Generative Deep Learning

Prof. Kuan-Ting Lai

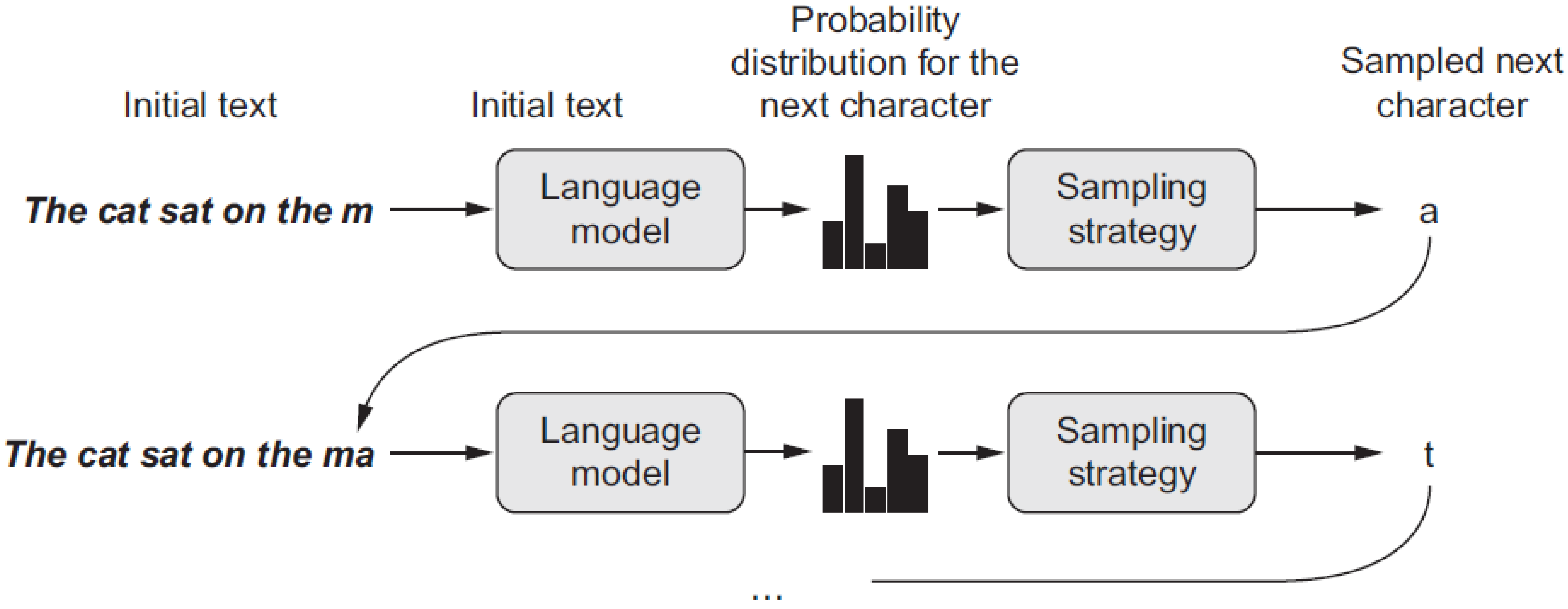
2020/5/12

DeepFake (Intro)

Generative Recurrent Networks

- Douglas Eck (2002), Music Generation using LSTM
- Alex Graves, “Generating Sequences With Recurrent Neural Networks,” arXiv (2013), https://arxiv.org/abs/1308.0850.

Text Generation with LSTM



Sampling Strategy

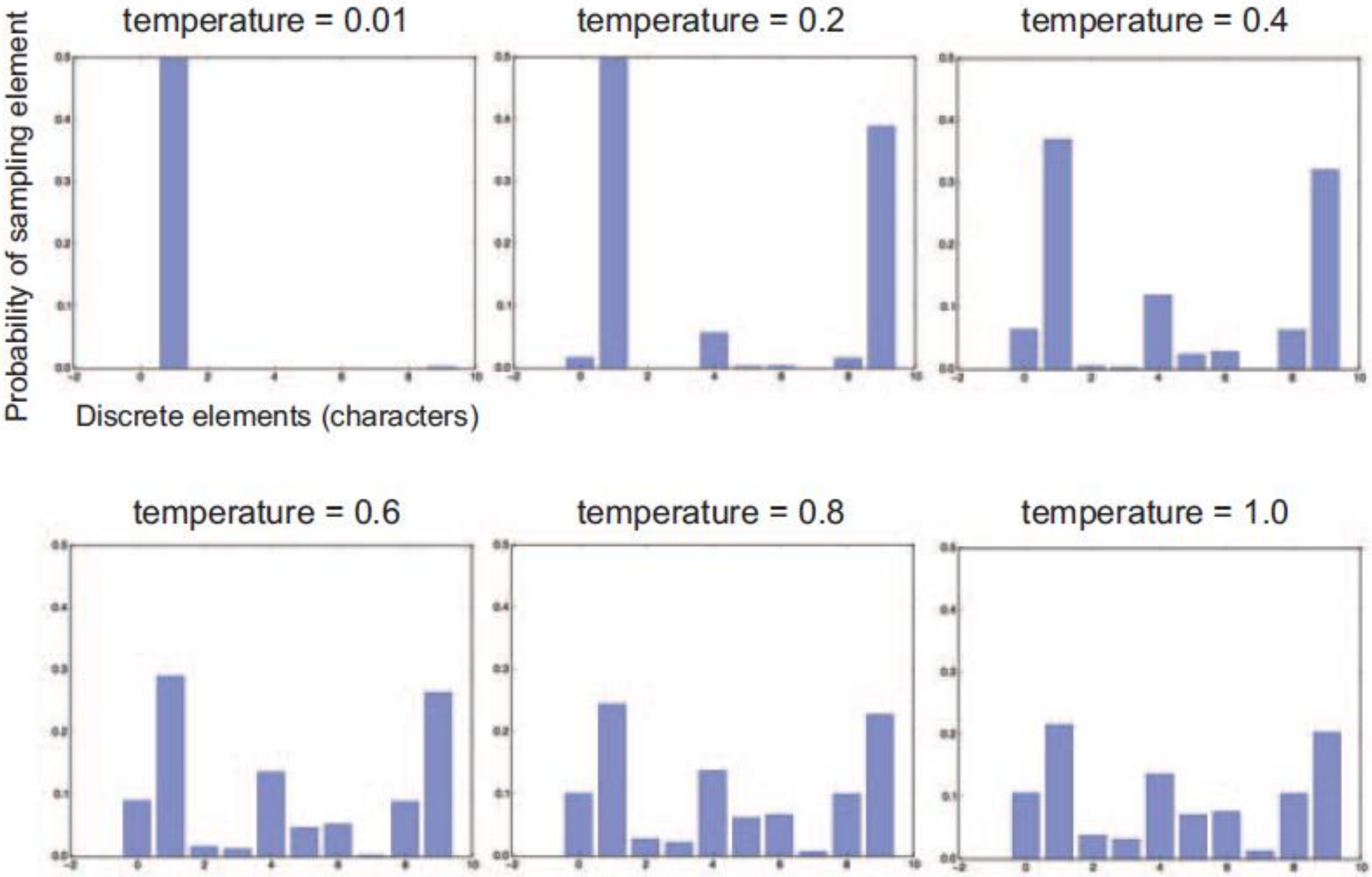
- Greedy sampling: select the one with highest possibility
- Stochastic sampling
- More randomness -> more surprises

Temperature

- Reweighting a probability distribution

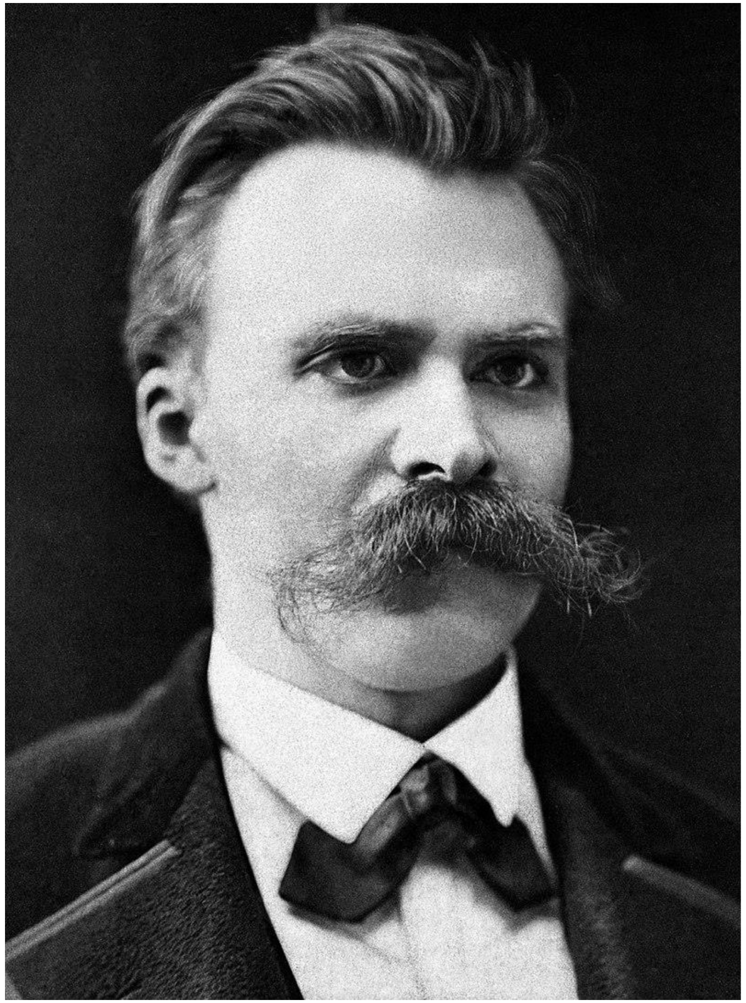
```
import numpy as np
def reweight_distribution(original_distribution, temperature=0.5):
    distribution = np.log(original_distribution) / temperature
    distribution = np.exp(distribution)
    return distribution / np.sum(distribution)
```

Higher Temperature = More Randomness



Generating Text of Nietzsche

- *That which does not kill us makes us stronger.*
- *Man is the cruelest animal.*
- *Sometimes people don't want to hear the truth because they don't want their illusions destroyed.*
- *The true man wants two things: danger and play. For that reason he wants woman, as the most dangerous plaything.*



Character-level LSTM Text Generation

- Download training data
- Things to note:
  - At least 20 epochs are required before the generated text starts sounding coherent.
  - If you try this script on new data, make sure your corpus
  - has at least ~100k characters. ~1M is better.

```
import keras
import numpy as np

path = keras.utils.get_file(
    'nietzsche.txt',
    origin='https://s3.amazonaws.com/text-datasets/nietzsche.txt')
text = open(path).read().lower()
print('Corpus length:', len(text))
```

https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/8.1-text-generation-with-lstm.ipynb

Convert Characters into Indices

- 57 unique characters in the data

```
chars = sorted(list(set(text)))
print('total chars:', len(chars))
char_indices = dict((c, i) for i, c in enumerate(chars))
indices_char = dict((i, c) for i, c in enumerate(chars))
```

```
(tf2) PS C:\Users\kuant\OneDrive\Teaching\108-2深度學習應用開發實務\code> python .\lstm_text_generation.py
2020-05-09 18:30:19.774540: I tensorflow/stream_executor/platform/default/dso_loader.cc:44] Successfully opened dynamic library cudart64_
Using TensorFlow backend.
corpus length: 600893
total chars: 57
nb sequences: 200285
Vectorization...
Build model...
```

Vectorizing Sequences of Characters

```
maxlen = 60
step = 3
sentences = []
next_chars = []
for i in range(0, len(text) - maxlen, step):
    sentences.append(text[i: i + maxlen])
    next_chars.append(text[i + maxlen])
print('Number of sequences:', len(sentences))
chars = sorted(list(set(text)))
print('Unique characters:', len(chars))
char_indices = dict((char, chars.index(char)) for char in chars)
print('Vectorization...')
x = np.zeros((len(sentences), maxlen, len(chars)), dtype=np.bool)
y = np.zeros((len(sentences), len(chars)), dtype=np.bool)
for i, sentence in enumerate(sentences):
    for t, char in enumerate(sentence):
        x[i, t, char_indices[char]] = 1
    y[i, char_indices[next_chars[i]]] = 1
```

You'll extract sequences of 60 characters.

You'll sample a new sequence every three characters.

