loss + (LAMBDA * l1_loss)

  return total_gen_loss, gan_loss, l1_loss
```

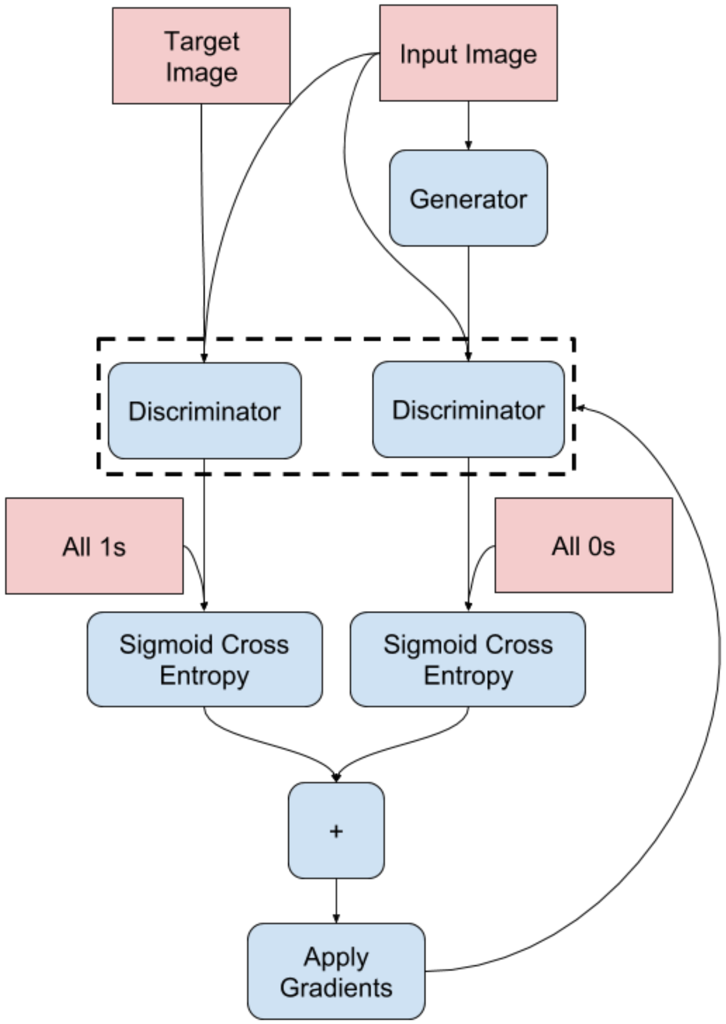


# Discriminator in cGAN

- Discriminator: a convolutional PatchGAN classifier—it tries to classify if each image *patch* is real or fake.
- The input image and the target image, which it should classify as real.
- The input image and the generated image (the output of the generator), which it should classify as fake.

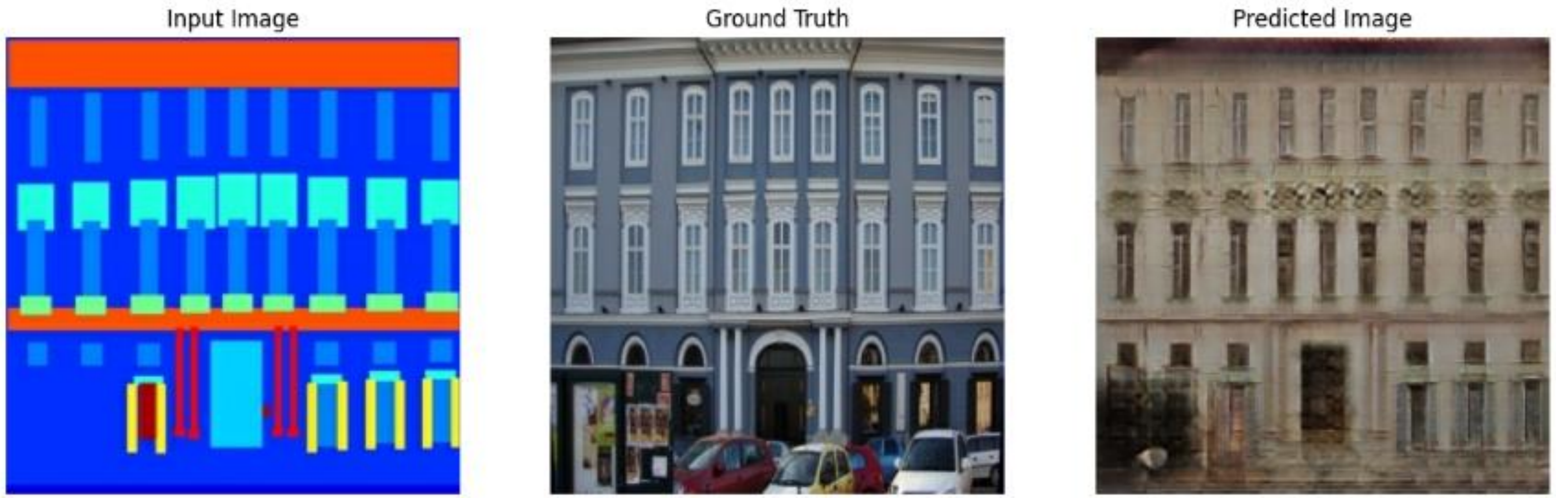
# Train of Discriminator

