Generative Learning

June 1, 2021

ddebarr@uw.edu

http://cross-entropy.net/ML530/Deep_Learning_6.pdf

[DLP] Generative Deep Learning

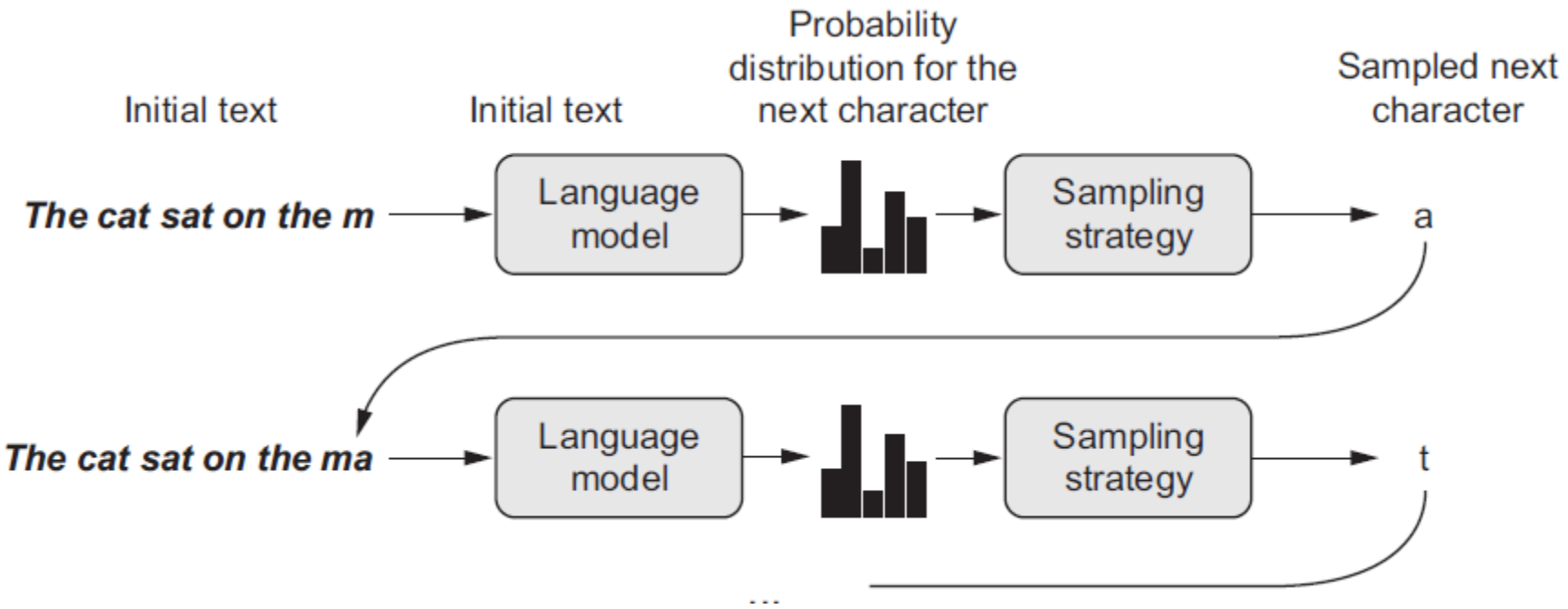
1. Text Generation with LSTM
2. Implementing Deep Dream
3. Performing Neural Style Transfer
4. Variational AutoEncoders
5. Understanding Generative Adversarial Networks

Augmented Intelligence (the other AI)

- "… algorithmic generation … can make artistic creation more accessible by eliminating the need for technical skill and practice - setting up a new medium of pure expression, factoring art apart from craft"
- "Iannis Xenakis, a visionary pioneer of electronic and algorithmic music":

  "Freed from tedious calculations, the composer is able to devote himself to the general problems that the new musical form poses and to explore the nooks and crannies of this form while modifying the values of the input data. For example, he may test all instrumental combinations from soloists to chamber orchestras, to large orchestras. With the aid of electronic computers the composer becomes a sort of pilot: he presses the buttons, introduces coordinates, and supervises the controls of a cosmic vessel sailing in the space of sound, across sonic constellations and galaxies that he could formerly glimpse only as a distant dream."

Character-Level Neural Language Model



Sampling Strategy

- Greedy Sampling: always choose the most likely candidate character
- Stochastic Sampling: sampling from the probability distribution for the next character

Reweighting a Distribution to a Different Temperature

```
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)
```

original_distribution is a 1D Numpy array of probability values that must sum to 1. temperature is a factor quantifying the entropy of the output distribution.

Returns a reweighted version of the original distribution. The sum of the distribution may no longer be 1, so you divide it by its sum to obtain the new distribution.

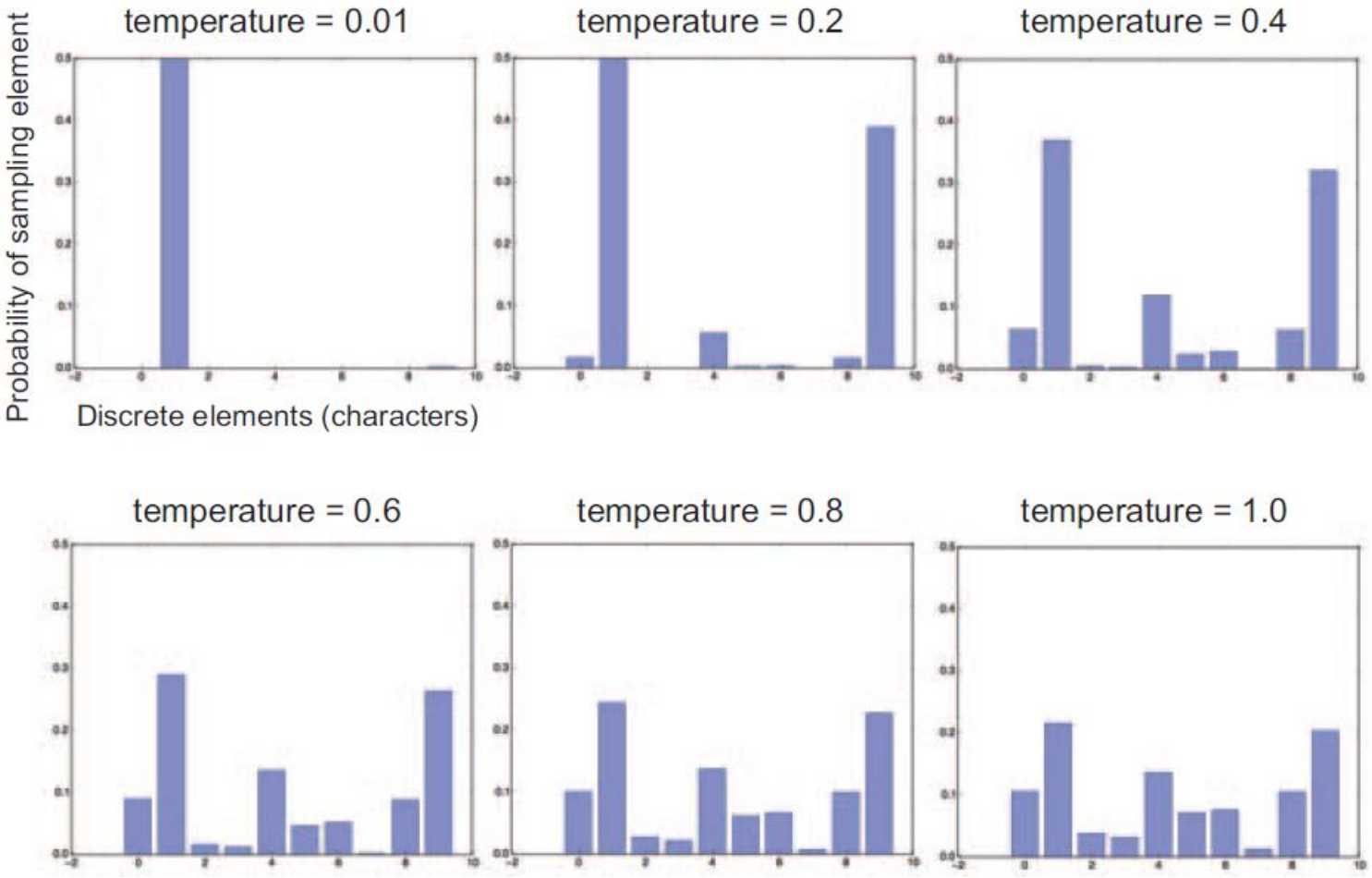
- Higher temperature: increases the probability of selecting a lower probability token

```
>>> temp = 10;    exp(log([.1, .2, .7]) / temp) / sum(exp(log([.1, .2, .7]) / temp))
array([0.30426696, 0.32610526, 0.36962778])
```

- Lower temperature: decreases the probability of selecting a lower probability token

```
>>> temp = .1;    exp(log([.1, .2, .7]) / temp) / sum(exp(log([.1, .2, .7]) / temp))
array([3.54012033e-09, 3.62508322e-06, 9.99996371e-01])
```

Higher Temp: More Random
Lower Temp: More Deterministic



Temperature = 1 returns the original probabilities

LSTM Text Generation: Downloading Text

```
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))
```

Friedrich Nietzsche was a German philosopher

LSTM Text Generation: Vectorizing the Text

```
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

[includes '\n']

LSTM Text Generation: Define the Model

```
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 the Language Model and Sampling from It

1. Draw from the model a probability distribution for the next character, given the generated text available so far.
2. Reweight the distribution to a certain temperature.
3. Sample the next character at random according to the reweighted distribution.
4. Add the new character at the end of the available text.

Stochastic Sampling for the Next Character

```
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)
```

LSTM Text Generation: Train the Model

```
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

LSTM Text Generation: Generate the Text [part of the "temperature" loop (previous slide)]

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

Epoch 20: Temperature = 0.2

Prompt: “new faculty, and the jubilation reached its climax when kant”

```
new faculty, and the jubilation reached its climax when kant and such a man
in the same time the spirit of the surely and the such the such
as a man is the sunligh and subject the present to the superiority of the
special pain the most man and strange the subjection of the
special conscience the special and nature and such men the subjection of the
special men, the most surely the subjection of the special
intellect of the subjection of the same things and
```

Epoch 20: Temperature = 0.5

Prompt: “new faculty, and the jubilation reached its climax when kant”

```
new faculty, and the jubilation reached its climax when kant in the eterned
and such man as it's also become himself the condition of the
experience of off the basis the superiory and the special morty of the
strength, in the langus, as which the same time life and "even who
discless the mankind, with a subject and fact all you have to be the stand
and lave no comes a troveration of the man and surely the
conscience the superiority, and when one must be w
```

Epoch 20: Temperature = 1.0

Prompt: “new faculty, and the jubilation reached its climax when kant”

```
new faculty, and the jubilation reached its climax when kant, as a
periliting of manner to all definites and transpects it it so
hicable and ont him artiar resull
too such as if ever the proping to makes as cnecience. to been juden,
all every could coldiciousnike hother aw passife, the plies like
which might thiod was account, indifferent germin, that everythery
certain destrution, intellect into the deteriorablen origin of moralian,
and a lessority o
```

Epoch 60: Temperature = 0.2

Prompt: “cheerfulness, friendliness and kindness of a heart are”

```
cheerfulness, friendliness and kindness of a heart are the sense of the
spirit is a man with the sense of the sense of the world of the
self-end and self-concerning the subjection of the strengthorixes--the
subjection of the subjection of the subjection of the
self-concerning the feelings in the superiority in the subjection of the
subjection of the spirit isn't to be a man of the sense of the
subjection and said to the strength of the sense of the
```

Epoch 60: Temperature = 0.5

Prompt: “cheerfulness, friendliness and kindness of a heart are”