Holds the extracted sequences

Holds the targets (the follow-up characters)

List of unique characters in the corpus

Dictionary that maps unique characters to their index in the list "chars"

One-hot encodes the characters into binary arrays

Building the Network

```
from keras import layers
model = keras.models.Sequential()
model.add(layers.LSTM(128, input_shape=(maxlen, len(chars))))
model.add(layers.Dense(len(chars), activation='softmax'))

optimizer = keras.optimizers.RMSprop(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=optimizer)
```

Training & Sampling the Language Model

1. Drawing from the model a probability distribution over the next character given the text available
2. Reweighting the distribution to a certain "temperature"
3. Sampling the next character at random according to the reweighted distribution
4. Adding the new character at the end of the available text

Sampling Next Characters

```
def sample(preds, temperature=1.0):
    preds = np.asarray(preds).astype('float64')
    preds = np.log(preds) / temperature
    exp_preds = np.exp(preds)
    preds = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, preds, 1)
    return np.argmax(probas)
```

Text-generation Loop

```
import random
import sys

for epoch in range(1, 60):
    print('epoch', epoch)
    model.fit(x, y, batch_size=128, epochs=1)
    start_index = random.randint(0, len(text) - maxlen - 1)
    generated_text = text[start_index: start_index + maxlen]
    print('--- Generating with seed: "' + generated_text + '"')

    for temperature in [0.2, 0.5, 1.0, 1.2]:
        print('------ temperature:', temperature)
        sys.stdout.write(generated_text)
```

Trains the model for 60 epochs

Fits the model for one iteration on the data

Selects a text seed at random

Tries a range of different sampling temperatures

Text-generation Loop (Cont'd)

Generates 400 characters, starting from the seed text

```
for i in range(400):
    sampled = np.zeros((1, maxlen, len(chars)))
    for t, char in enumerate(generated_text):
        sampled[0, t, char_indices[char]] = 1.

    preds = model.predict(sampled, verbose=0)[0]
    next_index = sample(preds, temperature)
    next_char = chars[next_index]

    generated_text += next_char
    generated_text = generated_text[1:]

    sys.stdout.write(next_char)
```

One-hot encodes the characters generated so far

Samples the next character

Results of Epoch 60

```
Epoch 60/60
199936/200285 [============================>.] - ETA: 0s - loss: 1.2384
----- Generating text after Epoch: 59
----- diversity: 0.2
----- Generating with seed: "ange an opinion about any one, we charge"
ange an opinion about any one, we charger and the sense of the factity of the sense of the sense of the
continuation of the sense of the sense of the heart and superstitions, and in the sense of the sense of the
most spirit of the sense of the sense of the sense of the most portentous and as the sense of the sense of the
sense of the sense of the heart and self-distrust of the sense of the sense of the sense of the sense of the
sense of
----- diversity: 0.5
----- Generating with seed: "ange an opinion about any one, we charge"
ange an opinion about any one, we charges and contempleting and self-delight and in the sensive reports in
the portent and morality of the sense of a fainh purpose of the effective century and that struckon and be
conceptions and disposition of them as the sense of the fact that is the sense. the most foreign and the best
and
who has almost science in the people more secret to the survivaling some man the belief in the other hand
```


Deep Dream

Implementing DeepDream in Keras

```
from keras.applications import inception_v3
from keras import backend as K

K.set_learning_phase(0)

model = inception_v3.InceptionV3(weights='imagenet',
                                 include_top=False)
```

You won't be training the model, so this command disables all training-specific operations.

Builds the Inception V3 network, without its convolutional base. The model will be loaded with pretrained ImageNet weights.

Configuring DeepDream

```
layer_contributions = {
    'mixed2': 0.2,
    'mixed3': 3.,
    'mixed4': 2.,
    'mixed5': 1.5,
}
```

Dictionary mapping layer names to a coefficient quantifying how much the layer's activation contributes to the loss you'll seek to maximize. Note that the layer names are hardcoded in the built-in Inception V3 application. You can list all layer names using model.summary().

Defining the Loss

Creates a dictionary that maps layer names to layer instances

```
layer_dict = dict([(layer.name, layer) for layer in model.layers])

loss = K.variable(0.)
for layer_name in layer_contributions:
    coeff = layer_contributions[layer_name]
    activation = layer_dict[layer_name].output

    scaling = K.prod(K.cast(K.shape(activation), 'float32'))
    loss += coeff * K.sum(K.square(activation[:, 2: -2, 2: -2, :])) / scaling
```

You'll define the loss by adding layer contributions to this scalar variable.

Retrieves the layer's output

Adds the L2 norm of the features of a layer to the loss. You avoid border artifacts by only involving nonborder pixels in the loss.

Gradient-ascent Process

This tensor holds the generated image: the dream.

Computes the gradients of the dream with regard to the loss

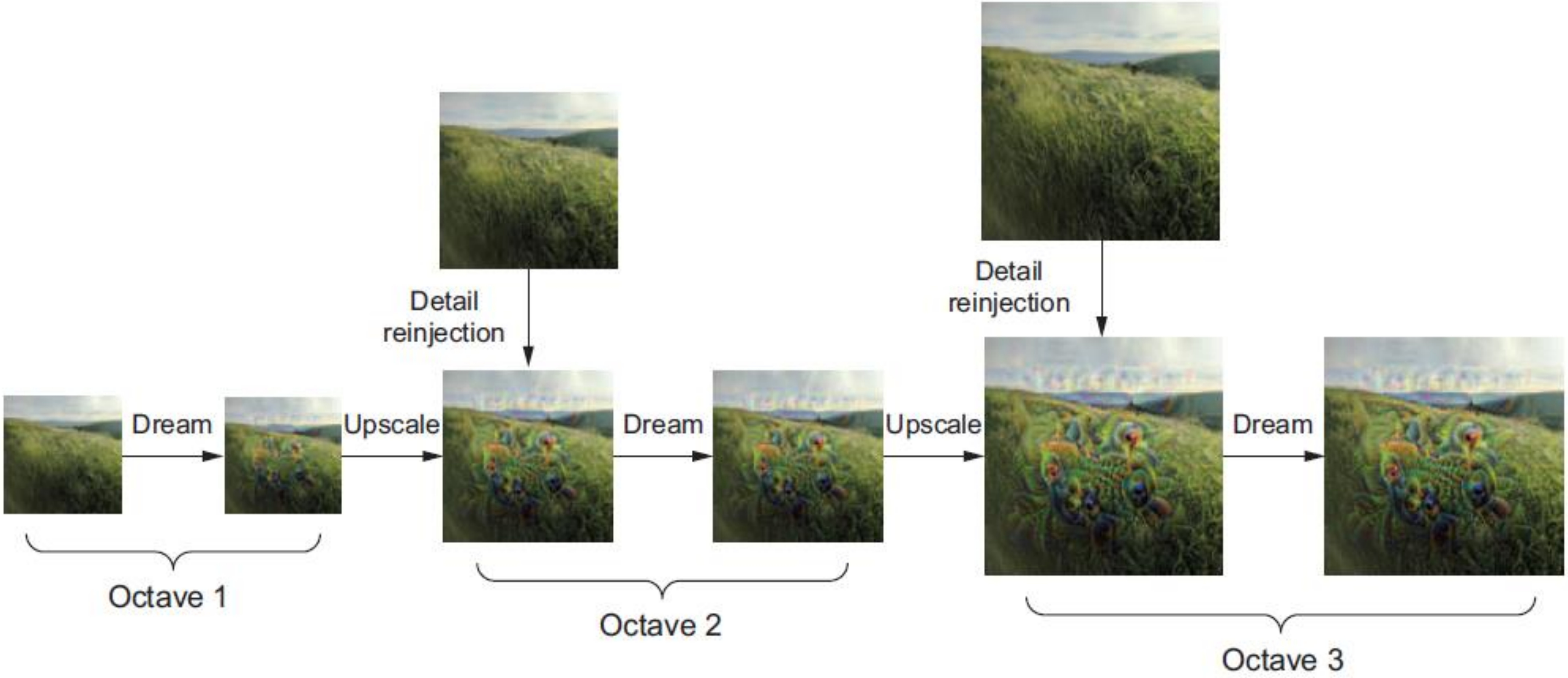
Normalizes the gradients (important trick)

Sets up a Keras function to retrieve the value of the loss and gradients, given an input image

This function runs gradient ascent for a number of iterations.

```
dream = model.input

grads = K.gradients(loss, dream)[0]

grads /= K.maximum(K.mean(K.abs(grads)), 1e-7)

outputs = [loss, grads]
fetch_loss_and_grads = K.function([dream], outputs)

def eval_loss_and_grads(x):
    outs = fetch_loss_and_grads([x])
    loss_value = outs[0]
    grad_values = outs[1]
    return loss_value, grad_values

def gradient_ascent(x, iterations, step, max_loss=None):
    for i in range(iterations):
        loss_value, grad_values = eval_loss_and_grads(x)
        if max_loss is not None and loss_value > max_loss:
            break
        print('...Loss value at', i, ':', loss_value)
        x += step * grad_values
    return x
```

DeepDream Process: Scaling and Detail Reinjection



Running Gradient Ascent over Different Successive Scales

Playing with these hyperparameters will let you achieve new effects.

```
import numpy as np

step = 0.01
num_octave = 3
octave_scale = 1.4
iterations = 20

max_loss = 10.

base_image_path = '...'

img = preprocess_image(base_image_path)
```

Gradient ascent step size

Number of scales at which to run gradient ascent

Size ratio between scales

Number of ascent steps to run at each scale

If the loss grows larger than 10, you'll interrupt the gradient-ascent process to avoid ugly artifacts.

Fill this with the path to the image you want to use.

Loads the base image into a Numpy array (function is defined in listing 8.13)

```python
original_shape = img.shape[1:3]
successive_shapes = [original_shape]
for i in range(1, num_octave):
    shape = tuple([int(dim / (octave_scale ** i))
        for dim in original_shape])
    successive_shapes.append(shape)

successive_shapes = successive_shapes[::-1]

original_img = np.copy(img)
shrunk_original_img = resize_img(img, successive_shapes[0])

for shape in successive_shapes:
    print('Processing image shape', shape)
    img = resize_img(img, shape)
    img = gradient_ascent(img,
                          iterations=iterations,
                          step=step,
                          max_loss=max_loss)
    upscaled_shrunk_original_img = resize_img(shrunk_original_img, shape)
    same_size_original = resize_img(original_img, shape)
    lost_detail = same_size_original - upscaled_shrunk_original_img

    img += lost_detail
    shrunk_original_img = resize_img(original_img, shape)
    save_img(img, fname='dream_at_scale_' + str(shape) + '.png')

save_img(img, fname='final_dream.png')
```

Prepares a list of shape tuples defining the different scales at which to run gradient ascent

Reverses the list of shapes so they're in increasing order

Resizes the Numpy array of the image to the smallest scale

Scales up the dream image

Runs gradient ascent, altering the dream

Scales up the smaller version of the original image: it will be pixellated.

Computes the high-quality version of the original image at this size

The difference between the two is the detail that was lost when scaling up.

Reinjects lost detail into the dream

Neural Style Transfer

- Leon A. Gatys, Alexander S. Ecker, and Matthias Bethge, “A Neural Algorithm of Artistic Style,” arXiv (2015), https://arxiv.org/abs/1508.06576 .

Prisma Photo Editor 12+

Art Filters & Photo Effects

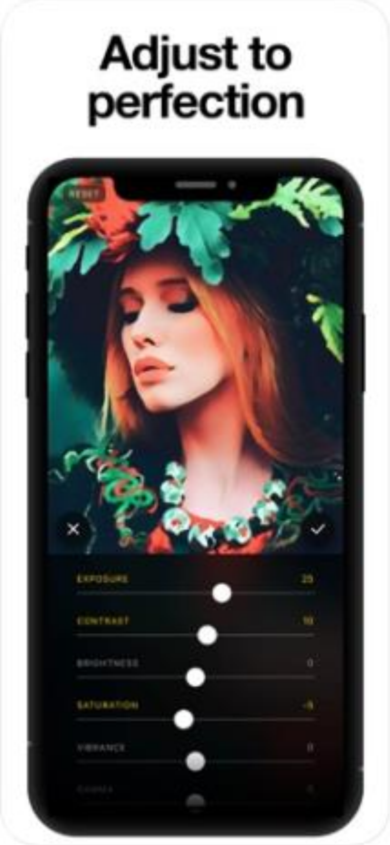
Prisma labs, inc.