```
def discriminator_loss(disc_real_output, disc_generated_output):
  real_loss = loss_object(tf.ones_like(disc_real_output), disc_real_output)

  generated_loss = loss_object(tf.zeros_like(disc_generated_output), disc_generated_output)

  total_disc_loss = real_loss + generated_loss

  return total_disc_loss
```



# Effect of cGAN (Pixel2Pixel)

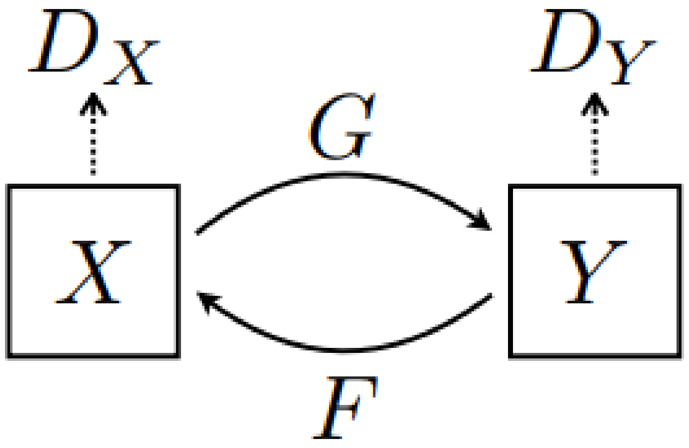
- Pass images from the test set to the generator.
- The generator will then translate the input image into the output.




https://www.tensorflow.org/tutorials/generative/pix2pix

# CycleGAN

There are 2 generators (G and F) and 2 discriminators (X and Y) being trained here.



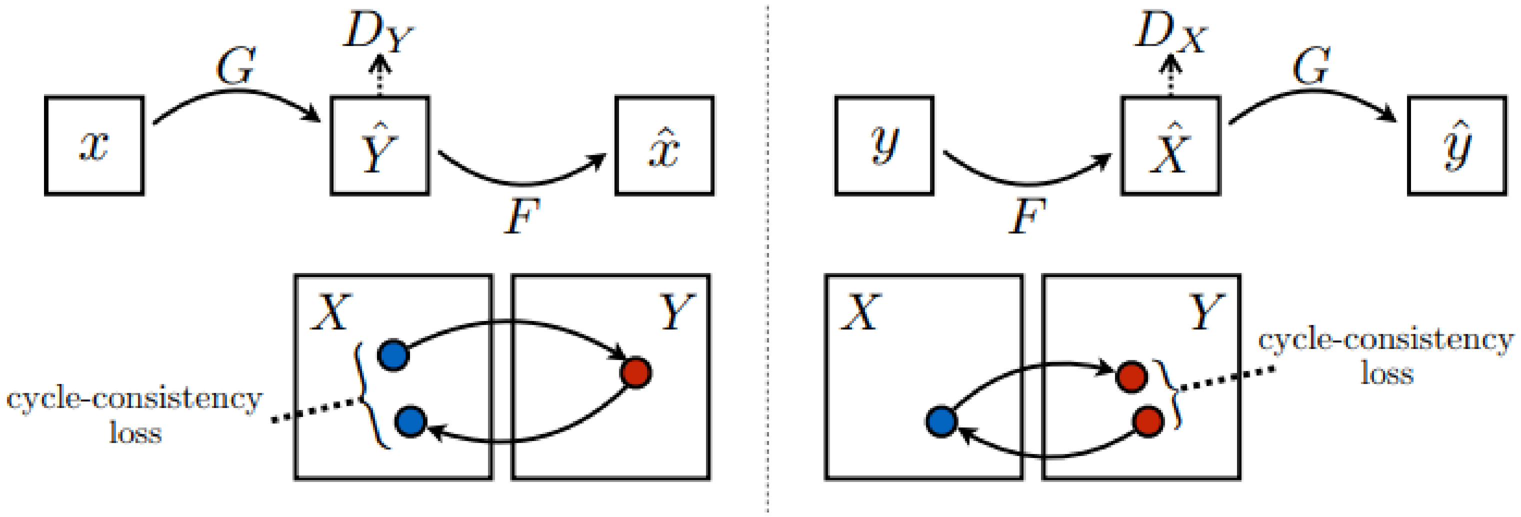
# CycleGAN

- Pixel2Pixel needs paired training data.
- CycleGAN: unpaired training data.

- CycleGAN uses instance normalization instead of batch normalization.
- The CycleGAN paper uses a modified Resnet based Generator

# Loss Function in CycleGAN

- There is no pair data to train on, so cycle consistency loss is designed to enforce the network to learn meaningful mapping.
- Cycle consistency means the result should be close to the original input.



# Feature Inversion

Given a CNN feature vector for an image, find a new image that:

- Matches the given feature vector
- "looks natural" (image prior regularization)

$$\mathbf{x}^* = \underset{\mathbf{x}\in\mathbb{R}^{H\times W\times C}}{\operatorname{argmin}} \ \ell(\Phi(\mathbf{x}), \Phi_0) + \lambda\mathcal{R}(\mathbf{x})$$

Given feature vector

Features of new image

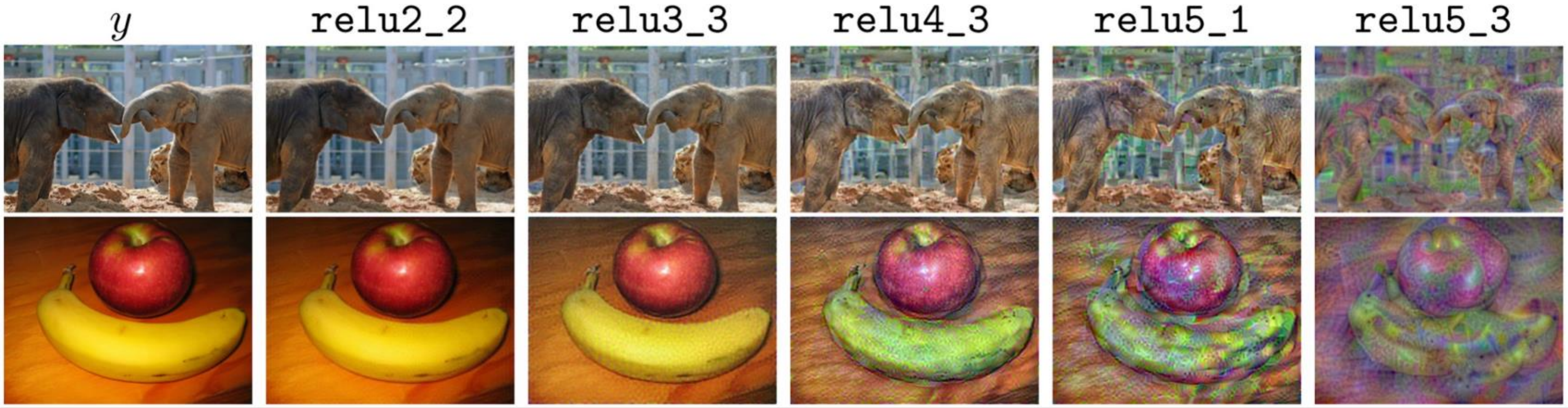
$$\ell(\Phi(\mathbf{x}), \Phi_0) = \|\Phi(\mathbf{x}) - \Phi_0\|^2$$

$$\mathcal{R}_{V^\beta}(\mathbf{x}) = \sum_{i,j}\left((x_{i,j+1} - x_{ij})^2 + (x_{i+1,j} - x_{ij})^2\right)^{\frac{\beta}{2}}$$

Total Variation regularizer
(encourages spatial smoothness)

# Feature Inversion

Potential privacy issues in deep learning

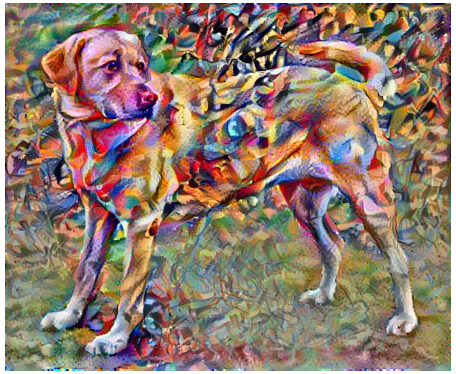


# Neural Style Transfer

- A *content image* and a *style reference image* (such as an artwork by a famous painter)
- Blend them together so the output image looks like the content image, but "painted" in the style of the style reference image.

Content Image

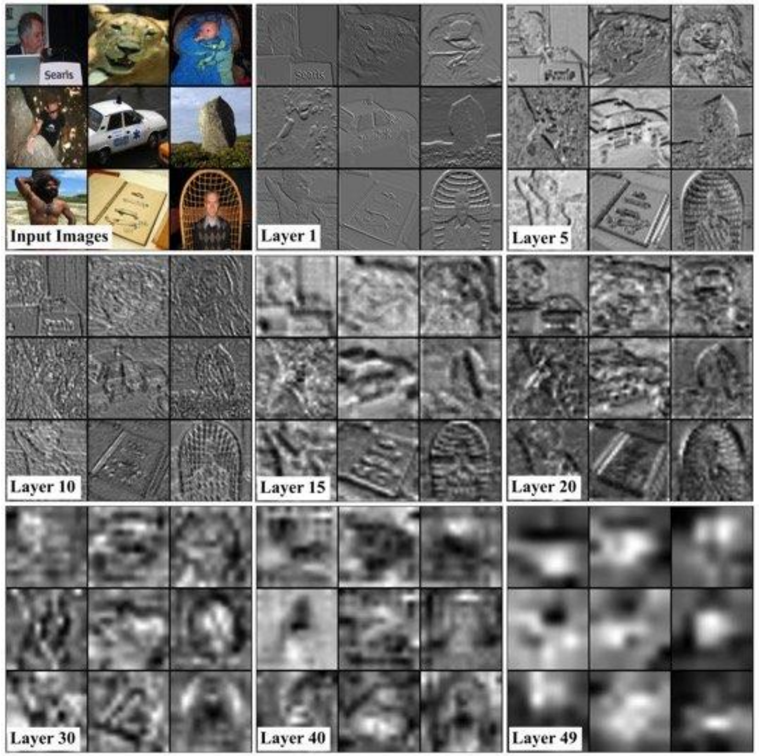
Style Image



Synthesized Image

# Content and Style Representations

- Use the intermediate layers of the model to get the *content* and *style* representations of the image.