```
cheerfulness, friendliness and kindness of a heart are the part of the soul
who have been the art of the philosophers, and which the one
won't say, which is it the higher the and with religion of the frences.
the life of the spirit among the most continuess of the
strengther of the sense the conscience of men of precisely before enough
presumption, and can mankind, and something the conceptions, the
subjection of the sense and suffering and the
```

Epoch 60: Temperature = 1.0

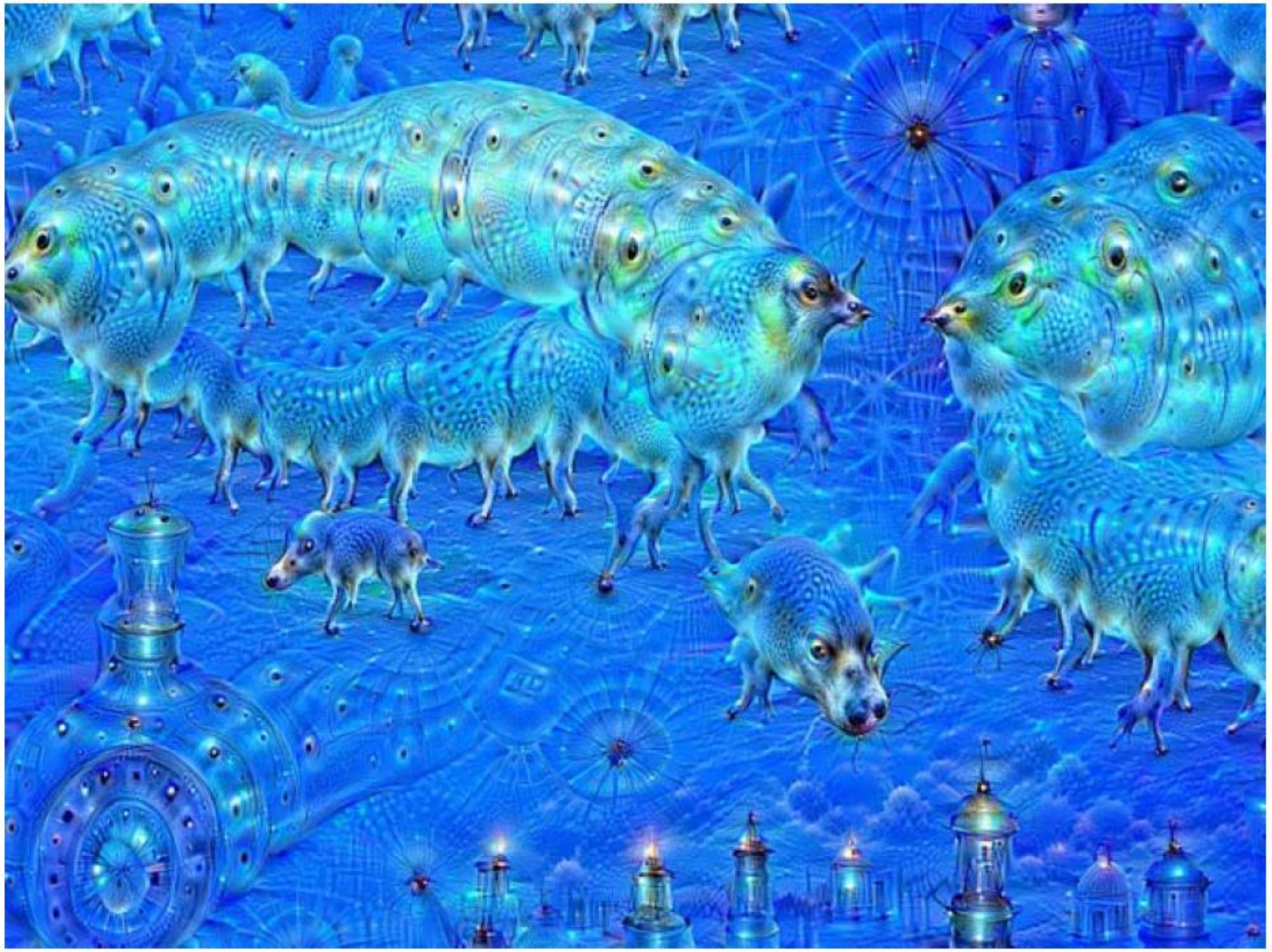
Prompt: “cheerfulness, friendliness and kindness of a heart are”

```
cheerfulness, friendliness and kindness of a heart are spiritual by the
ciuture for the
entalled is, he astraged, or errors to our you idstood--and it needs,
to think by spars to whole the amvives of the newoatly, prefectly
raals! it was
name, for example but voludd atu-especity"--or rank onee, or even all
"solett increessic of the world and
implussional tragedy experience, transf, or insiderar,--must hast
if desires of the strubction is be stronges
```

Wrapping Up

- You can generate discrete sequence data by training a model to predict the next token(s), given previous tokens.
- In the case of text, such a model is called a *language model*. It can be based on either words or characters.
- Sampling the next token requires balance between adhering to what the model judges likely, and introducing randomness.
- One way to handle this is the notion of softmax temperature. Always experiment with different temperatures to find the right one.

DeepDream Output Example



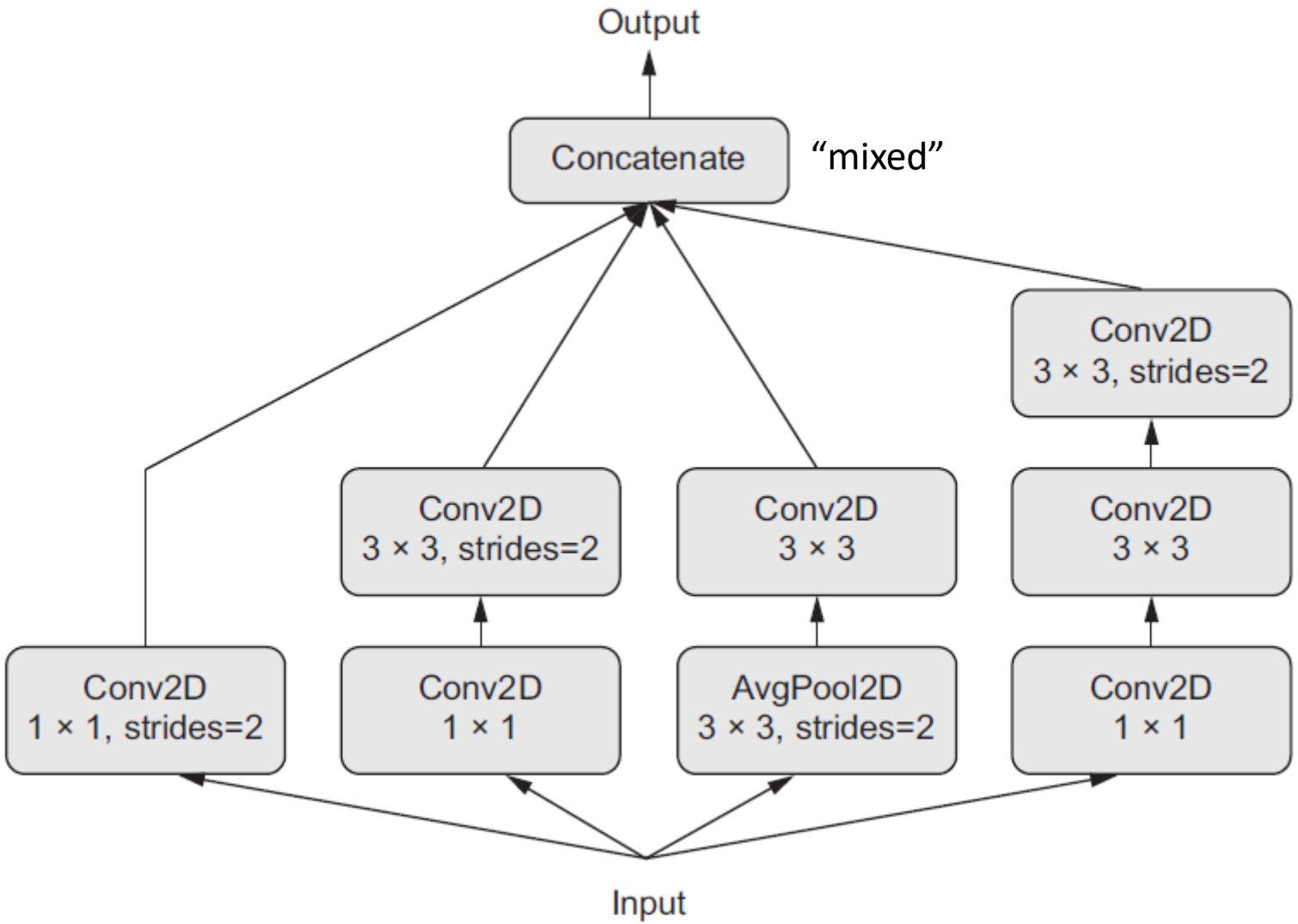
Loading the Pretrained Inception v3 Model

```
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.

An Example of an Inception Module



Setting Up the DeepDream Configuration

```
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~~ Utility to be Maximized

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.

```
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
```

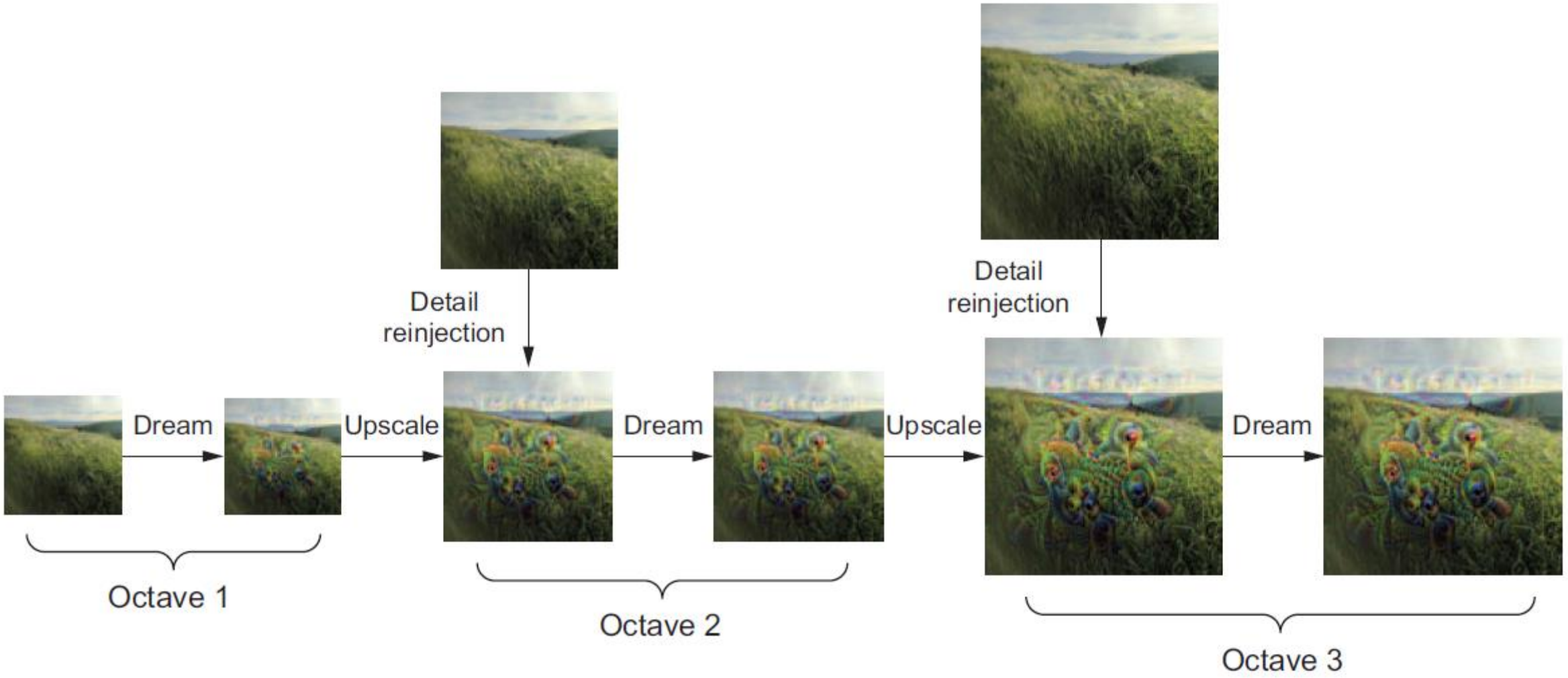
Computes the gradients of the dream with regard to the loss

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.

DeepDream Processing



Gradient Ascent Configuration

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)

Gradient Ascent Over Multiple Scales

```
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

Reinjects lost detail into the dream

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

Auxillary Functions: 1 of 2

```
import scipy
from keras.preprocessing import image

def resize_img(img, size):
    img = np.copy(img)
    factors = (1,
               float(size[0]) / img.shape[1],
               float(size[1]) / img.shape[2],
               1)
    return scipy.ndimage.zoom(img, factors, order=1)

def save_img(img, fname):
    pil_img = deprocess_image(np.copy(img))
    scipy.misc.imsave(fname, pil_img)

def preprocess_image(image_path):
    img = image.load_img(image_path)
    img = image.img_to_array(img)
    img = np.expand_dims(img, axis=0)
    img = inception_v3.preprocess_input(img)
    return img
```

Util function to open, resize, and format pictures into tensors that Inception V3 can process

Auxillary Functions: 2 of 2

```
def deprocess_image(x):
    if K.image_data_format() == 'channels_first':
        x = x.reshape((3, x.shape[2], x.shape[3]))
        x = x.transpose((1, 2, 0))
    else:
        x = x.reshape((x.shape[1], x.shape[2], 3))
    x /= 2.
    x += 0.5
    x *= 255.
    x = np.clip(x, 0, 255).astype('uint8')
    return x
```

Util function to convert a tensor into a valid image

Undoes preprocessing that was performed by inception_v3.preprocess_input

Running DeepDream on an Example Image

Emphasizing Different Layers

Wrapping Up

- DeepDream consists of running a convnet in reverse to generate inputs based on the representations learned by the network.
- The results produced are fun and somewhat similar to the visual artifacts induced in humans by the disruption of the visual cortex via psychedelics.
- Note that the process isn't specific to image models or even to convnets. It can be done for speech, music, and more.