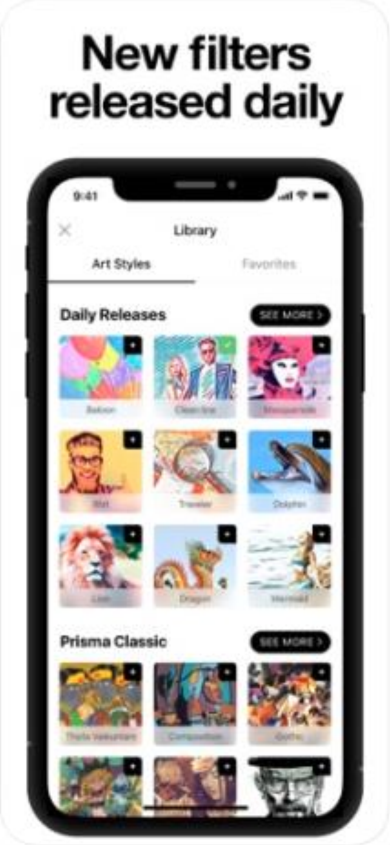
★★★★★ 4.7, 95.5K Ratings

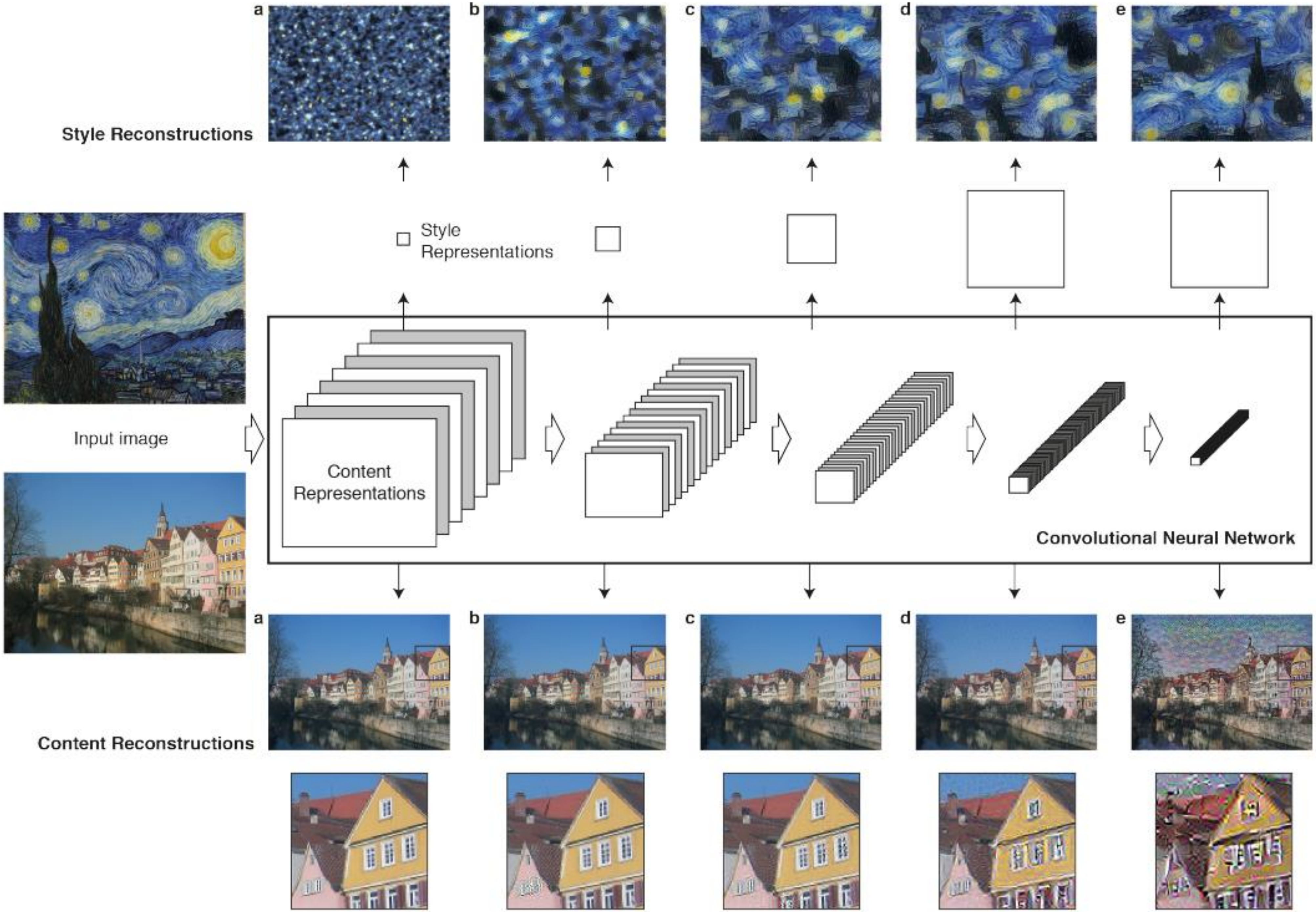
Free · Offers In-App Purchases

Screenshots

iPhone iPad







Style Reconstructions
a
b
c
d
e
Style
Representations
Input image
Content
Representations
Convolutional Neural Network
Content Reconstructions
a
b
c
d
e

Content Loss + Style Loss

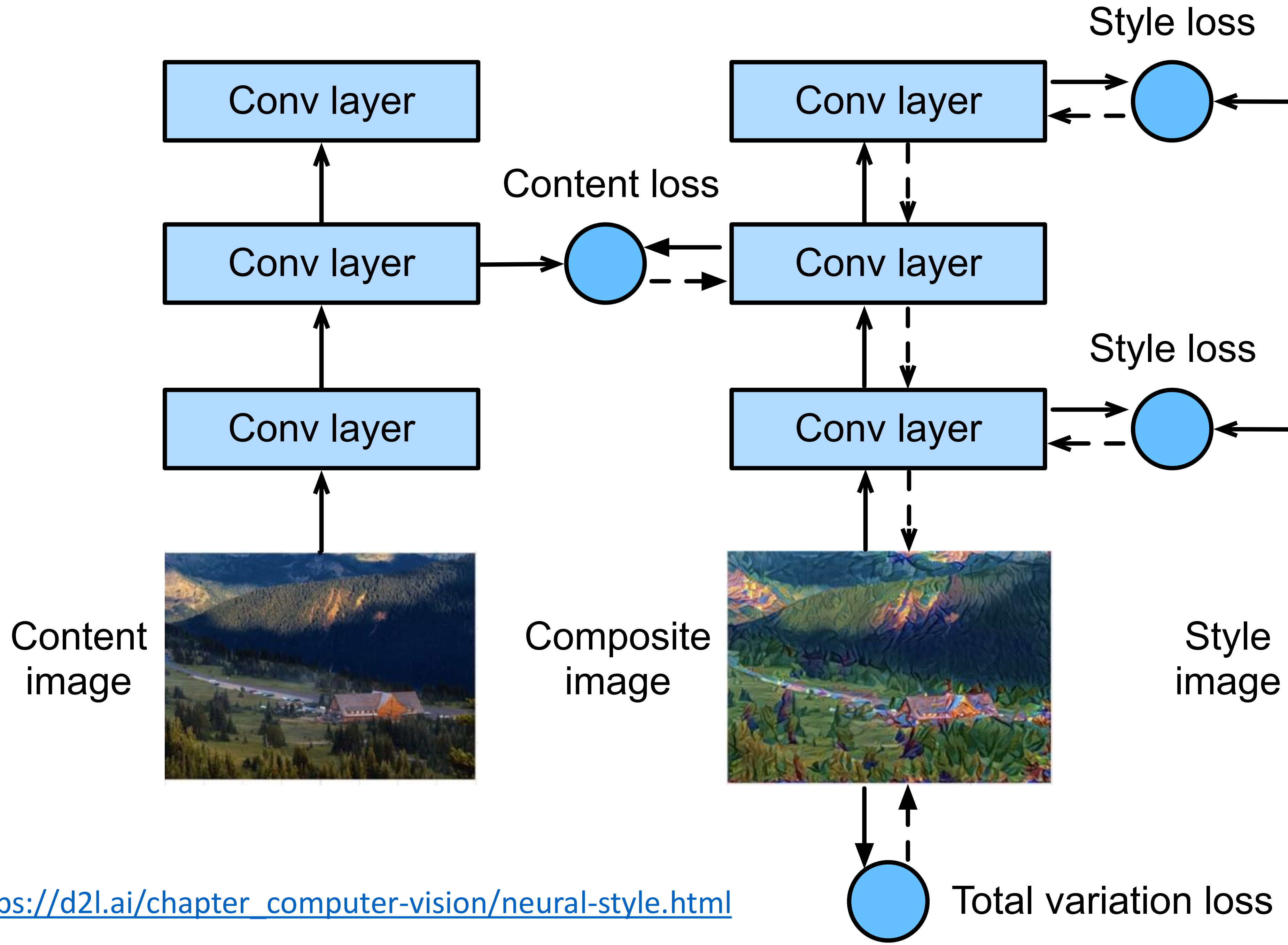
- Using pre-trained model (VGG)
- Content Loss

$$\mathcal{L}_{content}(\vec{p}, \vec{x}, l) = \frac{1}{2} \sum_{i,j} \left(F_{ij}^{l} - P_{ij}^{l}\right)^2$$

- The style representations simply compute the correlations between different convolution layers, correlation is calculated by Gram matrix

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

$$\mathcal{L}_{total}(\vec{p}, \vec{a}, \vec{x}) = \alpha \mathcal{L}_{content}(\vec{p}, \vec{x}) + \beta \mathcal{L}_{style}(\vec{a}, \vec{x})$$