https://www.researchgate.net/figure/Visualization-of-example-features-of-layers-1-10-20-30-40-and-49-of-a-deep_fig1_319622441

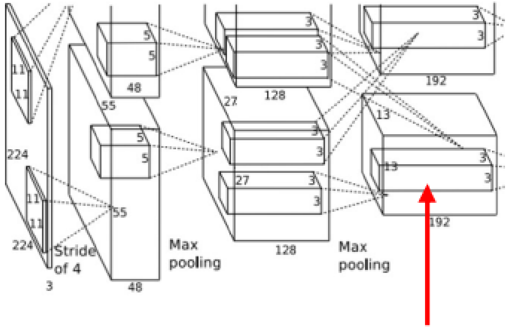
# Content and Style Representations

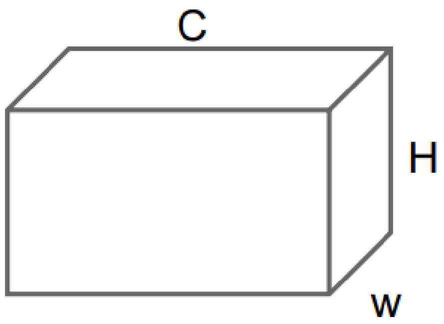
- The content of an image is represented by the values of the intermediate feature maps.

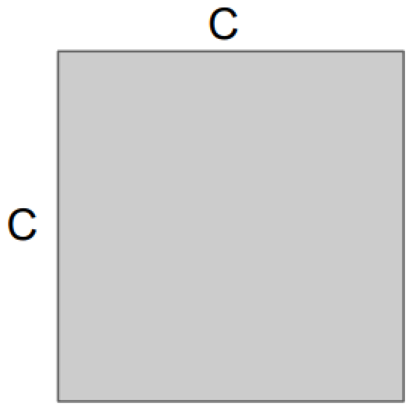
- The style of an image can be described by the means and correlations across the different feature maps.

# Style Representation: Gram Matrix

This image is in the public domain.




Each layer of CNN gives C x H x W tensor of features; H x W grid of C-dimensional vectors

Outer product of two C-dimensional vectors gives C x C matrix measuring co-occurrence

Average over all HW pairs of vectors, giving **Gram matrix** of shape C x C

Efficient to compute; reshape features from

C x H x W to =C x HW

then compute $G = FF^T$

Ignore the positions of features and get correlations among features.

# Style Representation: Gram Matrix

The Gram Matrix takes the outer product of the feature vector with itself at each location and averaging that outer product over all locations.

$$G^l_{cd} = \frac{\sum_{ij} F^l_{ijc}(x) F^l_{ijd}(x)}{IJ}$$