A Style Transfer Example

- Style refers to the textures, colors, and visual patterns of an image
- Content refers the higher-level macrostructure of an image

Neural Style Transfer Objectives

- Preserve content by maintaining similar high-level layer activations between the target content image and the generated image. The convnet should "see" both the target image and the generated image as containing the same things.
- Preserve style by maintaining similar *correlations* within activations for both low-level layers and high-level layers. Feature correlations capture *textures*: the generated image and the style-reference image should share the same textures at different spatial scales.

```
loss = distance(style(reference_image) - style(generated_image)) +
       distance(content(original_image) - content(generated_image))
```

Neural Style Transfer Processing

1. Set up a network that computes VGG19 layer activations for the style-reference image, the target image, and the generated image at the same time.
2. Use the layer activations computed over these three images to define the loss function described earlier, which you'll minimize in order to achieve style transfer.
3. Set up a gradient-descent process to minimize this loss function.

Variable Initialization

```
from keras.preprocessing.image import load_img, img_to_array

target_image_path = 'img/portrait.jpg'
style_reference_image_path = 'img/transfer_style_reference.jpg'

width, height = load_img(target_image_path).size
img_height = 400
img_width = int(width * img_height / height)
```

Path to the image you want to transform

Path to the style image

Dimensions of the generated picture

Auxillary Functions

```
import numpy as np
from keras.applications import vgg19

def preprocess_image(image_path):
    img = load_img(image_path, target_size=(img_height, img_width))
    img = img_to_array(img)
    img = np.expand_dims(img, axis=0)
    img = vgg19.preprocess_input(img)
    return img


def deprocess_image(x):
    x[:, :, 0] += 103.939
    x[:, :, 1] += 116.779
    x[:, :, 2] += 123.68
    x = x[:, :, ::-1]
    x = np.clip(x, 0, 255).astype('uint8')
    return x
```

Zero-centering by removing the mean pixel value from ImageNet. This reverses a transformation done by vgg19.preprocess_input.

Converts images from 'BGR' to 'RGB'. This is also part of the reversal of vgg19.preprocess_input.

Loading the Images and the Model

```
from keras import backend as K

target_image = K.constant(preprocess_image(target_image_path))
style_reference_image = K.constant(preprocess_image(style_reference_image_path))
combination_image = K.placeholder((1, img_height, img_width, 3))

input_tensor = K.concatenate([target_image,
                              style_reference_image,
                              combination_image], axis=0)

model = vgg19.VGG19(input_tensor=input_tensor,
                    weights='imagenet',
                    include_top=False)
print('Model loaded.')
```

Placeholder that will contain the generated image

Combines the three images in a single batch

Builds the VGG19 network with the batch of three images as input. The model will be loaded with pretrained ImageNet weights.

Neural Style Transfer Loss

Neural Style Transfer Loss is the Weighted Sum of 3 Types of Loss:

- Content Loss
- Style Loss
- Total Variation Loss

Content Loss Function

```
def content_loss(base, combination):
    return K.sum(K.square(combination - base))
```

Style Loss Function

```
def gram_matrix(x):
    features = K.batch_flatten(K.permute_dimensions(x, (2, 0, 1)))
    gram = K.dot(features, K.transpose(features))
    return gram

def style_loss(style, combination):
    S = gram_matrix(style)
    C = gram_matrix(combination)
    channels = 3
    size = img_height * img_width
    return K.sum(K.square(S - C)) / (4. * (channels ** 2) * (size ** 2))
```

Total Variation Loss Function

```
def total_variation_loss(x):
    a = K.square(
        x[:, :img_height - 1, :img_width - 1, :] -
        x[:, 1:, :img_width - 1, :])
    b = K.square(
        x[:, :img_height - 1, :img_width - 1, :] -
        x[:, :img_height - 1, 1:, :])
    return K.sum(K.pow(a + b, 1.25))
```

Fetching Layers and Defining Weights

Dictionary that maps layer names to activation tensors

```
outputs_dict = dict([(layer.name, layer.output) for layer in model.layers])
content_layer = 'block5_conv2'
style_layers = ['block1_conv1',
                'block2_conv1',
                'block3_conv1',
                'block4_conv1',
                'block5_conv1']
total_variation_weight = 1e-4
style_weight = 1.
content_weight = 0.025
```

Layer used for content loss

Layers used for style loss

Weights in the weighted average of the loss components

Add Loss Values

```
loss = K.variable(0.)
layer_features = outputs_dict[content_layer]
target_image_features = layer_features[0, :, :, :]
combination_features = layer_features[2, :, :, :]
loss += content_weight * content_loss(target_image_features,
                                      combination_features)

for layer_name in style_layers:
    layer_features = outputs_dict[layer_name]
    style_reference_features = layer_features[1, :, :, :]
    combination_features = layer_features[2, :, :, :]
    sl = style_loss(style_reference_features, combination_features)
    loss += (style_weight / len(style_layers)) * sl

loss += total_variation_weight * total_variation_loss(combination_image)
```

Adds the content loss

You'll define the loss by adding all components to this scalar variable.

Adds a style loss component for each target layer

Adds the total variation loss

Setting Up the Evaluator: 1 of 2

Gets the gradients of the generated image with regard to the loss

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

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

class Evaluator(object):

    def __init__(self):
        self.loss_value = None
        self.grads_values = None

    def loss(self, x):
        assert self.loss_value is None
        x = x.reshape((1, img_height, img_width, 3))
        outs = fetch_loss_and_grads([x])

        loss_value = outs[0]
        grad_values = outs[1].flatten().astype('float64')
        self.loss_value = loss_value
        self.grad_values = grad_values
        return self.loss_value
```

Function to fetch the values of the current loss and the current gradients

This class wraps fetch_loss_and_grads in a way that lets you retrieve the losses and gradients via two separate method calls, which is required by the SciPy optimizer you'll use.

Setting Up the Evaluator: 2 of 2

```
    def grads(self, x):
        assert self.loss_value is not None
        grad_values = np.copy(self.grad_values)
        self.loss_value = None
        self.grad_values = None
        return grad_values

evaluator = Evaluator()
```

Minimizing Neural Style Transfer Loss

```
from scipy.optimize import fmin_l_bfgs_b
from scipy.misc import imsave
import time