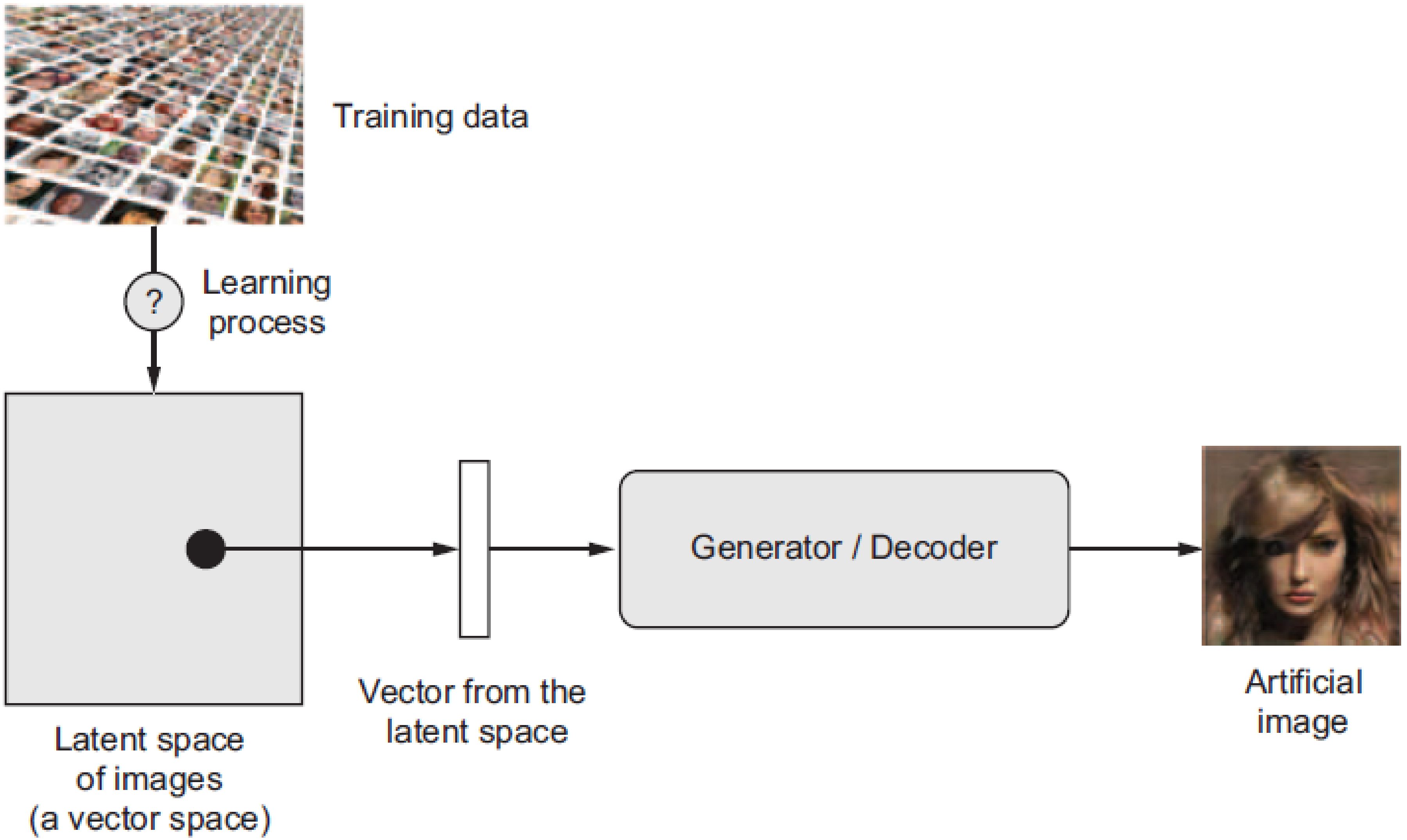


https://d2l.ai/chapter_computer-vision/neural-style.html

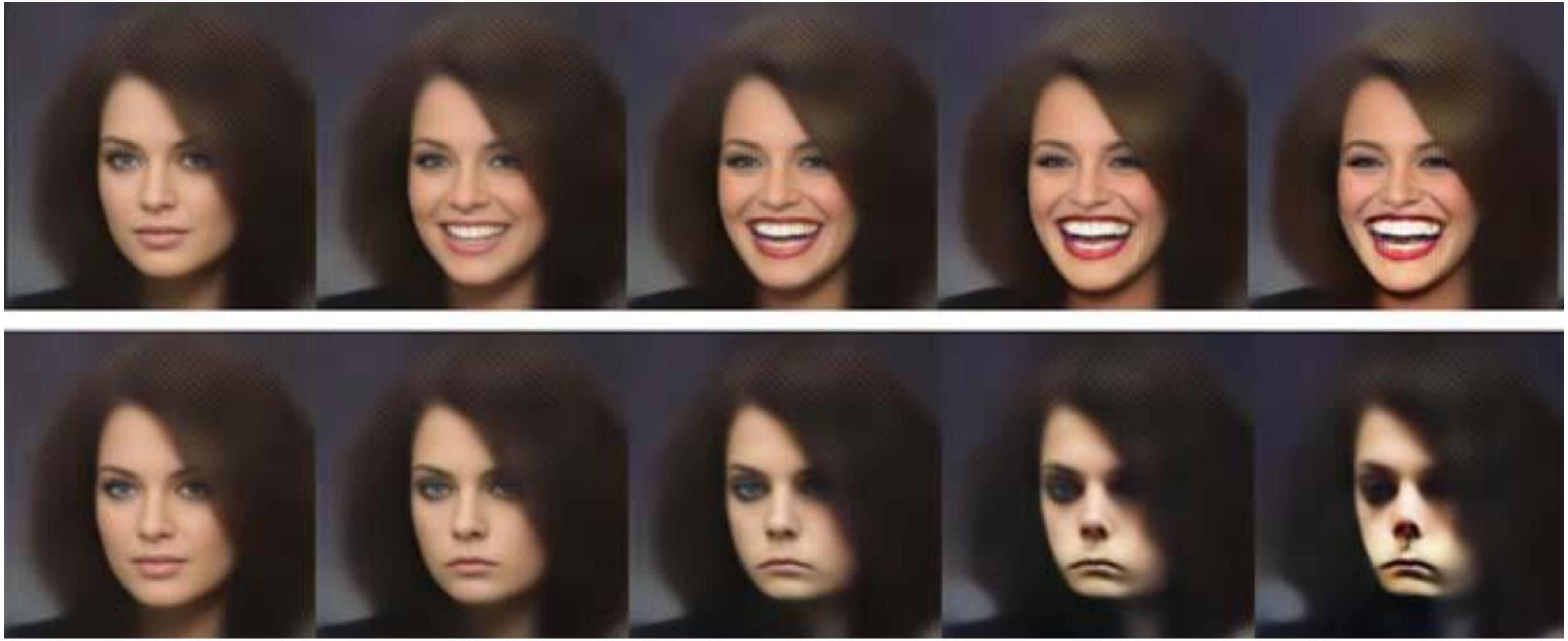
Example

- https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/8.3-neural-style-transfer.ipynb

Generating Images with Variational Auto-encoder

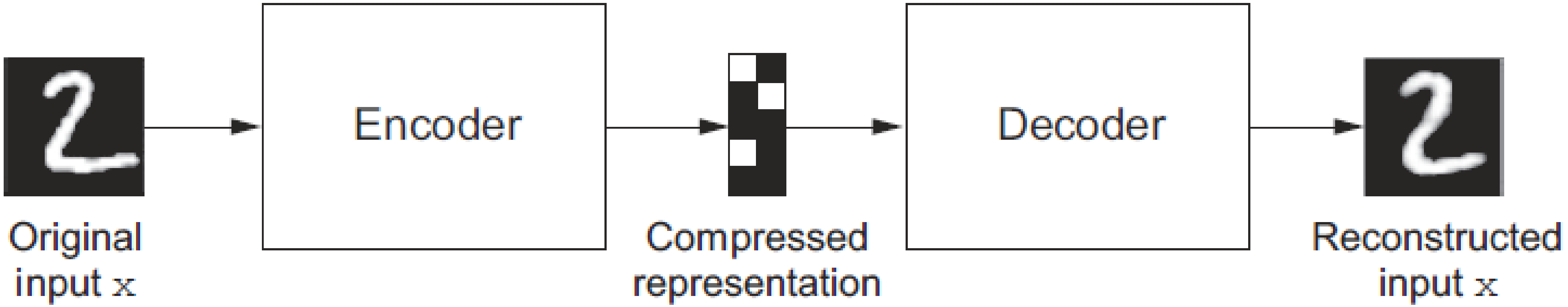


The Smile Vector



Auto-encoder

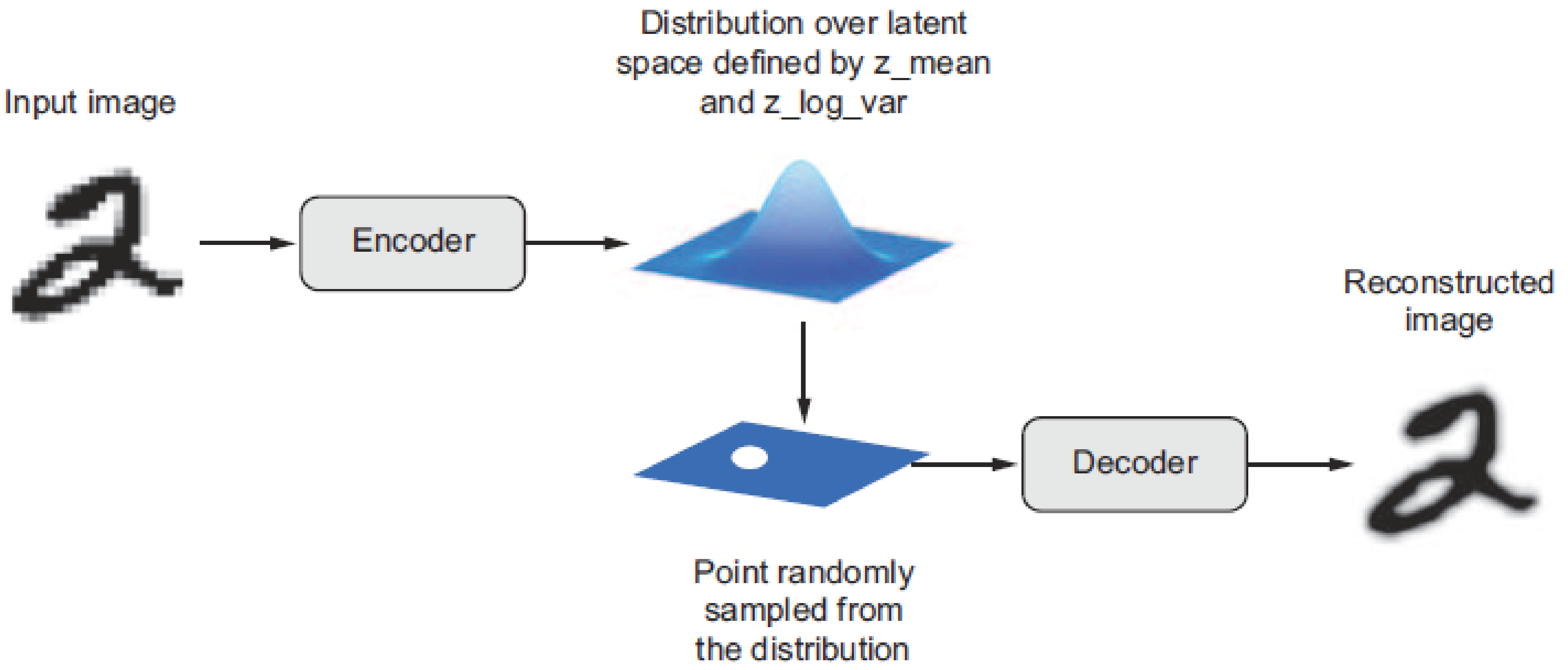
- Learn compressed representation of input x



Variational Auto-encoder

- Assume images are generated by a statistical process
- Randomness of this process is considered during encoding and decoding

https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/8.4-generating-images-with-vaes.ipynb



Pseudo Code of Encode and Decoder

```
# Encode the input into a mean and variance parameter
z_mean, z_log_variance = encoder(input_img)

# Draw a latent point using a small random epsilon
z = z_mean + exp(z_log_variance) * epsilon

# Then decode z back to an image
reconstructed_img = decoder(z)

# Instantiate a model
model = Model(input_img, reconstructed_img)

# Then train the model using 2 losses:
# a reconstruction loss and a regularization loss
```

Encoder

```python
import keras
from keras import layers
from keras import backend as K
from keras.models import Model
import numpy as np

img_shape = (28, 28, 1)
batch_size = 16
latent_dim = 2  # Dimensionality of the latent space: a plane

input_img = keras.Input(shape=img_shape)

x = layers.Conv2D(32, 3, padding='same', activation='relu')(input_img)
x = layers.Conv2D(64, 3, padding='same', activation='relu', strides=(2, 2))(x)
x = layers.Conv2D(64, 3, padding='same', activation='relu')(x)
x = layers.Conv2D(64, 3, padding='same', activation='relu')(x)
shape_before_flattening = K.int_shape(x)

x = layers.Flatten()(x)
x = layers.Dense(32, activation='relu')(x)

z_mean = layers.Dense(latent_dim)(x)
z_log_var = layers.Dense(latent_dim)(x)
```

Sampling

- In Keras, everything needs to be a layer, so code that isn't part of a built-in layer should be wrapped in a Lambda (or else, in a custom layer).

```
def sampling(args):
    z_mean, z_log_var = args
    epsilon = K.random_normal(shape=(K.shape(z_mean)[0], latent_dim),
                              mean=0., stddev=1.)
    return z_mean + K.exp(z_log_var) * epsilon

z = layers.Lambda(sampling)([z_mean, z_log_var])
```

Decoder

```
# This is the input where we will feed `z`.
decoder_input = layers.Input(K.int_shape(z)[1:])

# Upsample to the correct number of units
x = layers.Dense(np.prod(shape_before_flattening[1:]), activation='relu')(decoder_input)

# Reshape into an image of the same shape as before our last `Flatten` layer
x = layers.Reshape(shape_before_flattening[1:])(x)

# We then apply then reverse operation to the initial stack of convolution layers:
# a `Conv2DTranspose` layers with corresponding parameters.
x = layers.Conv2DTranspose(32, 3, padding='same', activation='relu', strides=(2, 2))(x)
x = layers.Conv2D(1, 3, padding='same', activation='sigmoid')(x)

# This is our decoder model.
decoder = Model(decoder_input, x)

# We then apply it to `z` to recover the decoded `z`.
z_decoded = decoder(z)
```

```python
class CustomVariationalLayer(keras.layers.Layer):

    def vae_loss(self, x, z_decoded):
        x = K.flatten(x)
        z_decoded = K.flatten(z_decoded)
        xent_loss = keras.metrics.binary_crossentropy(x, z_decoded)
        kl_loss = -5e-4 * K.mean(
            1 + z_log_var - K.square(z_mean) - K.exp(z_log_var), axis=-1)
        return K.mean(xent_loss + kl_loss)

    def call(self, inputs):
        x = inputs[0]
        z_decoded = inputs[1]
        loss = self.vae_loss(x, z_decoded)
        self.add_loss(loss, inputs=inputs)
        # We don't use this output.
        return x

# We call our custom layer on the input and the decoded output,
# to obtain the final model output.
y = CustomVariationalLayer()([input_img, z_decoded])
```