```python
def gram_matrix(input_tensor):
  result = tf.linalg.einsum('bijc,bijd->bcd', input_tensor, input_tensor)
  input_shape = tf.shape(input_tensor)
  num_locations = tf.cast(input_shape[1]*input_shape[2], tf.float32)
  return result/(num_locations)
```

Tensorflow Implementation

```python
def gram_matrix(input):
    a, b, c, d = input.size()  # a=batch size(=1)
    # b=number of feature maps
    # (c,d)=dimensions of a f. map (N=c*d)

    features = input.view(a * b, c * d)  # resize F_XL into \hat F_XL

    G = torch.mm(features, features.t())  # compute the gram product

    # we 'normalize' the values of the gram matrix
    # by dividing by the number of element in each feature maps.
    return G.div(a * b * c * d)
```

Pytorch Implementation

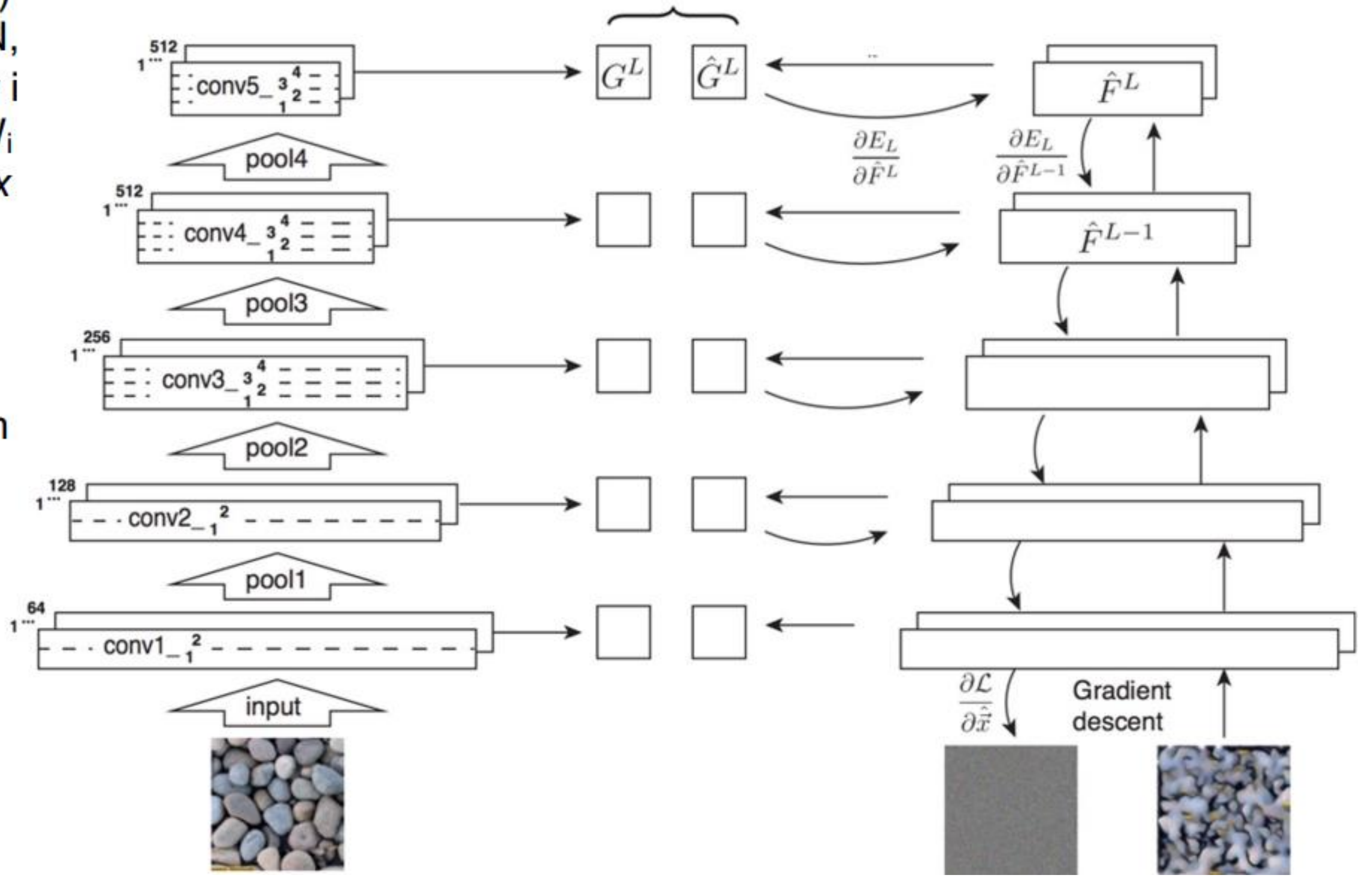
# Neural Style Transfer

$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} \left( G_{ij}^l - \hat{G}_{ij}^l \right)^2 \qquad \mathcal{L}(\vec{x}, \hat{\vec{x}}) = \sum_{l=0}^{L} w_l E_l$$

1. Pretrain a CNN on ImageNet (VGG-19)
2. Run input texture forward through CNN, record activations on every layer; layer i gives feature map of shape $C_i \times H_i \times W_i$
3. At each layer compute the *Gram matrix* giving outer product of features:

$$G_{ij}^l = \sum_k F_{ik}^l F_{jk}^l \text{ (shape } C_i \times C_i)$$

4. Initialize generated image from random noise
5. Pass generated image through CNN, compute Gram matrix on each layer
6. Compute loss: weighted sum of L2 distance between Gram matrices
7. Backprop to get gradient on image
8. Make gradient step on image
9. GOTO 5



# Learning Objective: MSE loss

```
def style_content_loss(outputs):