result_prefix = 'my_result'
iterations = 20

x = preprocess_image(target_image_path)
x = x.flatten()
for i in range(iterations):
    print('Start of iteration', i)
    start_time = time.time()
    x, min_val, info = fmin_l_bfgs_b(evaluator.loss,
                                     x,
                                     fprime=evaluator.grads,
                                     maxfun=20)
    print('Current loss value:', min_val)
    img = x.copy().reshape((img_height, img_width, 3))
    img = deprocess_image(img)
    fname = result_prefix + '_at_iteration_%d.png' % i
    imsave(fname, img)
    print('Image saved as', fname)
    end_time = time.time()
    print('Iteration %d completed in %ds' % (i, end_time - start_time))
```

This is the initial state: the target image.

You flatten the image because scipy.optimize.fmin_l_bfgs_b can only process flat vectors.

Runs L-BFGS optimization over the pixels of the generated image to minimize the neural style loss. Note that you have to pass the function that computes the loss and the function that computes the gradients as two separate arguments.

Saves the current generated image.

Neural Style Transfer for “The Starry Night” [Vincent van Gogh]

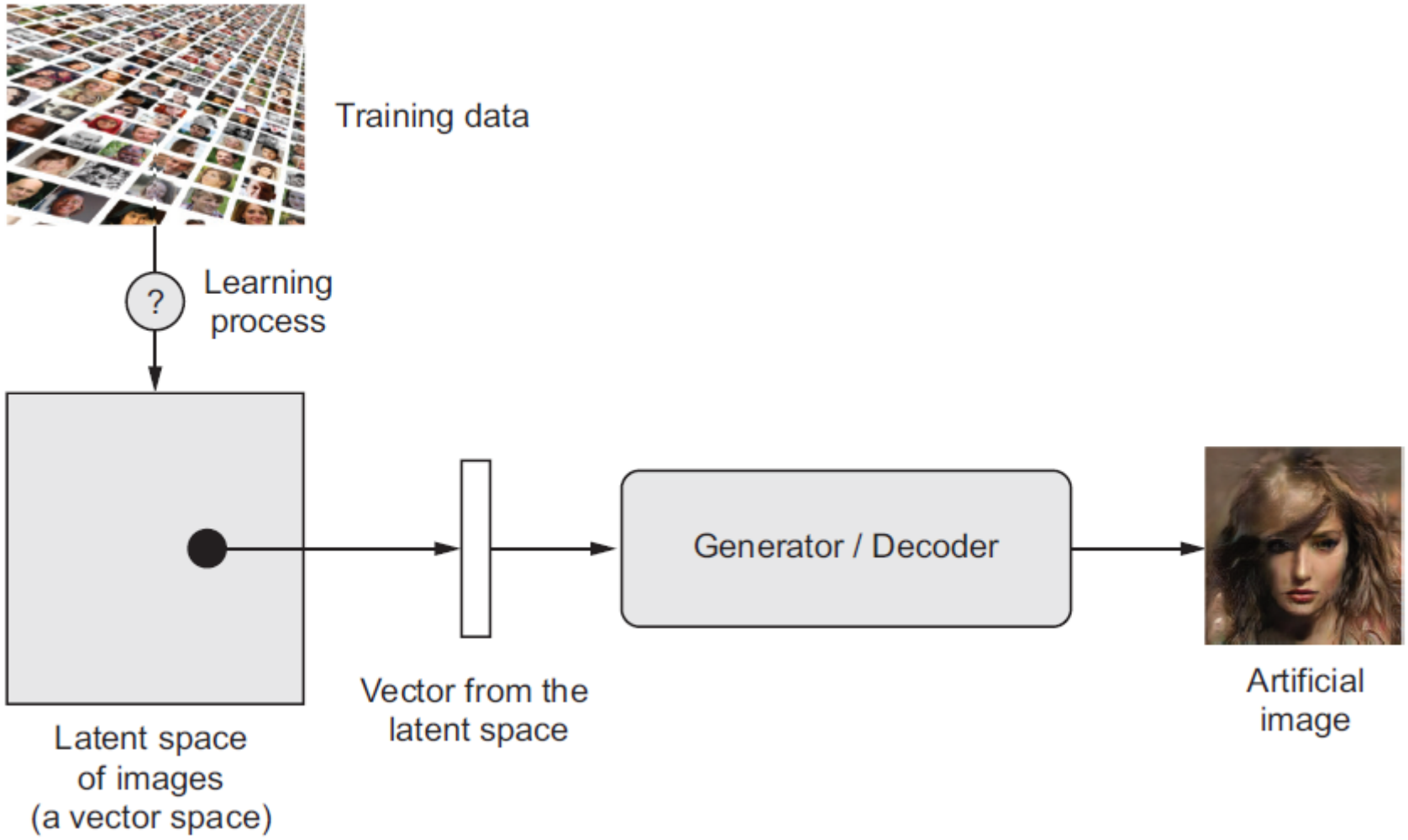
Neural Style Transfer for “The Scream” [Edvard Munch]

Neural Style Transfer for “Air+Man+Space” [Lyubov Popova]

Wrapping Up

- Style transfer consists of creating a new image that preserves the contents of a target image while also capturing the style of a reference image.
- Content can be captured by the high-level activations of a convnet.
- Style can be captured by the internal correlations of the activations of different layers of a convnet.
- Hence, deep learning allows style transfer to be formulated as an optimization process using a loss defined with a pretrained convnet.
- Starting from this basic idea, many variants and refinements are possible.

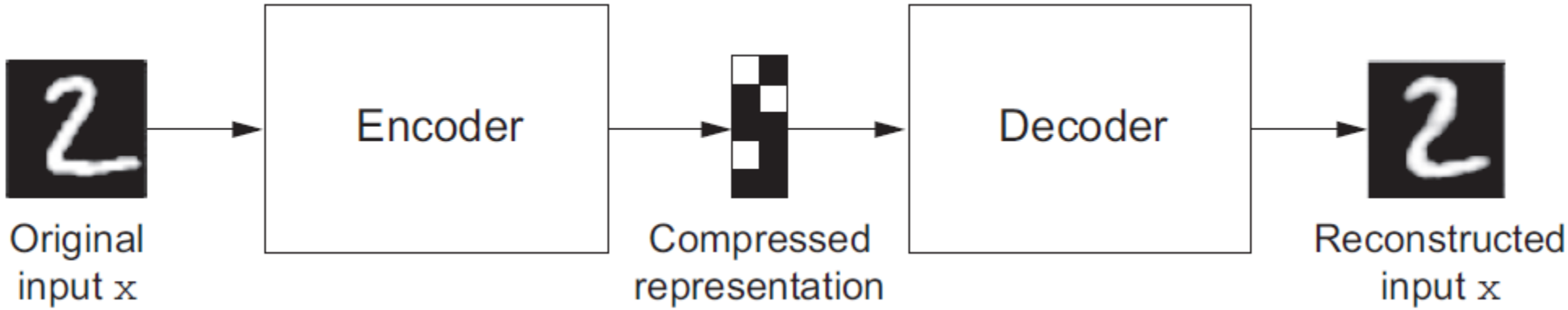
Learning a Latent Vector Space of Images



A Continuous Space of Faces [for Latent Representation: Tom White]

The Smile Vector

Standard AutoEncoder



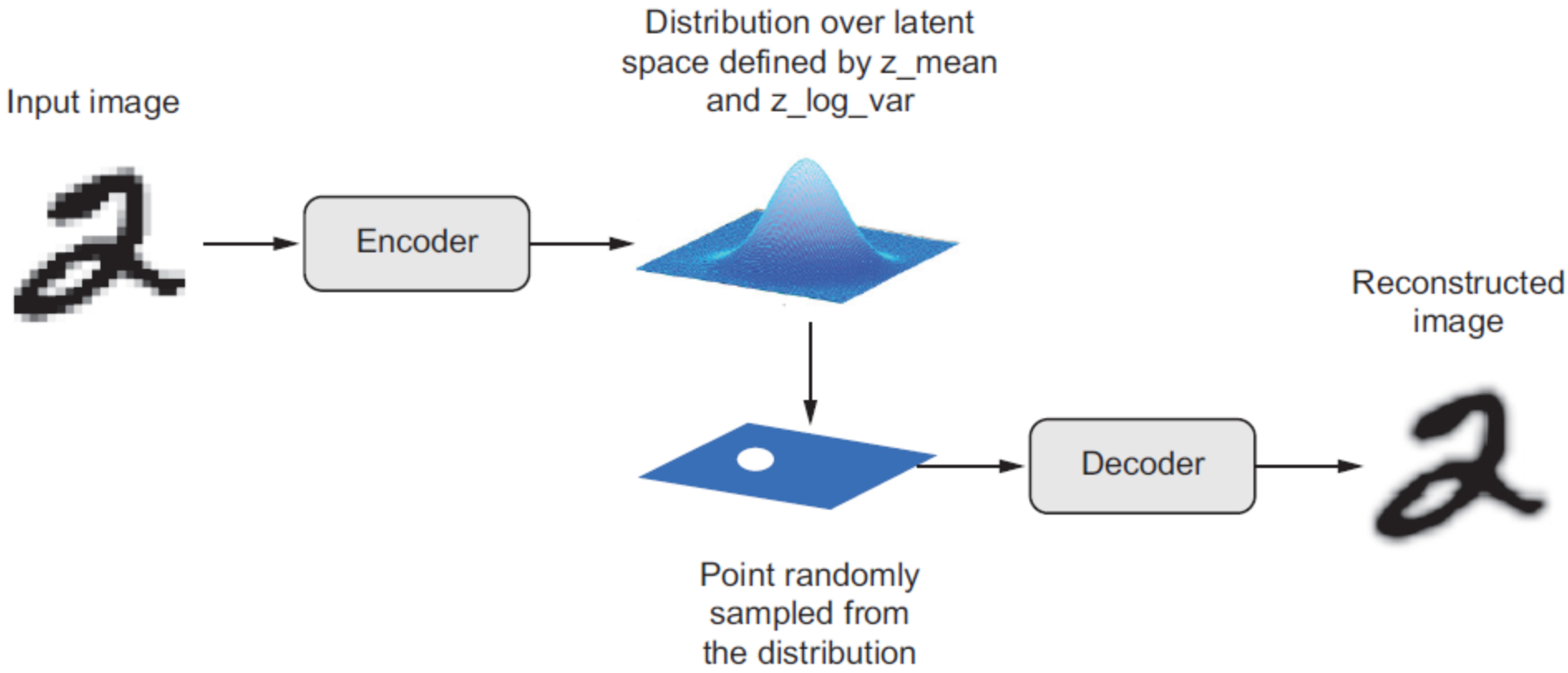
- The output to be predicted is identical to the input

- Most common [self-imposed] obstacle: dimensionality reduction; i.e. the size of the dense hidden layer must be smaller than the input

De-Noising AutoEncoder

- We add “noise” to the input pixels in the form of randomly zeroed pixels
- Typically about 50% of the pixels may be degraded in this way
- The AutoEncoder loss function is again the squared-error loss, this time between the pixels in the uncorrupted image and the output decoder image

Variational AutoEncoder



Variational AutoEncoder Processing

1. An encoder module turns the input samples `input_img` into two parameters in a latent space of representations, `z_mean` and `z_log_variance`.
2. You randomly sample a point `z` from the latent normal distribution that's assumed to generate the input image, via `z = z_mean + exp(z_log_variance) * epsilon`, where `epsilon` is a random tensor of small values.
3. A decoder module maps this point in the latent space back to the original input image.

Creating a Variational AutoEncoder

```
z_mean, z_log_variance = encoder(input_img)

z = z_mean + exp(z_log_variance) * epsilon

reconstructed_img = decoder(z)

model = Model(input_img, reconstructed_img)
```

Encodes the input into a mean and variance parameter

Draws a latent point using a small random epsilon

Decodes z back to an image

Instantiates the autoencoder model, which maps an input image to its reconstruction

Setting Up Input

```
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

input_img = keras.Input(shape=img_shape)
```

Dimensionality of the latent space: a 2D plane

Sampling

```
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])
```

Decoding

```
decoder_input = layers.Input(K.int_shape(z)[1:])

x = layers.Dense(np.prod(shape_before_flattening[1:]),
                 activation='relu')(decoder_input)

x = layers.Reshape(shape_before_flattening[1:])(x)

x = layers.Conv2DTranspose(32, 3,
                           padding='same',
                           activation='relu',
                           strides=(2, 2))(x)
x = layers.Conv2D(1, 3,
                  padding='same',
                  activation='sigmoid')(x)

decoder = Model(decoder_input, x)

z_decoded = decoder(z)
```

Input where you'll feed z

Upsamples the input

Reshapes z into a feature map of the same shape as the feature map just before the last Flatten layer in the encoder model

Uses a Conv2DTranspose layer and Conv2D layer to decode z into a feature map the same size as the original image input

Instantiates the decoder model, which turns "decoder_input" into the decoded image

Applies it to z to recover the decoded z

Encoding

```
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)
```

The input image ends up being encoded into these two parameters.

Variational AutoEncoder Loss

```
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)
        return x

y = CustomVariationalLayer()([input_img, z_decoded])
```

You implement custom layers by writing a call method.

You don't use this output, but the layer must return something.

Calls the custom layer on the input and the decoded output to obtain the final model output

Variational Loss

- Loss for the distribution parameterized by mu and sigma

```
kl_loss = -5e-4 * K.mean(
    1 + z_log_var - K.square(z_mean) - K.exp(z_log_var), axis=-1)
```

- What happens if we set mu = 0 …

  kl_loss = -5e-4 * K.mean(1 + z_log_var – 0 – K.exp(z_log_var), axis = -1)

  [setting variance to 1 minimizes kl_loss]

- What happens if we set sigma = 1 …

  kl_loss = -5e-4 * K.mean(1 + 0 – K.square(z_mean) – 1, axis = -1)

  [setting mu to 0 minimizes kl_loss]

Training the Variational AutoEncoder

```
from keras.datasets import mnist

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

(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))
```

Sampling from the 2D Latent Space

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

n = 15
digit_size = 28
figure = np.zeros((digit_size * n, digit_size * n))
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()
```

You'll display a grid of 15 × 15 digits (255 digits total).

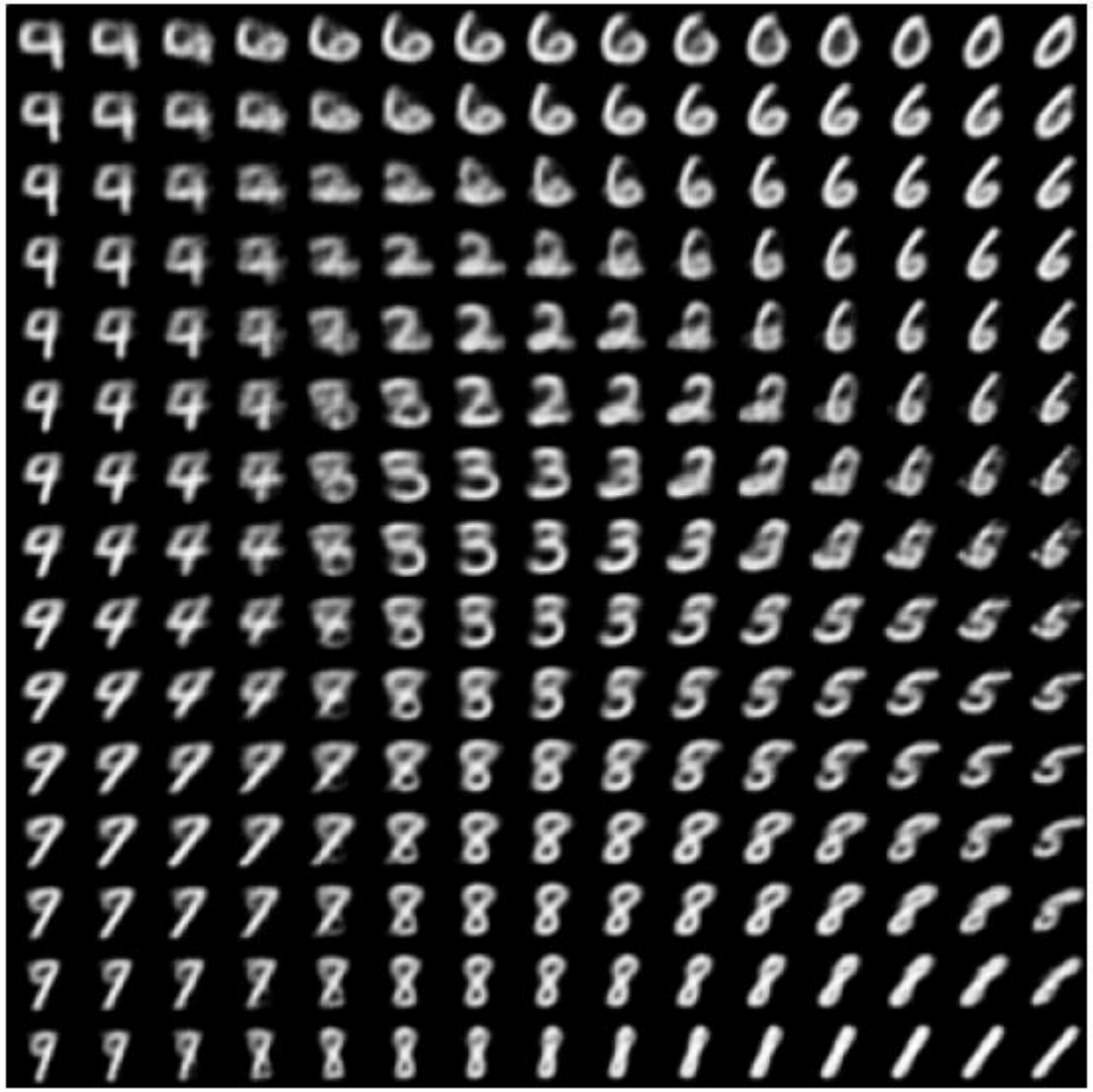
Transforms linearly spaced coordinates using the SciPy ppf function to produce values of the latent variable z (because the prior of the latent space is Gaussian)