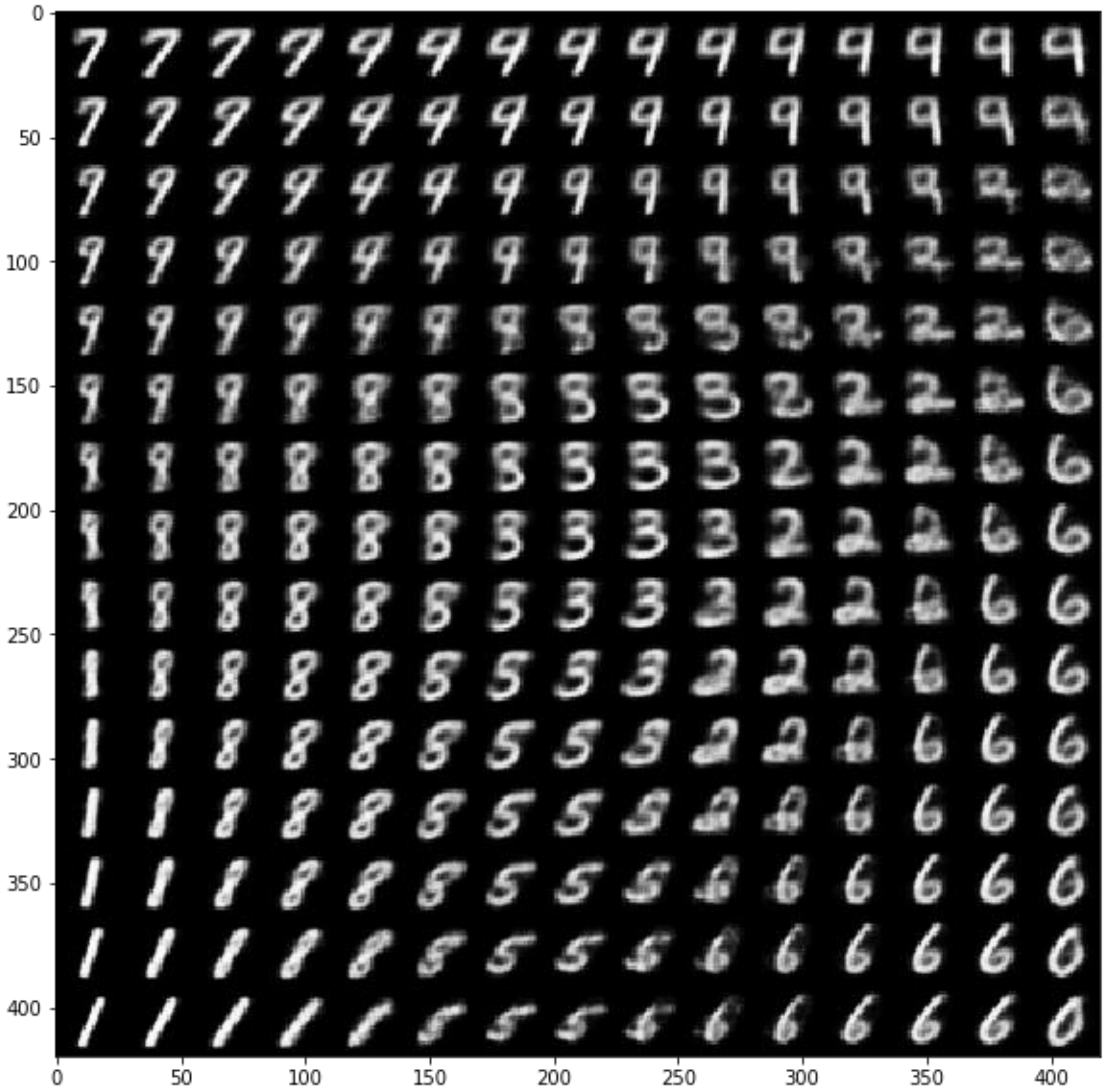
Training VAE

- We don’t pass target data during training (only pass x_train to the model in fit)

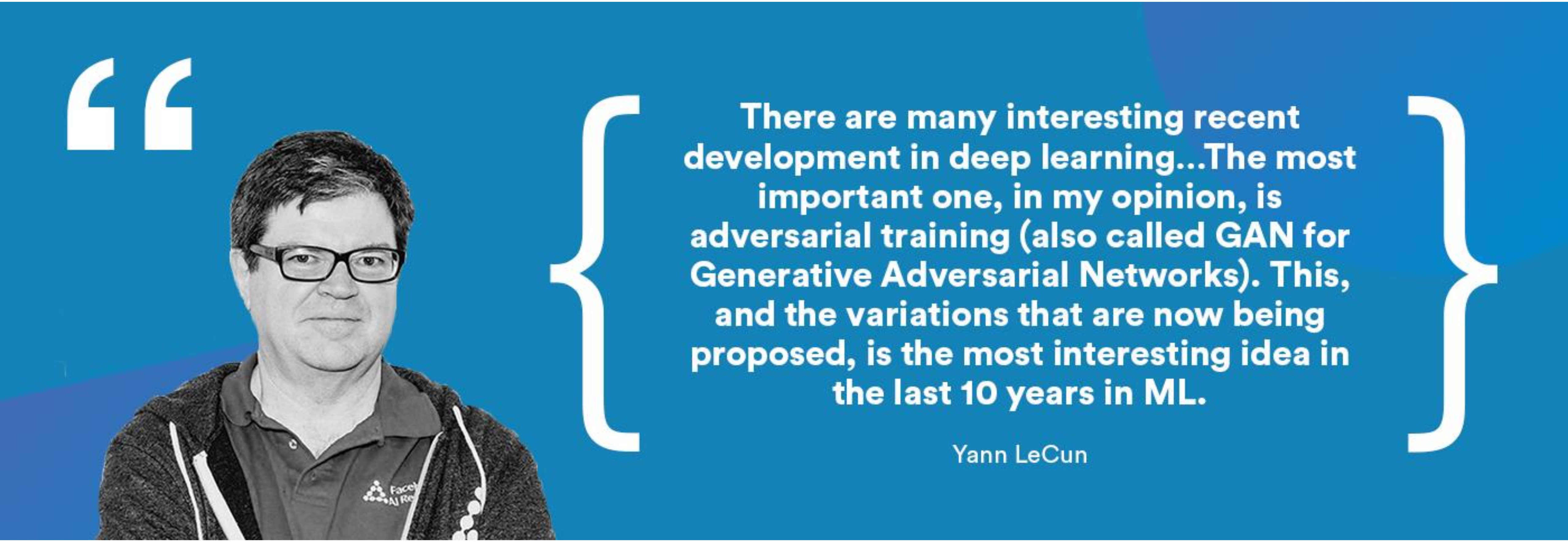
```
vae = Model(input_img, y)
vae.compile(optimizer='rmsprop', loss=None)
vae.summary()

# Train the VAE on MNIST digits
(x_train, _), (x_test, y_test) = mnist.load_data()

x_train = x_train.astype('float32') / 255.
x_train = x_train.reshape(x_train.shape + (1,))
x_test = x_test.astype('float32') / 255.
x_test = x_test.reshape(x_test.shape + (1,))

vae.fit(x=x_train, y=None, shuffle=True, epochs=10, batch_size=batch_size,
        validation_data=(x_test, None))
```

Use Decoder to Turn Latent Vectors into Images

```python
import matplotlib.pyplot as plt
from scipy.stats import norm

# Display a 2D manifold of the digits
n = 15  # figure with 15x15 digits
digit_size = 28
figure = np.zeros((digit_size * n, digit_size * n))
# Linearly spaced coordinates on the unit square transformed via the inverse CDF (ppf) of the Gaussian
# to produce values of the latent variables z, since the prior of the latent space is Gaussian
grid_x = norm.ppf(np.linspace(0.05, 0.95, n))
grid_y = norm.ppf(np.linspace(0.05, 0.95, n))

for i, yi in enumerate(grid_x):
    for j, xi in enumerate(grid_y):
        z_sample = np.array([[xi, yi]])
        z_sample = np.tile(z_sample, batch_size).reshape(batch_size, 2)
        x_decoded = decoder.predict(z_sample, batch_size=batch_size)
        digit = x_decoded[0].reshape(digit_size, digit_size)
        figure[i * digit_size: (i + 1) * digit_size, j * digit_size: (j + 1) * digit_size] = digit

plt.figure(figsize=(10, 10))
plt.imshow(figure, cmap='Greys_r')
plt.show()
```


0
50
100
150
200
250
300
350
400
0
50
100
150
200
250
300
350
400

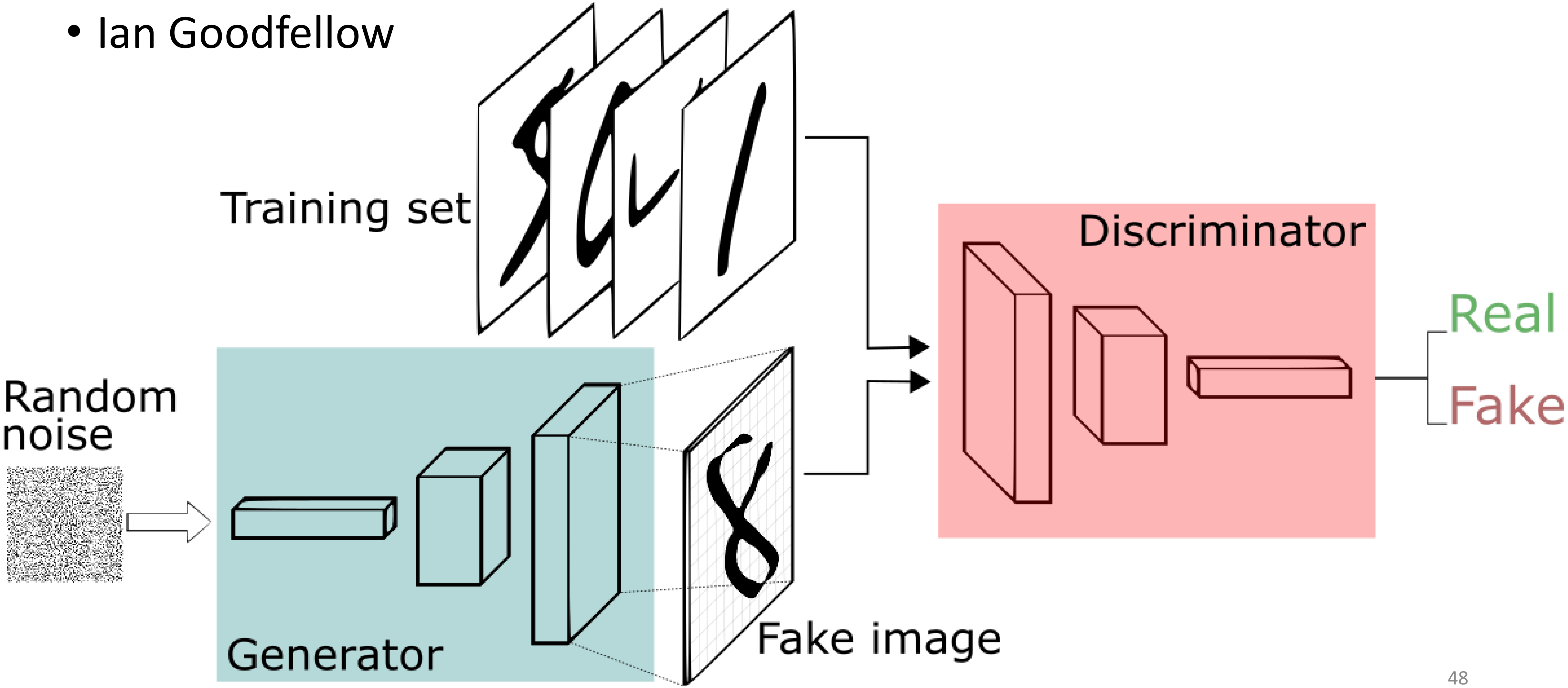
Generative Adversarial Networks (GAN)



- https://www.youtube.com/watch?v=9JpdAg6uMXs
- https://arxiv.org/abs/1701.00160

Generative Adversarial Networks (GAN)

- Ian Goodfellow



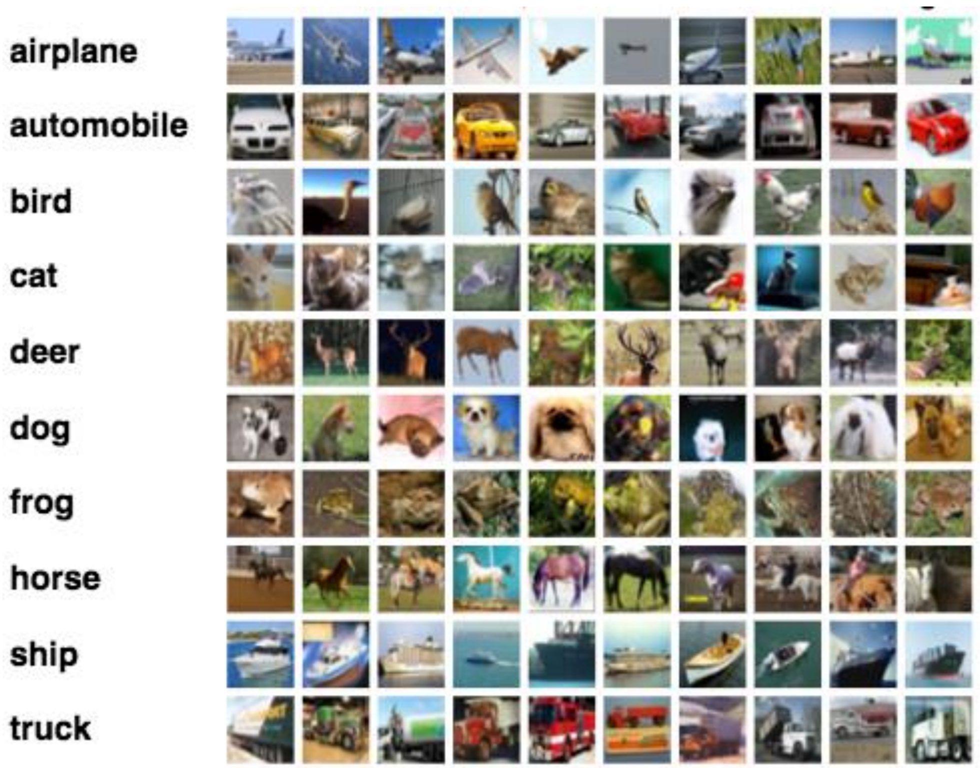
Bag of Tricks for Training GANs

- Use tanh as the last activation in the generator, instead of sigmoid
- Sample points from the latent space using a normal distribution
- Stochasticity is good to induce robustness. Introducing randomness during training helps prevent GAN to get stuck.
  - Use dropout in the discriminator
  - Add some random noise to the labels for the discriminator.
- Sparse gradients can hinder GAN training. There are two things that can induce gradient sparsity: 1) max pooling operations, 2) ReLU activations.
  - Use strided convolutions for downsampling
  - Use LeakyReLU, which allows small negative activation values.
- In generated images, it is common to see "checkerboard artifacts" caused by unequal coverage of the pixel space in the generator.
  - Use a kernel size that is divisible by the stride size