    style_outputs = outputs['style']
    content_outputs = outputs['content']
    style_loss = tf.add_n([tf.reduce_mean((style_outputs[name]-style_targets[name])**2)
                           for name in style_outputs.keys()])
    style_loss *= style_weight / num_style_layers

    content_loss = tf.add_n([tf.reduce_mean((content_outputs[name]-content_targets[name])**2)
                             for name in content_outputs.keys()])
    content_loss *= content_weight / num_content_layers
    loss = style_loss + content_loss
    return loss
```

Tensorflow Implementation

```
@tf.function()
def train_step(image):
  with tf.GradientTape() as tape:
    outputs = extractor(image)
    loss = style_content_loss(outputs)

  grad = tape.gradient(loss, image)
  opt.apply_gradients([(grad, image)])
  image.assign(clip_0_1(image))
```

Pytorch Implementation

```
# We want to optimize the input and not the model parameters so we
# update all the requires_grad fields accordingly
input_img.requires_grad_(True)
# We also put the model in evaluation mode, so that specific layers
# such as dropout or batch normalization layers behave correctly.
model.eval()
model.requires_grad_(False)

optimizer = get_input_optimizer(input_img)
```

# Neural Style Transfer

# References

- https://cs231n.stanford.edu/slides/2024/lecture_11.pdf
- https://www.tensorflow.org/tutorials/generative/style_transfer
- https://pytorch.org/tutorials/advanced/neural_style_tutorial.html
- https://www.tensorflow.org/tutorials/generative/dcgan
- https://www.tensorflow.org/tutorials/generative/pix2pix
- https://www.tensorflow.org/tutorials/generative/cyclegan