Repeats z multiple times to form a complete batch

Reshapes the first digit in the batch from 28 × 28 × 1 to 28 × 28

Decodes the batch into digit images

Grid of Digits Decoded from the Latent Space



Wrapping Up

- Image generation with deep learning is done by learning latent spaces that capture statistical information about a dataset of images. By sampling and decoding points from the latent space, you can generate never-before-seen images. There are two major tools to do this: VAEs and GANs.
- VAEs result in highly structured, continuous latent representations. For this reason, they work well for doing all sorts of image editing in latent space: face swapping, turning a frowning face into a smiling face, and so on. They also work nicely for doing latent-space-based animations, such as animating a walk along a cross section of the latent space, showing a starting image slowly morphing into different images in a continuous way.
- GANs enable the generation of realistic single-frame images but may not induce latent spaces with solid structure and high continuity.

Celebrity Faces Attributes (CelebA) Dataset

TIP To play further with image generation, I suggest working with the Large-scale Celeb Faces Attributes (CelebA) dataset. It's a free-to-download image dataset containing more than 200,000 celebrity portraits. It's great for experimenting with concept vectors in particular—it definitely beats MNIST.

10,177 number of **identities**,

202,599 number of **face images**, and

5 landmark locations, **40 binary attributes** annotations per image.

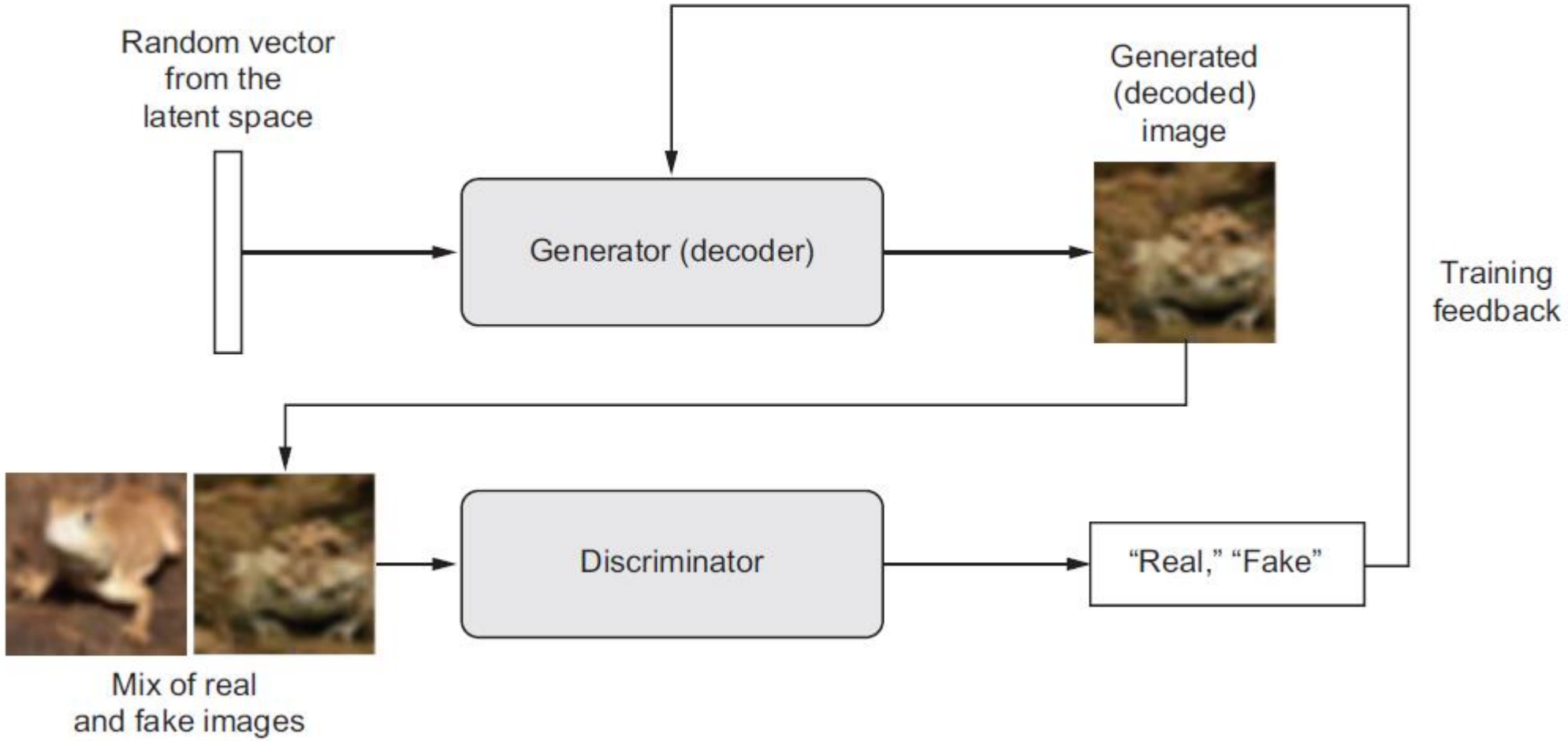
http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html

Generative Adversarial Network (GAN)

A GAN consists of two networks …

- *Generator network*—Takes as input a random vector (a random point in the latent space), and decodes it into a synthetic image
- *Discriminator network (or adversary)*—Takes as input an image (real or synthetic), and predicts whether the image came from the training set or was created by the generator network.

Generator (G) versus Discriminator (D)



Examples of GAN Outputs

GAN Process for CIFAR10 “Frog” Class

1. A `generator` network maps vectors of shape `(latent_dim,)` to images of shape `(32, 32, 3)`.
2. A `discriminator` network maps images of shape `(32, 32, 3)` to a binary score estimating the probability that the image is real.
3. A `gan` network chains the generator and the discriminator together: `gan(x) = discriminator(generator(x))`. Thus this `gan` network maps latent space vectors to the discriminator’s assessment of the realism of these latent vectors as decoded by the generator.
4. You train the discriminator using examples of real and fake images along with “real”/“fake” labels, just as you train any regular image-classification model.
5. To train the generator, you use the gradients of the generator’s weights with regard to the loss of the `gan` model. This means, at every step, you move the weights of the generator in a direction that makes the discriminator more likely to classify as “real” the images decoded by the generator. In other words, you train the generator to fool the discriminator.

A Bag of "Tricks" (Suggestions) for GANs: 1 of 2

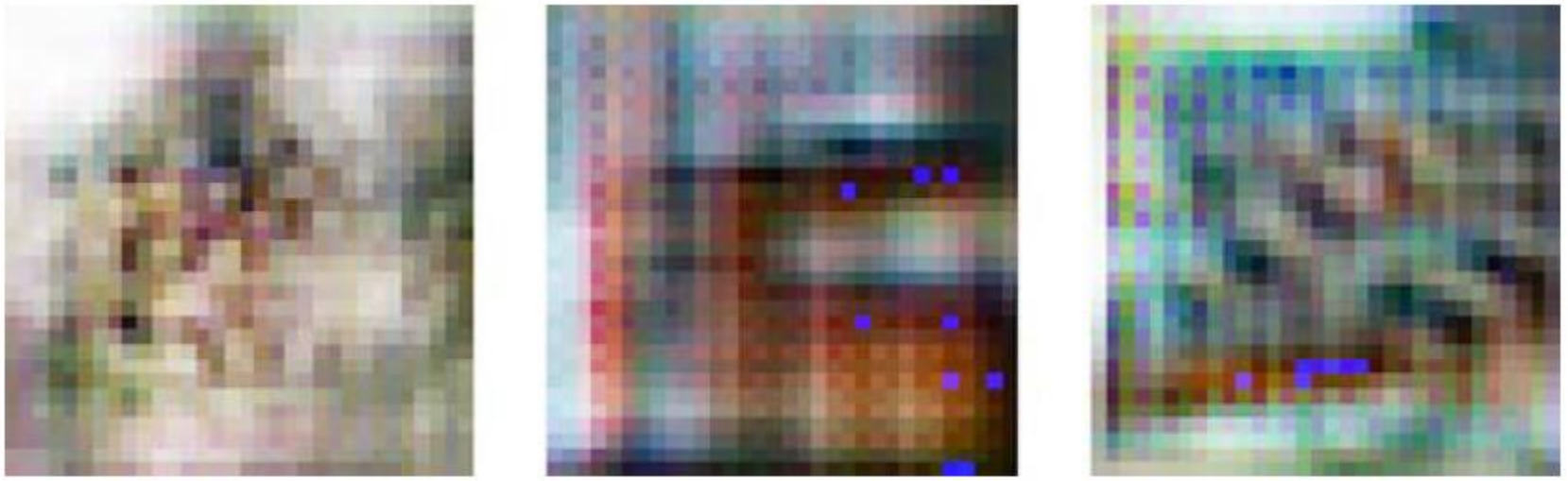
- We use `tanh` as the last activation in the generator, instead of `sigmoid`, which is more commonly found in other types of models.
- We sample points from the latent space using a *normal distribution* (Gaussian distribution), not a uniform distribution.
- Stochasticity is good to induce robustness. Because GAN training results in a dynamic equilibrium, GANs are likely to get stuck in all sorts of ways. Introducing randomness during training helps prevent this. We introduce randomness in two ways: by using dropout in the discriminator and by adding random noise to the labels for the discriminator.

A Bag of "Tricks" (Suggestions) for GANs: 2 of 2

- Sparse gradients can hinder GAN training. In deep learning, sparsity is often a desirable property, but not in GANs. Two things can induce gradient sparsity: max pooling operations and `ReLU` activations. Instead of max pooling, we recommend using strided convolutions for downsampling, and we recommend using a `LeakyReLU` layer instead of a `ReLU` activation. It's similar to `ReLU`, but it relaxes sparsity constraints by allowing small negative activation values.
- In generated images, it's common to see checkerboard artifacts caused by unequal coverage of the pixel space in the generator (see figure 8.17). To fix this, we use a kernel size that's divisible by the stride size whenever we use a strided `Conv2DTranpose` or `Conv2D` in both the generator and the discriminator.