Train a GAN of Frog

- Use frog images from CIFAR10
  - 50,000 32x32 RGB images belong to 10 classes (5,000 images per class).

https://github.com/fchollet/deep-learning-with-python-notebooks/blob/master/8.5-introduction-to-gans.ipynb



Generator

```
latent_dim = 32; height = 32; width = 32; channels = 3

generator_input = keras.Input(shape=(latent_dim,))

# First, transform the input into a 16x16 128-channels feature map
x = layers.Dense(128 * 16 * 16)(generator_input)
x = layers.LeakyReLU()(x)
x = layers.Reshape((16, 16, 128))(x)

# Then, add a convolution layer
x = layers.Conv2D(256, 5, padding='same')(x)
x = layers.LeakyReLU()(x)

# Upsample to 32x32
x = layers.Conv2DTranspose(256, 4, strides=2, padding='same')(x)
x = layers.LeakyReLU()(x)

# Few more conv layers
x = layers.Conv2D(256, 5, padding='same')(x)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(256, 5, padding='same')(x)
x = layers.LeakyReLU()(x)

# Produce a 32x32 1-channel feature map
x = layers.Conv2D(channels, 7, activation='tanh', padding='same')(x)
generator = keras.models.Model(generator_input, x)
generator.summary()
```

Discriminator

```
discriminator_input = layers.Input(shape=(height, width, channels))
x = layers.Conv2D(128, 3)(discriminator_input)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(128, 4, strides=2)(x)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(128, 4, strides=2)(x)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(128, 4, strides=2)(x)
x = layers.LeakyReLU()(x)
x = layers.Flatten()(x)

# One dropout layer - important trick!
x = layers.Dropout(0.4)(x)

# Classification layer
x = layers.Dense(1, activation='sigmoid')(x)

discriminator = keras.models.Model(discriminator_input, x)
discriminator.summary()

# To stabilize training, we use learning rate decay
# and gradient clipping (by value) in the optimizer.
discriminator_optimizer = keras.optimizers.RMSprop(lr=0.0008, clipvalue=1.0, decay=1e-8)
discriminator.compile(optimizer=discriminator_optimizer, loss='binary_crossentropy')
```

Freeze Discriminator When Training Generator

- We’ll train discriminator and generator alternately

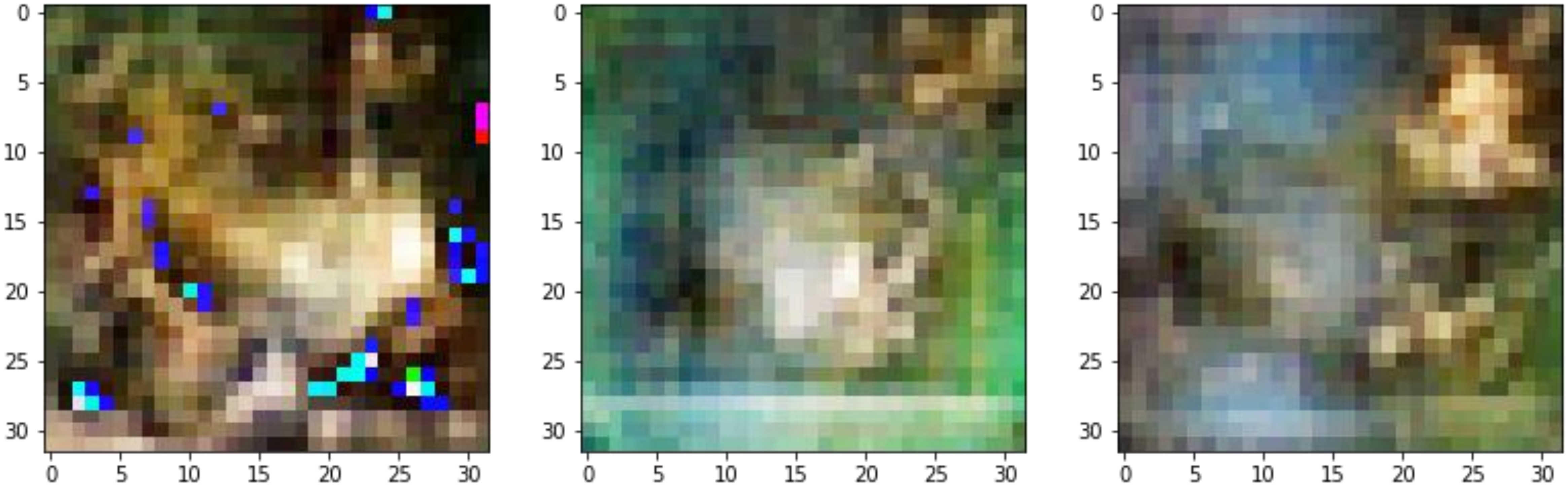
```
# Set discriminator weights to non-trainable
# (will only apply to the `gan` model)
discriminator.trainable = False

gan_input = keras.Input(shape=(latent_dim,))
gan_output = discriminator(generator(gan_input))
gan = keras.models.Model(gan_input, gan_output)

gan_optimizer = keras.optimizers.RMSprop(lr=0.0004, clipvalue=1.0, decay=1e-8)
gan.compile(optimizer=gan_optimizer, loss='binary_crossentropy')
```

Training DCGAN

- for each epoch:
  - Draw random points in the latent space (random noise).
  - Generate images with `generator` using this random noise.
  - Mix the generated images with real ones.
  - Train `discriminator` using these mixed images, with corresponding targets, either "real" (for the real images) or "fake" (for the generated images).
  - Draw new random points in the latent space.
  - Trains the generator to fool the discriminator => train `gan` using these random vectors, with targets that all say "these are real images".

```python
for step in range(iterations):
    # Sample random points in the latent space
    random_latent_vectors = np.random.normal(size=(batch_size, latent_dim))
    # Decode them to fake images
    generated_images = generator.predict(random_latent_vectors)
    # Combine them with real images
    stop = start + batch_size
    real_images = x_train[start: stop]
    combined_images = np.concatenate([generated_images, real_images])
    # Assemble labels discriminating real from fake images
    labels = np.concatenate([np.ones((batch_size, 1)), np.zeros((batch_size, 1))])
    # Add random noise to the labels - important trick!
    labels += 0.05 * np.random.random(labels.shape)
    # Train the discriminator
    d_loss = discriminator.train_on_batch(combined_images, labels)
    # sample random points in the latent space
    random_latent_vectors = np.random.normal(size=(batch_size, latent_dim))
    # Assemble labels that say "all real images"
    misleading_targets = np.zeros((batch_size, 1))
    # Train the generator (via the gan model,
    # where the discriminator weights are frozen)
    a_loss = gan.train_on_batch(random_latent_vectors, misleading_targets)
```