Examples of Checkerboarding Caused by Mismatched Kernel Size and Stride

Unequal pixel coverage results in checkerboarding …



Preliminaries

```
import keras
from keras import layers
import numpy as np

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

Code for Generator

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

x = layers.Dense(128 * 16 * 16)(generator_input)
x = layers.LeakyReLU()(x)
x = layers.Reshape((16, 16, 128))(x)

x = layers.Conv2D(256, 5, padding='same')(x)
x = layers.LeakyReLU()(x)

x = layers.Conv2DTranspose(256, 4, strides=2, padding='same')(x)
x = layers.LeakyReLU()(x)

x = layers.Conv2D(256, 5, padding='same')(x)
x = layers.LeakyReLU()(x)
x = layers.Conv2D(256, 5, padding='same')(x)
x = layers.LeakyReLU()(x)

x = layers.Conv2D(channels, 7, activation='tanh', padding='same')(x)
generator = keras.models.Model(generator_input, x)
generator.summary()
```

Transforms the input into a 16 × 16 128-channel feature map

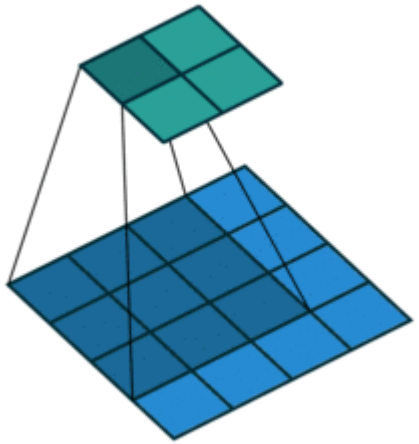
Upsamples to 32 × 32

Produces a 32 × 32 1-channel feature map (shape of a CIFAR10 image)

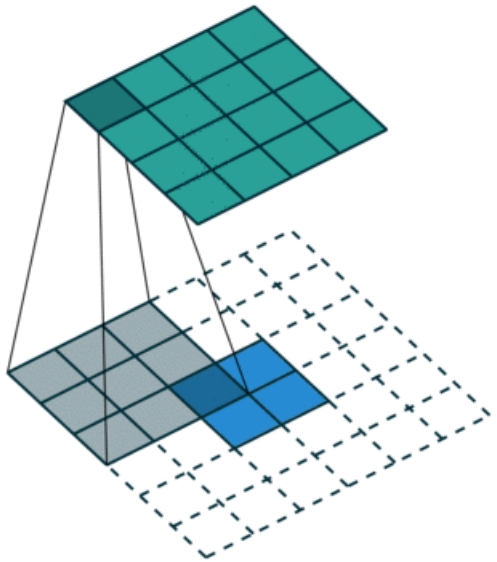
Instantiates the generator model, which maps the input of shape (latent_dim,) into an image of shape (32, 32, 3)

Transposed Convolution with Valid Padding [input and output sizes have been transposed]

- Convolution
  - Light-blue input: 4 x 4
  - Gray filter: 3 x 3
  - Light-green output: 2 x 2



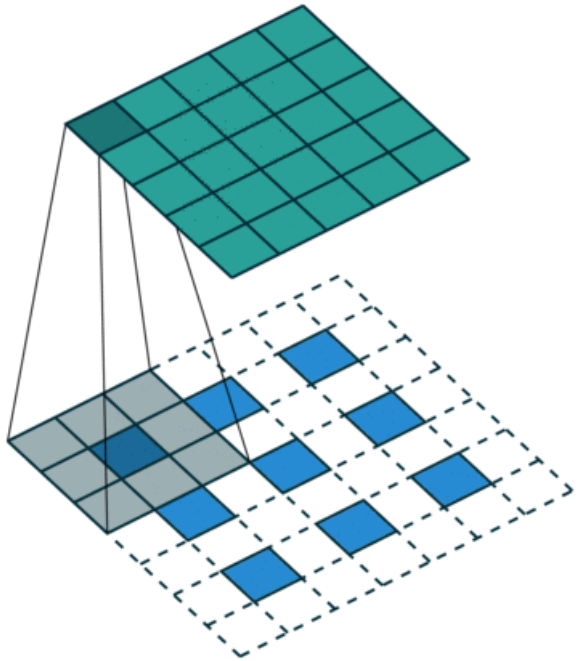
- Transposed Convolution
  - Light-blue input: 2 x 2
  - Gray filter: 3 x 3
  - Light-green output: 4 x 4



http://deeplearning.net/software/theano/tutorial/conv_arithmetic.html

Transposed Convolution with Stride > 1

Transpose of convolving a 3x3 kernel over a 5x5 input padded with a 1x1 border of zeros using 2x2 strides



http://deeplearning.net/software/theano/tutorial/conv_arithmetic.html

Code for 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)

x = layers.Dropout(0.4)(x)

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

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

discriminator_optimizer = keras.optimizers.RMSprop(
    lr=0.0008,
    clipvalue=1.0,
    decay=1e-8)

discriminator.compile(optimizer=discriminator_optimizer,
                      loss='binary_crossentropy')
```

One dropout layer: an important trick!

Classification layer

Instantiates the discriminator model, which turns a (32, 32, 3) input into a binary classifi-cation decision (fake/real)

Uses gradient clipping (by value) in the optimizer

To stabilize training, uses learning-rate decay

Adversarial Network

```
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')
```

Sets discriminator weights to non-trainable (this will only apply to the gan model)

Training the Deep Convolutional GAN (DCGAN)

1. Draw random points in the latent space (random noise).
2. Generate images with `generator` using this random noise.
3. Mix the generated images with real ones.
4. Train `discriminator` using these mixed images, with corresponding targets: either "real" (for the real images) or "fake" (for the generated images).
5. Draw new random points in the latent space.
6. Train `gan` using these random vectors, with targets that all say "these are real images." This updates the weights of the generator (only, because the discriminator is frozen inside `gan`) to move them toward getting the discriminator to predict "these are real images" for generated images: this trains the generator to fool the discriminator.

Code for Training the GAN: 1 of 3

```
import os
from keras.preprocessing import image

(x_train, y_train), (_, _) = keras.datasets.cifar10.load_data()

x_train = x_train[y_train.flatten() == 6]

x_train = x_train.reshape(
    (x_train.shape[0],) +
    (height, width, channels)).astype('float32') / 255.

iterations = 10000
batch_size = 20
save_dir = 'your_dir'

start = 0
for step in range(iterations):
    random_latent_vectors = np.random.normal(size=(batch_size,
                                                   latent_dim))
```

Loads CIFAR10 data

Selects frog images (class 6)

Normalizes data

Specifies where you want to save generated images

Samples random points in the latent space

Code for Training the GAN: 2 of 3

```
generated_images = generator.predict(random_latent_vectors)

stop = start + batch_size
real_images = x_train[start: stop]
combined_images = np.concatenate([generated_images, real_images])

labels = np.concatenate([np.ones((batch_size, 1)),
                         np.zeros((batch_size, 1))])
labels += 0.05 * np.random.random(labels.shape)

d_loss = discriminator.train_on_batch(combined_images, labels)

random_latent_vectors = np.random.normal(size=(batch_size,
                                               latent_dim))

misleading_targets = np.zeros((batch_size, 1))

a_loss = gan.train_on_batch(random_latent_vectors,
                            misleading_targets)

start += batch_size
if start > len(x_train) - batch_size:
  start = 0

if step % 100 == 0:
    gan.save_weights('gan.h5')
```

Decodes them to fake images

Combines them with real images

Assembles labels, discriminating real from fake images

Adds random noise to the labels—an important trick!

Trains the discriminator

Samples random points in the latent space

Assembles labels that say "these are all real images" (it's a lie!)

Trains the generator (via the gan model, where the discriminator weights are frozen)

Occasionally saves and plots (every 100 steps)

Saves model weights

Code for Training the GAN: 3 of 3

Prints metrics

```
print('discriminator loss:', d_loss)
print('adversarial loss:', a_loss)

img = image.array_to_img(generated_images[0] * 255., scale=False)
img.save(os.path.join(save_dir,
                      'generated_frog' + str(step) + '.png'))