Generated Frog Images



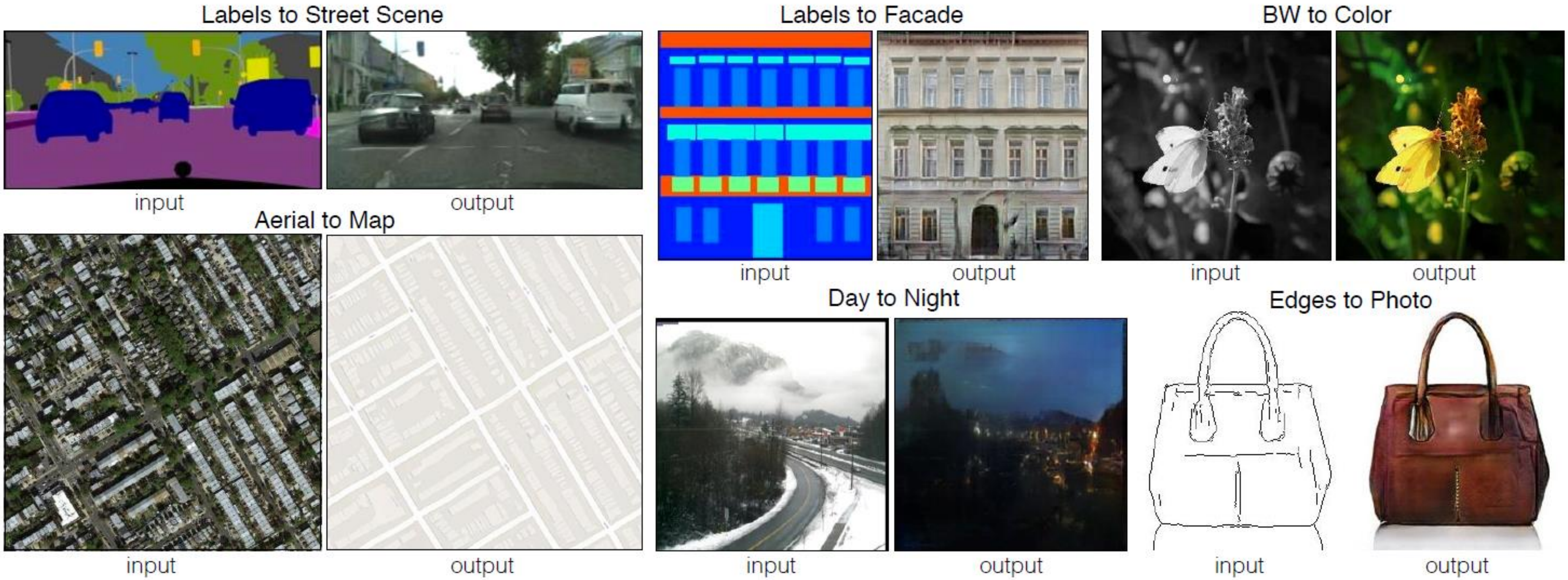
Other Advanced GAN Models

- TensorFlow Tutorial / Generative

1. Pixel-2-Pixel
2. CycleGAN
3. Adversarial FGSM

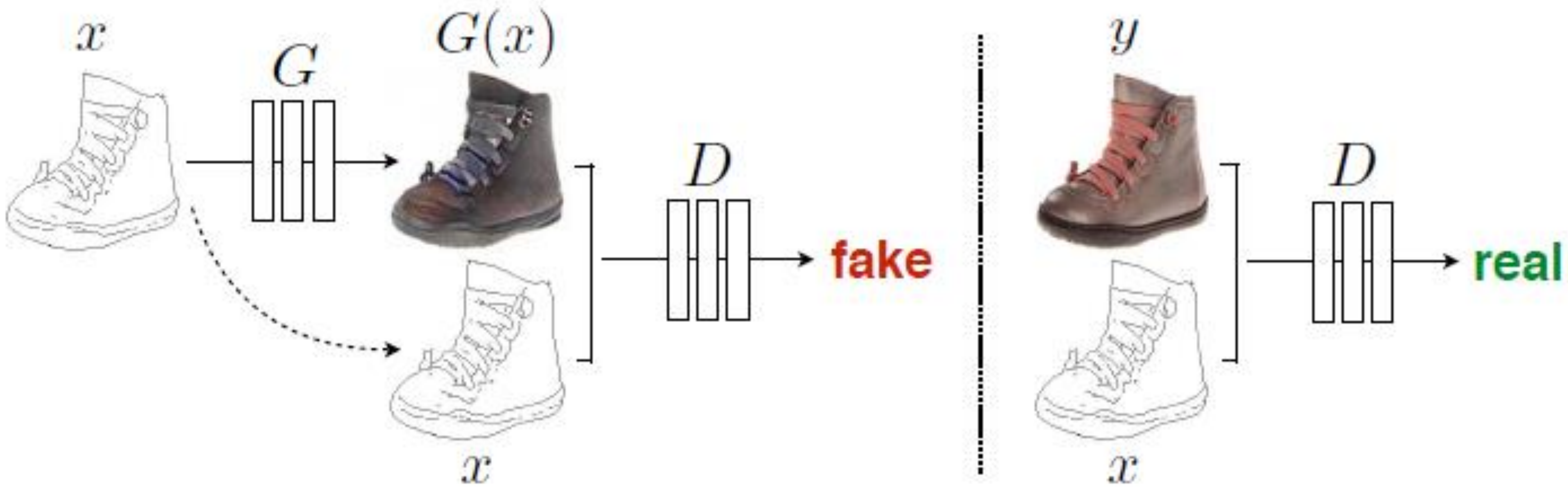
Pix2Pix

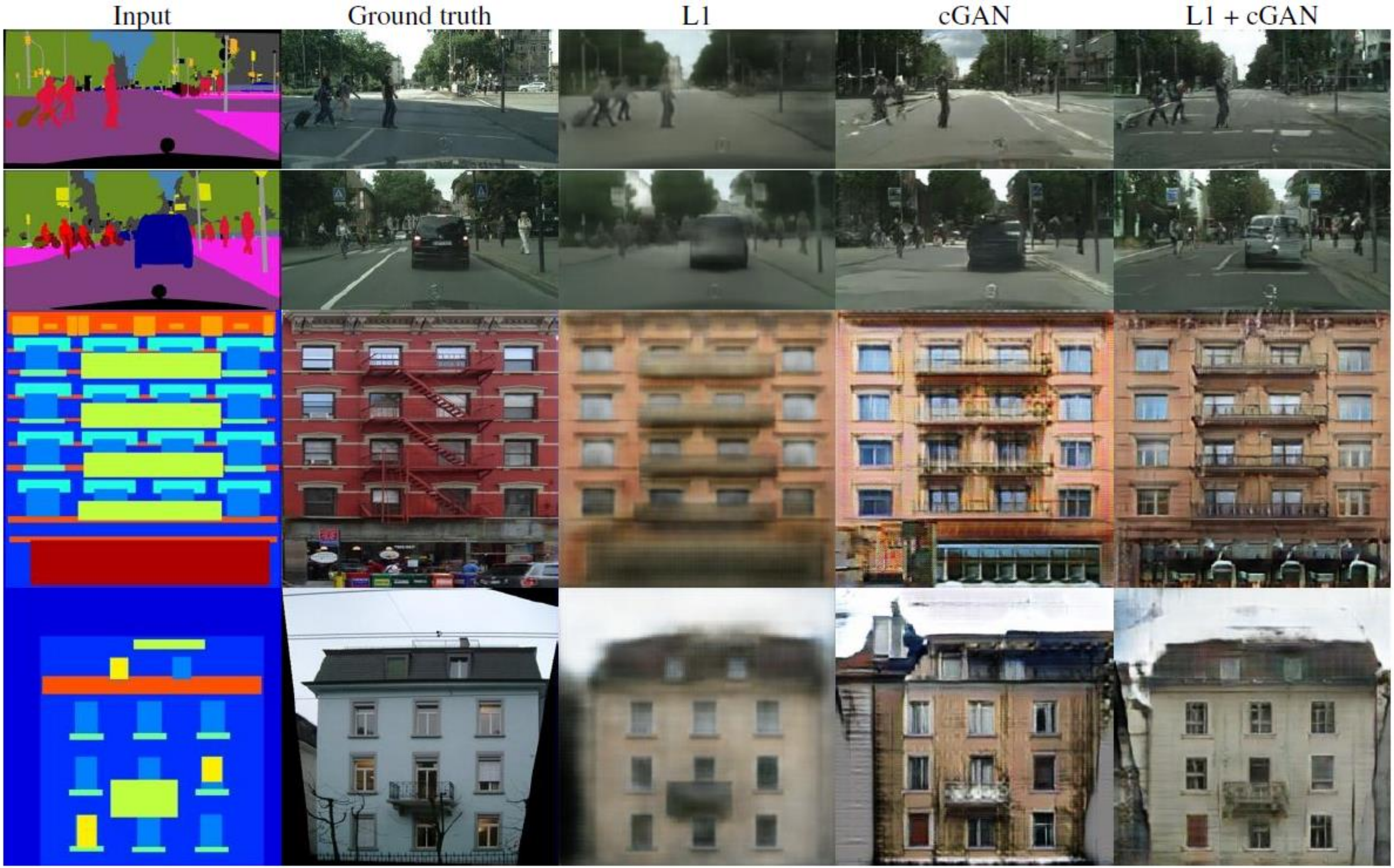
- Phillip Isola et al., [Image-to-Image Translation with Conditional Adversarial Networks](), 2018



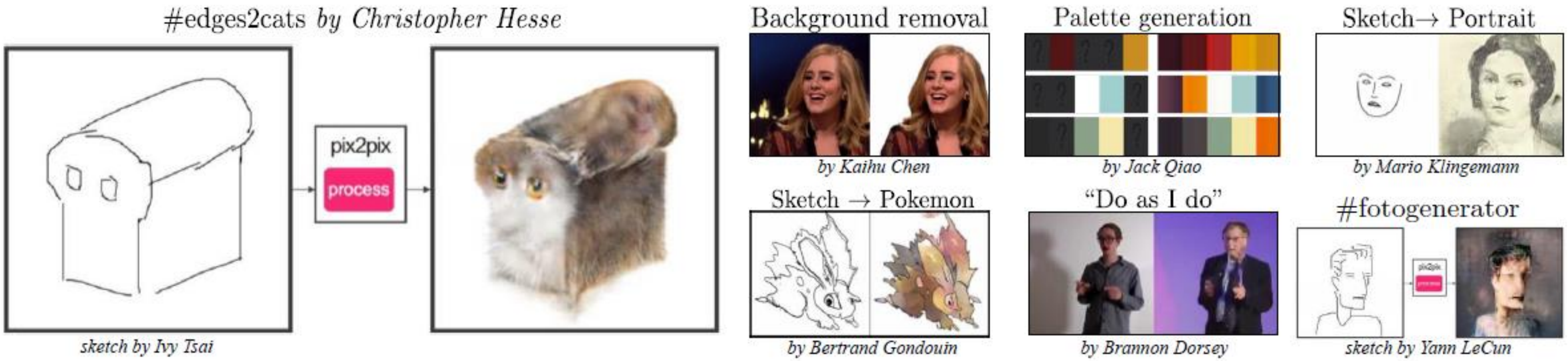
Training Conditional GAN

- Both the generator and discriminator observe the input edge map
- Use U-Net and PatchGAN discriminator