img = image.array_to_img(real_images[0] * 255., scale=False)
img.save(os.path.join(save_dir,
                      'real_frog' + str(step) + '.png'))
```

Saves one generated image

Saves one real image for comparison

GAN versus Real?

Left-to-Right: “real” is Middle, Top, Bottom, Middle; others are “fake”

Wrapping Up

- A GAN consists of a generator network coupled with a discriminator network. The discriminator is trained to differentiate between the output of the generator and real images from a training dataset, and the generator is trained to fool the discriminator. Remarkably, the generator never sees images from the training set directly; the information it has about the data comes from the discriminator.
- GANs are difficult to train, because training a GAN is a dynamic process rather than a simple gradient descent process with a fixed loss landscape. Getting a GAN to train correctly requires using a number of heuristic tricks, as well as extensive tuning.
- GANs can potentially produce highly realistic images. But unlike VAEs, the latent space they learn doesn't have a neat continuous structure and thus may not be suited for certain practical applications, such as image editing via latent-space concept vectors.

Chapter Summary

- With creative applications of deep learning, deep networks go beyond annotating existing content and start generating their own. You learned the following:
  - How to generate sequence data, one timestep at a time. This is applicable to text generation and also to note-by-note music generation or any other type of timeseries data.
  - How DeepDream works: by maximizing convnet layer activations through gradient ascent in input space.
  - How to perform style transfer, where a content image and a style image are combined to produce interesting-looking results.
  - What GANs and VAEs are, how they can be used to dream up new images, and how latent-space concept vectors can be used for image editing.
- These few techniques cover only the basics of this fast-expanding field. There's a lot more to discover out there—generative deep learning is deserving of an entire book of its own.

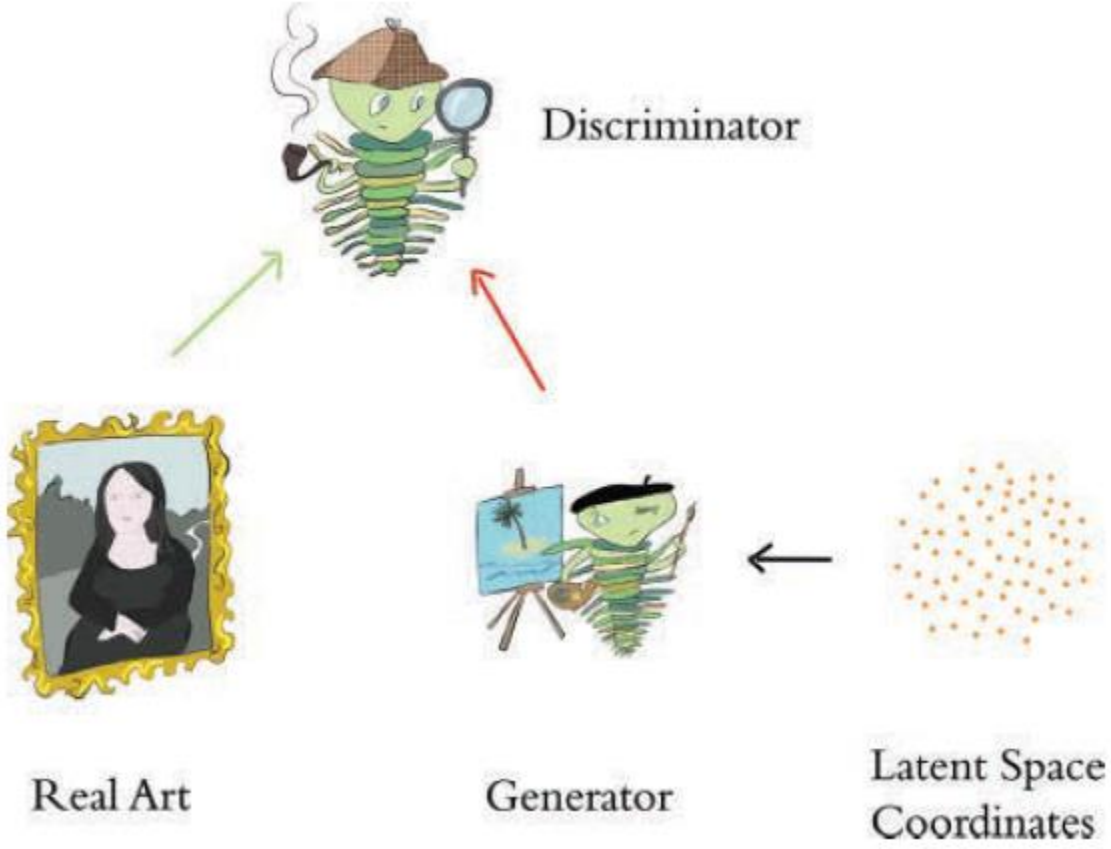
[DLI] Machine Art

- An All-Nighter
- Arithmetic on Fake Human Faces