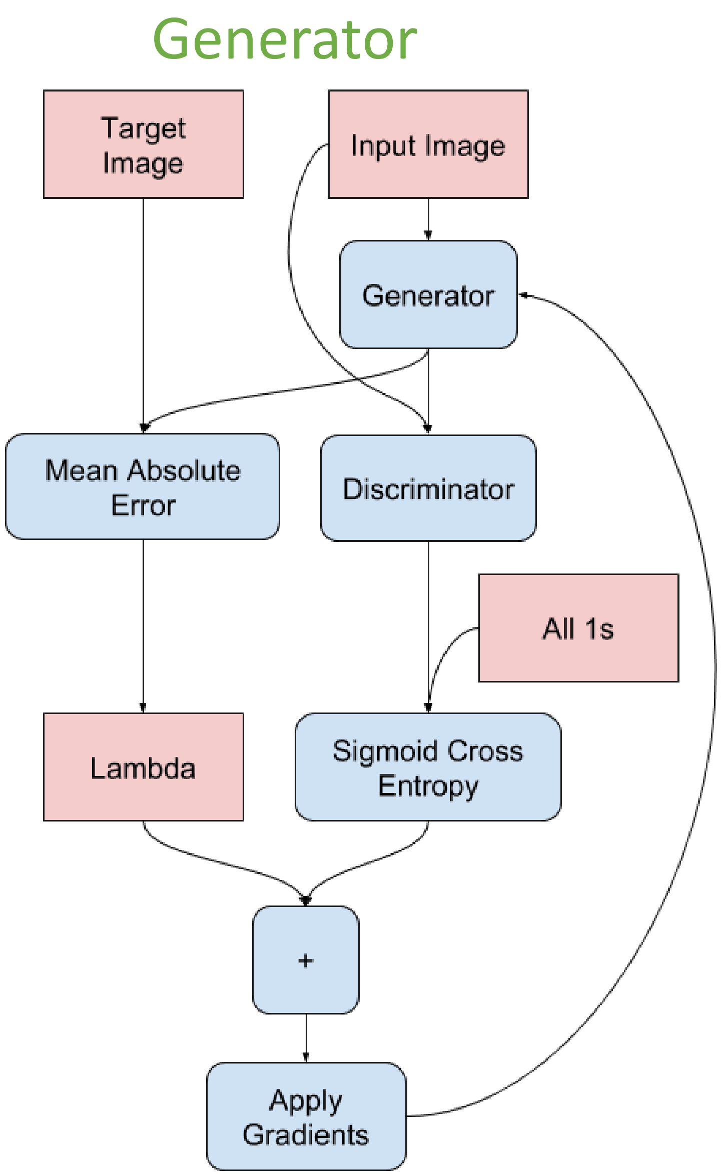


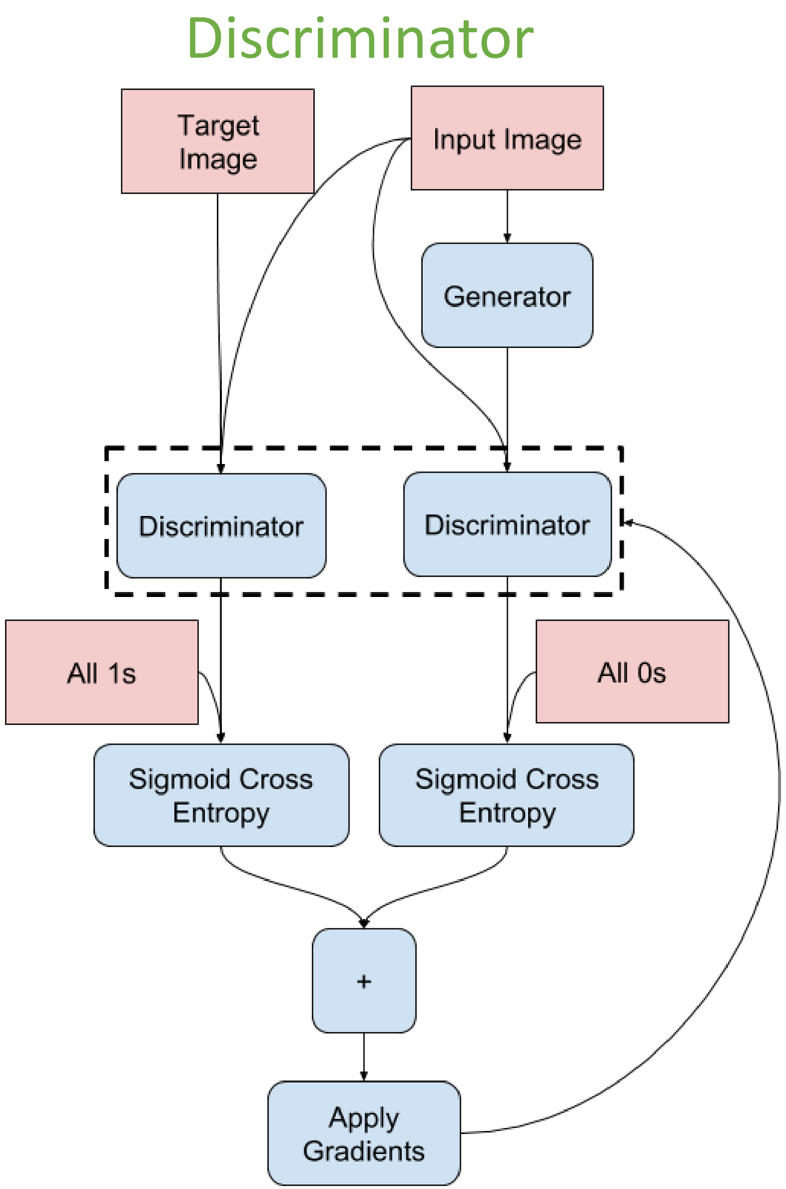

Input
Ground truth
L1
cGAN
L1 + cGAN

Applications based on Pix-2-Pix



Design of Generator and Discriminator

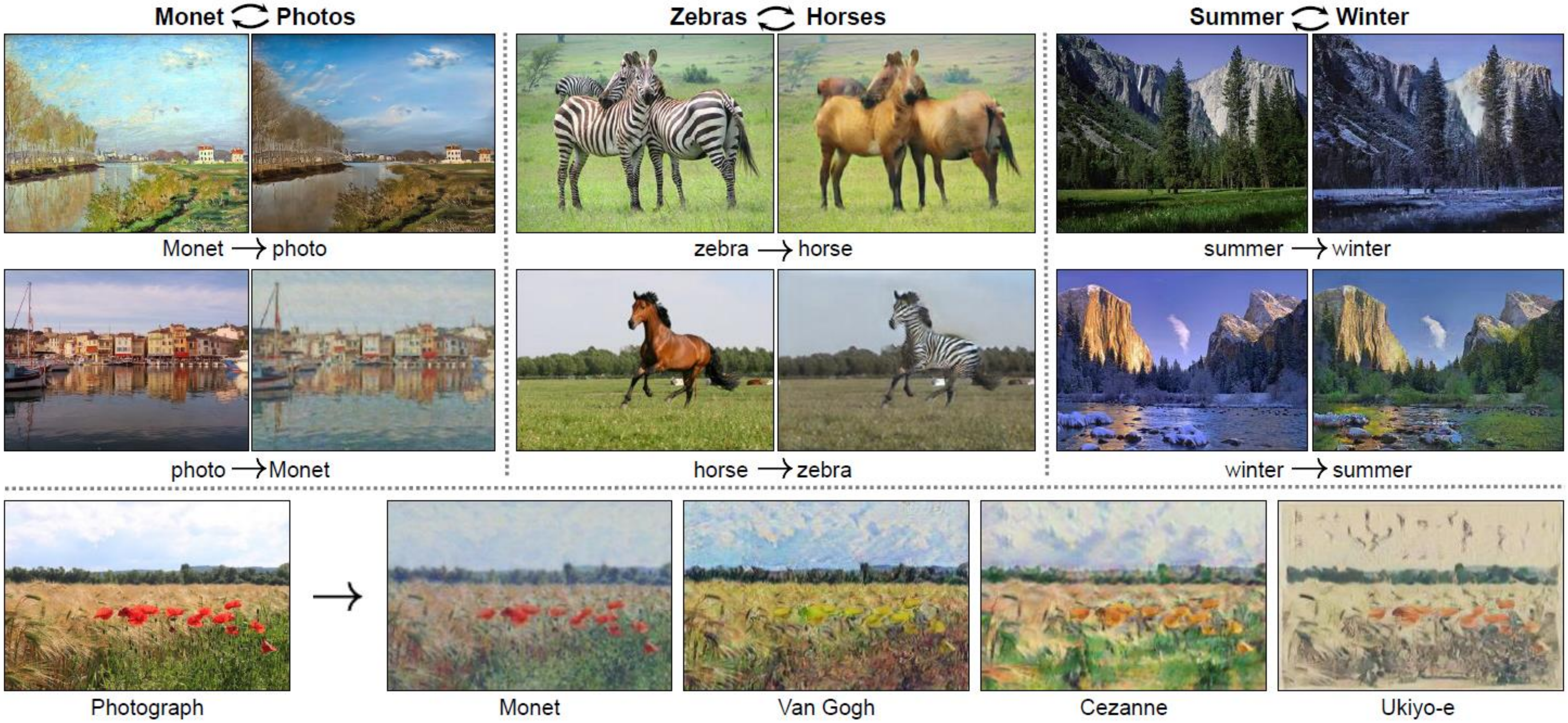




https://www.tensorflow.org/tutorials/generative/pix2pix

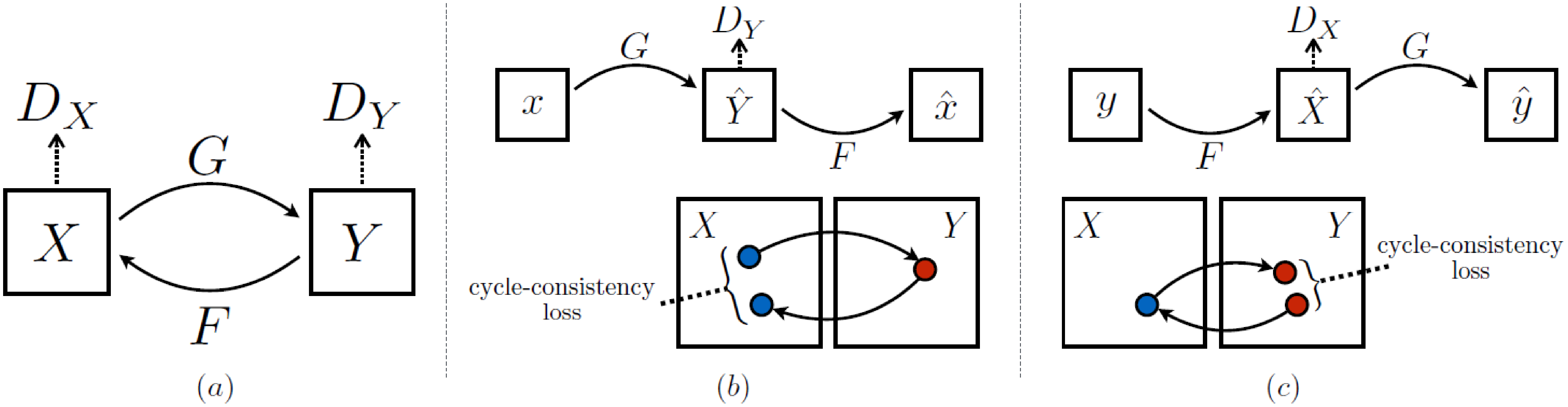
CycleGAN

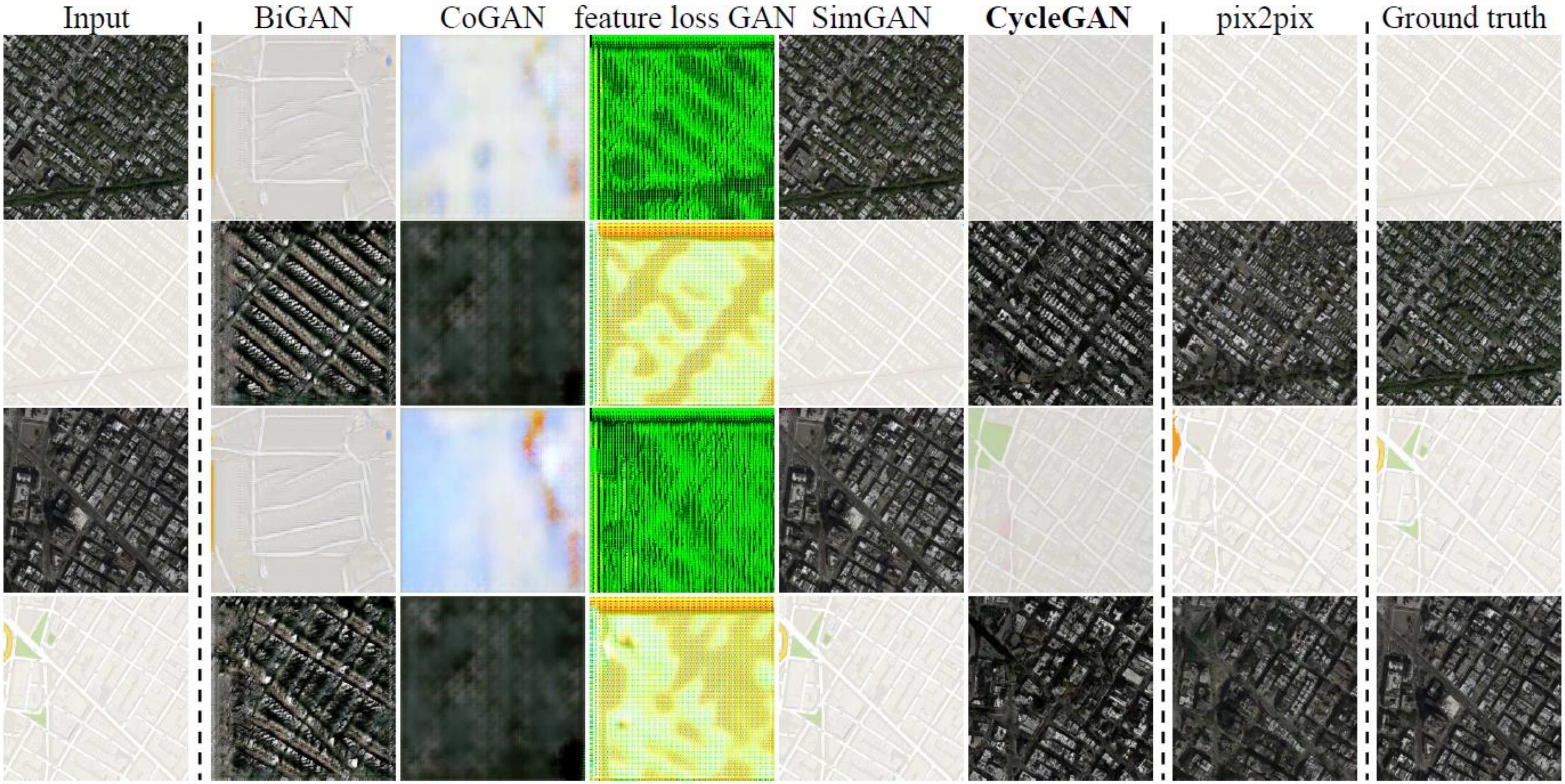
- Zhu et al., [Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks](), 2018
- Learn to automatically "translate" an image from one into the other and vice versa



Model of CycloneGAN

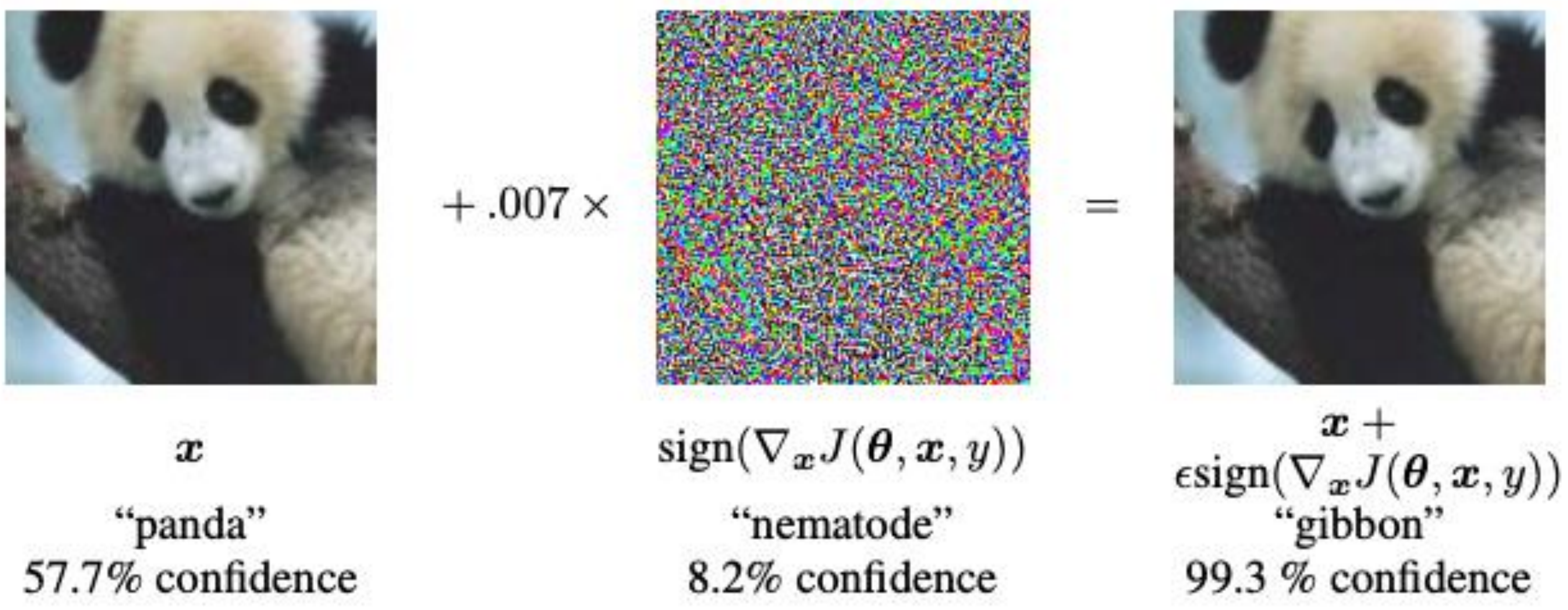
- Two mapping functions G : X → Y and F : Y → X
- Two cycle consistency losses:
  - Forward cycle-consistency loss: x → G(x) → F(G(x)) ≈ x
  - Backward cycle-consistency loss: y → F(y) → G(F(y)) ≈ y




Input
BiGAN
CoGAN
feature loss GAN
SimGAN
CycleGAN
pix2pix
Ground truth

Adversarial Attack

- Goodfellow *et al.*, Explaining and Harnessing Adversarial Examples, 2015
- Fast Gradient Signed Method (FGSM)



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

References

- Francois Chollet, “Deep Learning with Python,” Chapter 8
- https://www.tensorflow.org/tutorials/generative/