- Style Transfer: Converting Photos into Monet (and Vice Versa)
- Make Your Own Sketches Photorealistic
- Creating Photorealistic Images from Text
- Image Processing Using Deep Learning
- Summary

Generative Adversarial Network (GAN)

Q: What's better than training one model?

A: Training two models!



Samples from a GAN

Cut off the right-most column, showing nearest neighbor from training set

Modified National Institutes of Standards and Technology (MNIST)

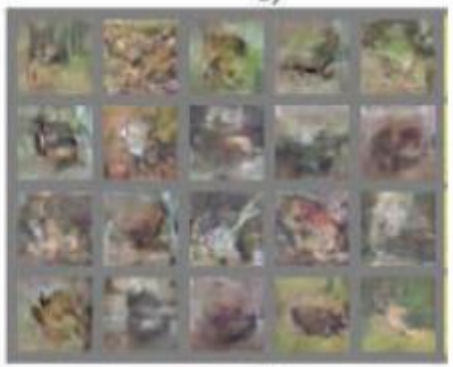
a)

b)

Toronto Face Database (TFD)

Canadian Institute For Advanced Research (CIFAR-10) [fully connected]

c)

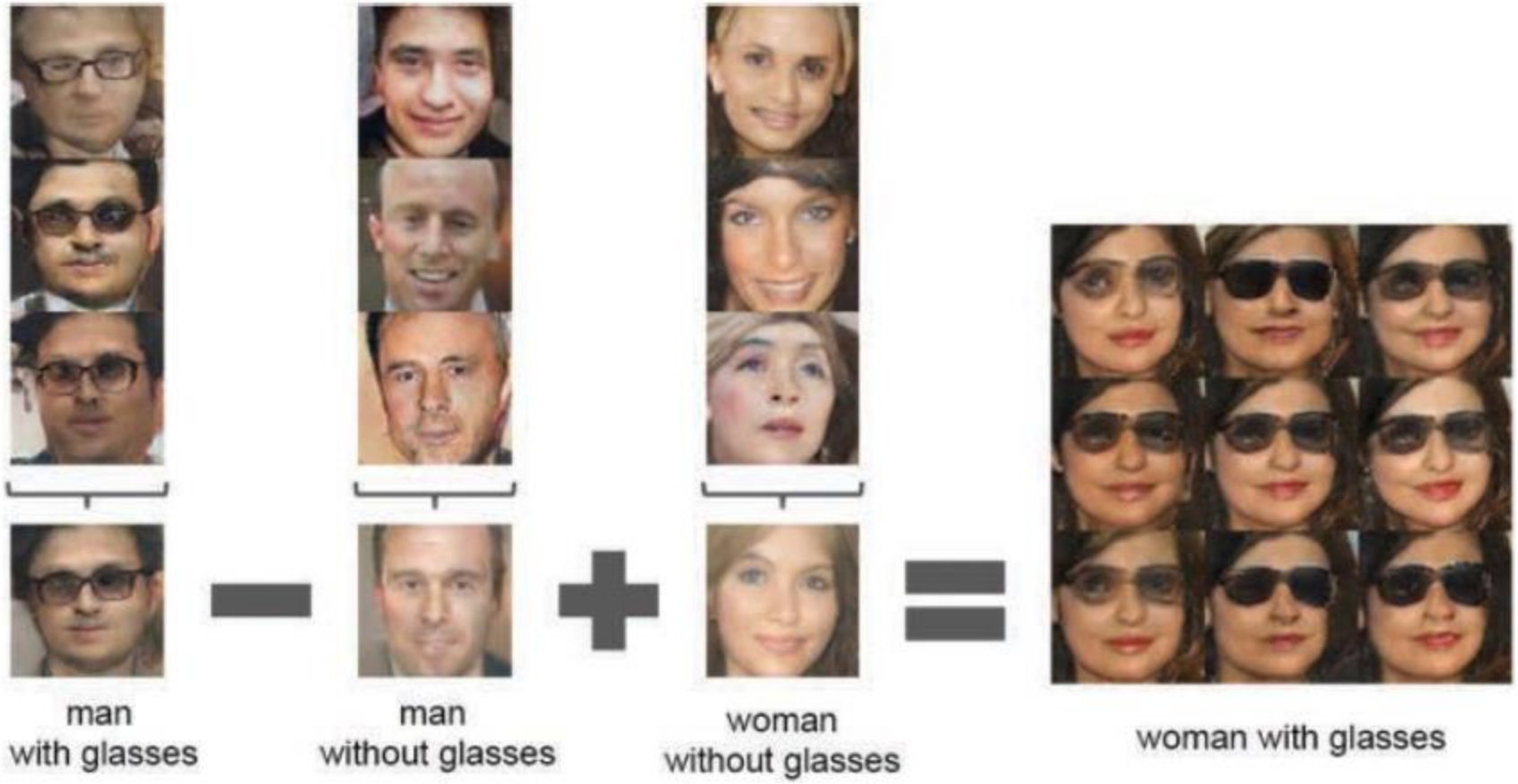


d)

CIFAR-10 [convolutional Discriminator] [deconvolutional Generator]

https://arxiv.org/pdf/1406.2661.pdf

Latent Space Arithmetic



For each column, the Z vectors of samples are averaged. Arithmetic was then performed on the mean vectors creating a new vector Y. The center sample on the right-hand side is produce by feeding Y as input to the generator. To demonstrate the interpolation capabilities of the generator, uniform noise sampled with scale ±0.25 was added to Y to produce the 8 other samples.

Cartoon Depicting Age, Gender, and Glasses Dimensions for Latent Space of GAN

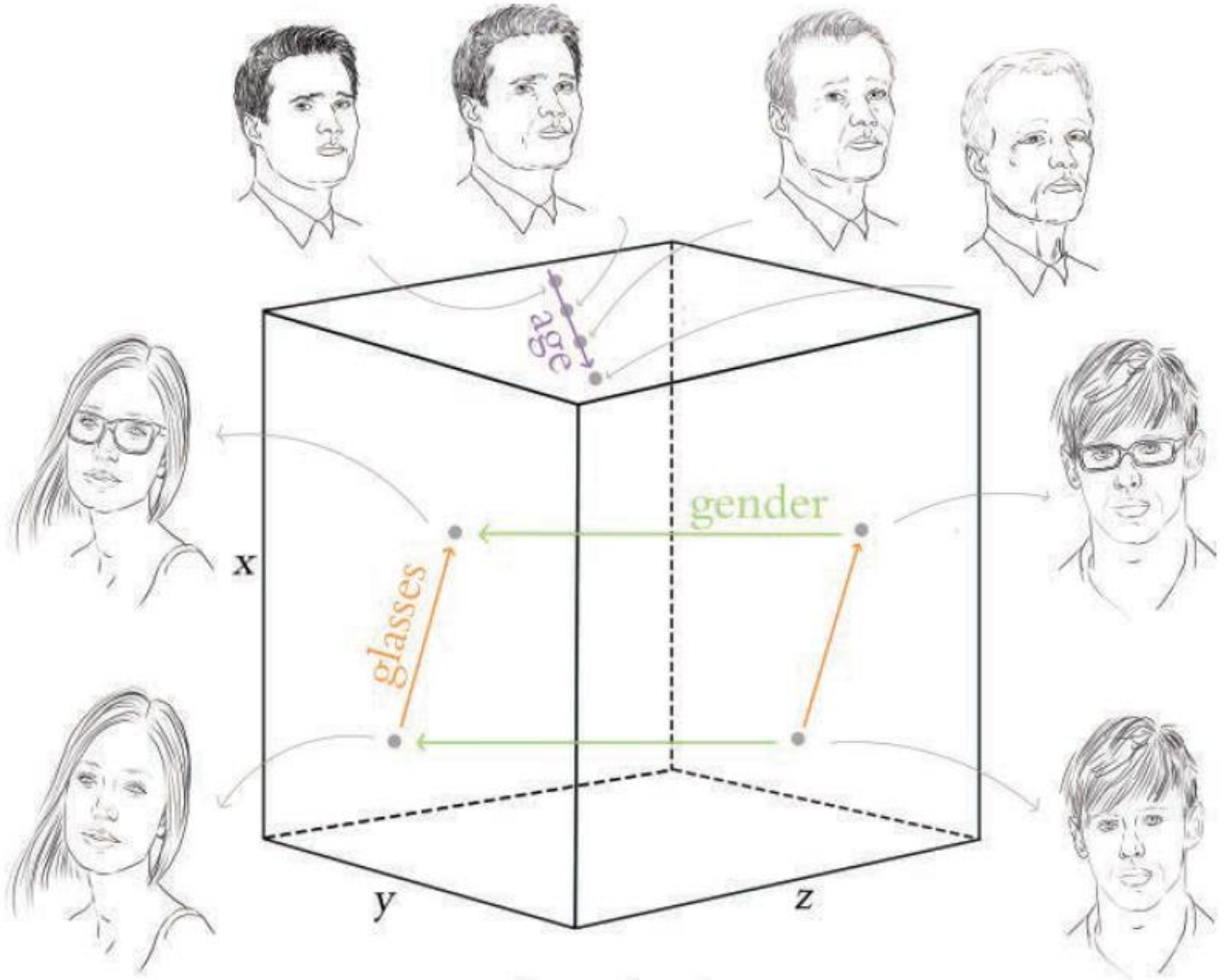
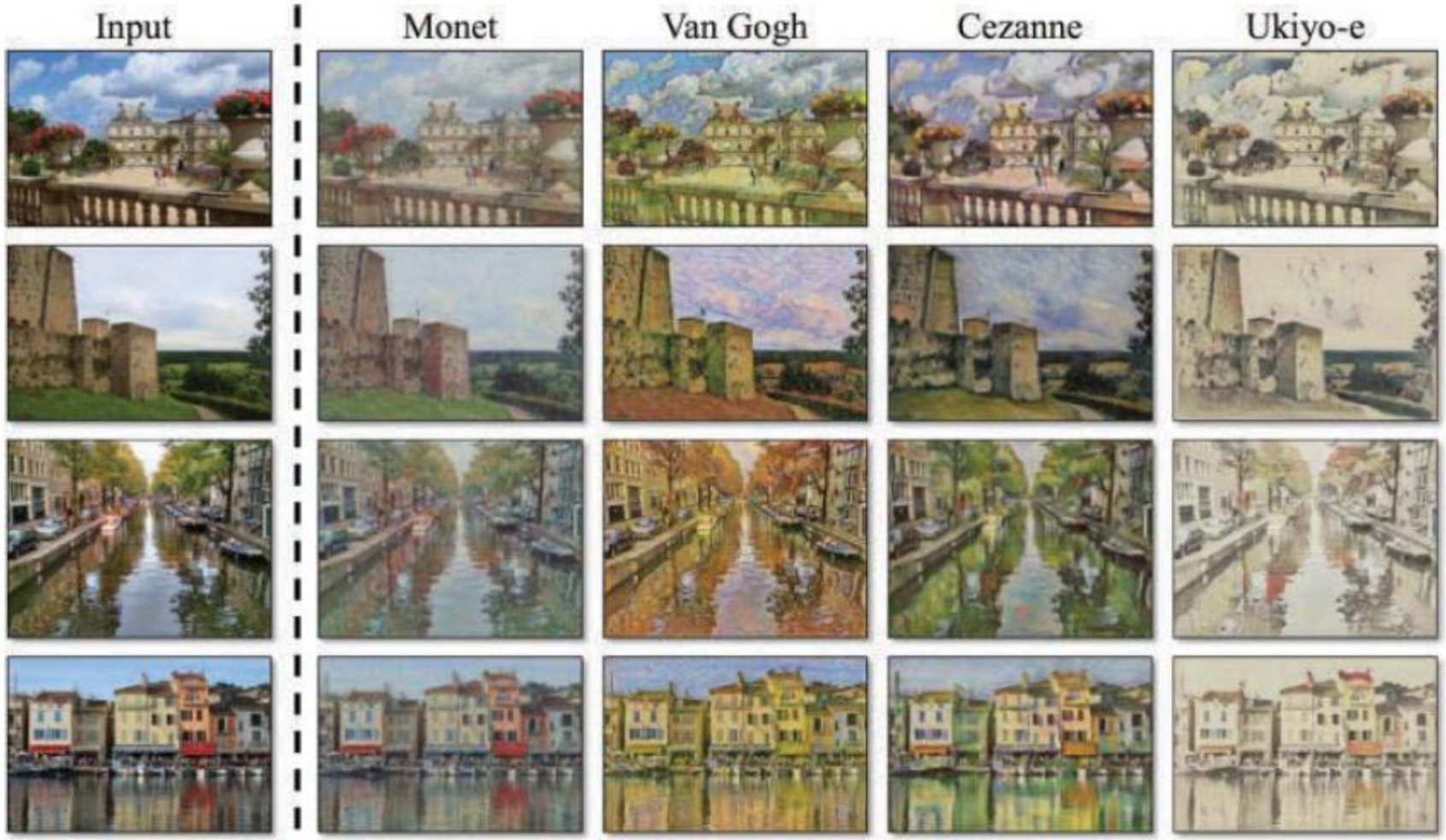
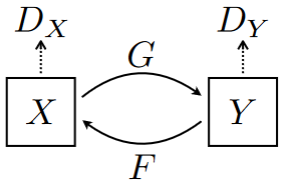


Photo + CycleGAN Image to Generate Output

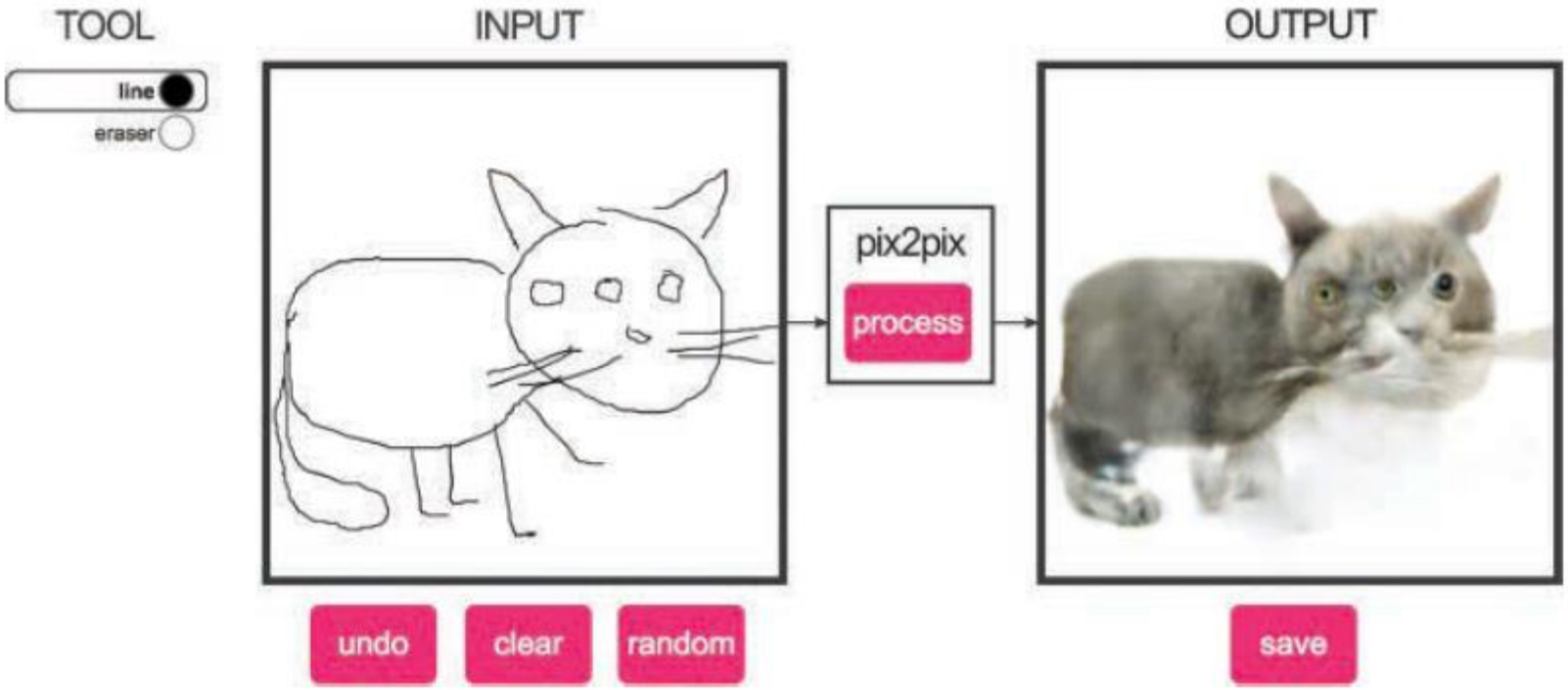


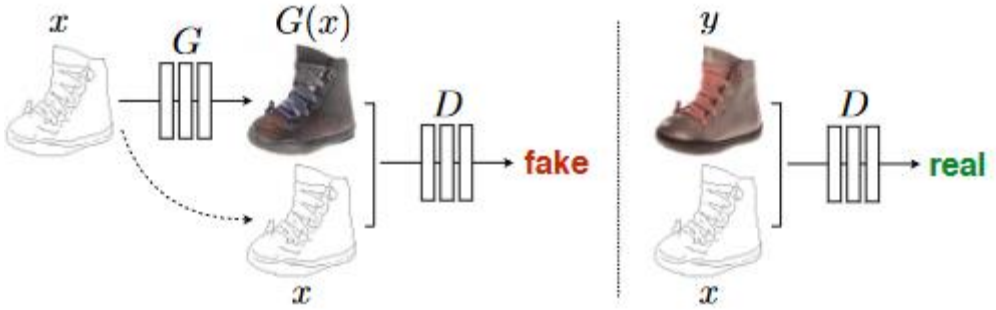
mo nay
van go
say zahn
you key oh ay



https://arxiv.org/pdf/1703.10593.pdf

Pix2Pix Demo



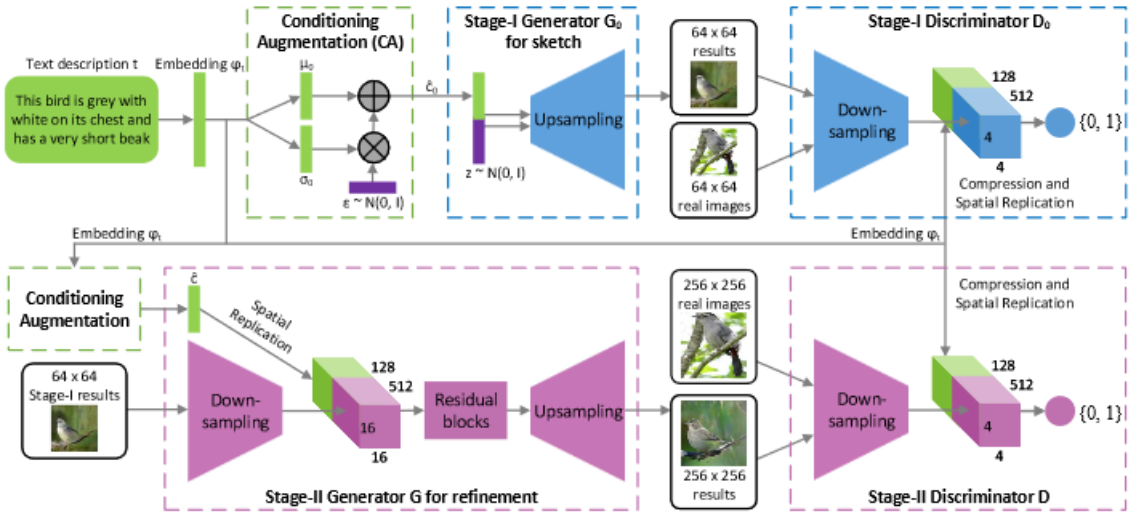


https://arxiv.org/pdf/1611.07004.pdf

StackGAN Examples

First GAN:
text to 64x64 image

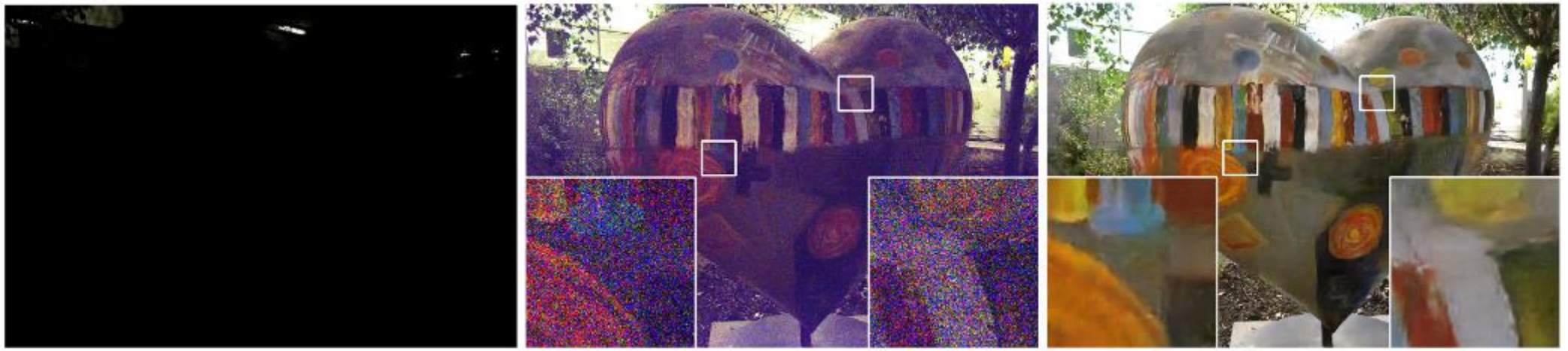
Second GAN:
64x64 image to 256x256 image



https://arxiv.org/pdf/1612.03242.pdf

Image Processing Example

- Left: short exposure image
- Center: traditional processing pipeline
- Right: image processed by U-net

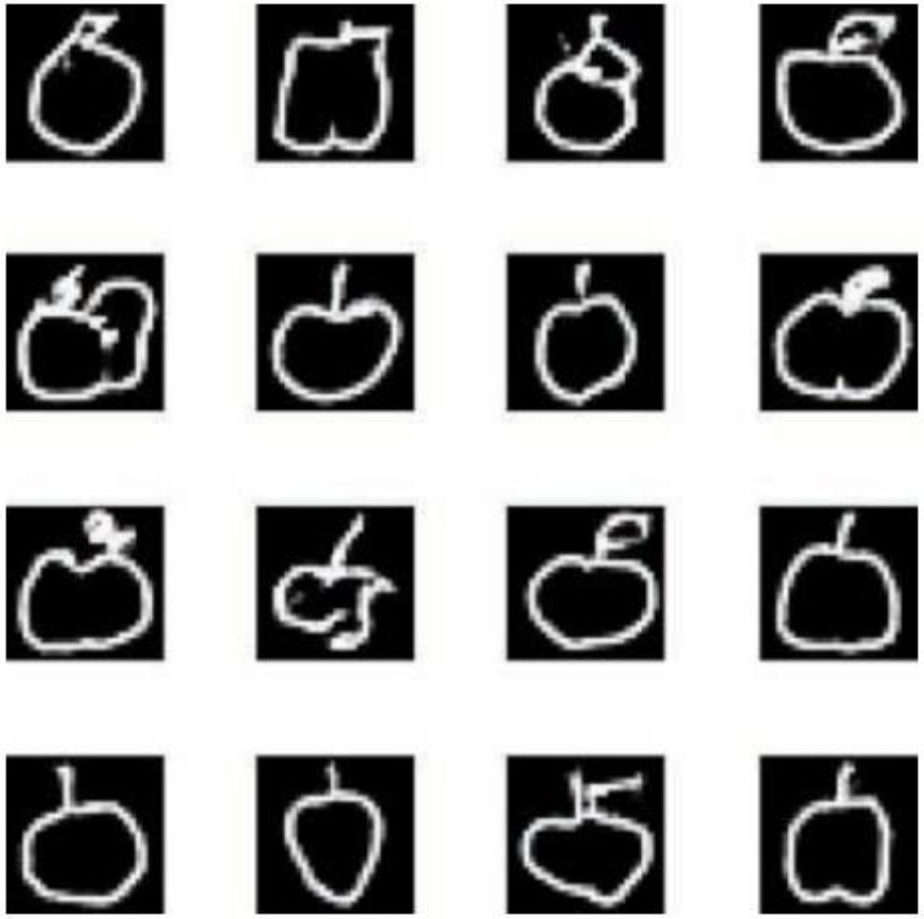


Learning to See in the Dark: trained a fully convolutional network to perform the entire image processing pipeline

https://arxiv.org/pdf/1805.01934.pdf

“Hand” Drawings of Apples

GAN trained on “Quick, Draw!” images

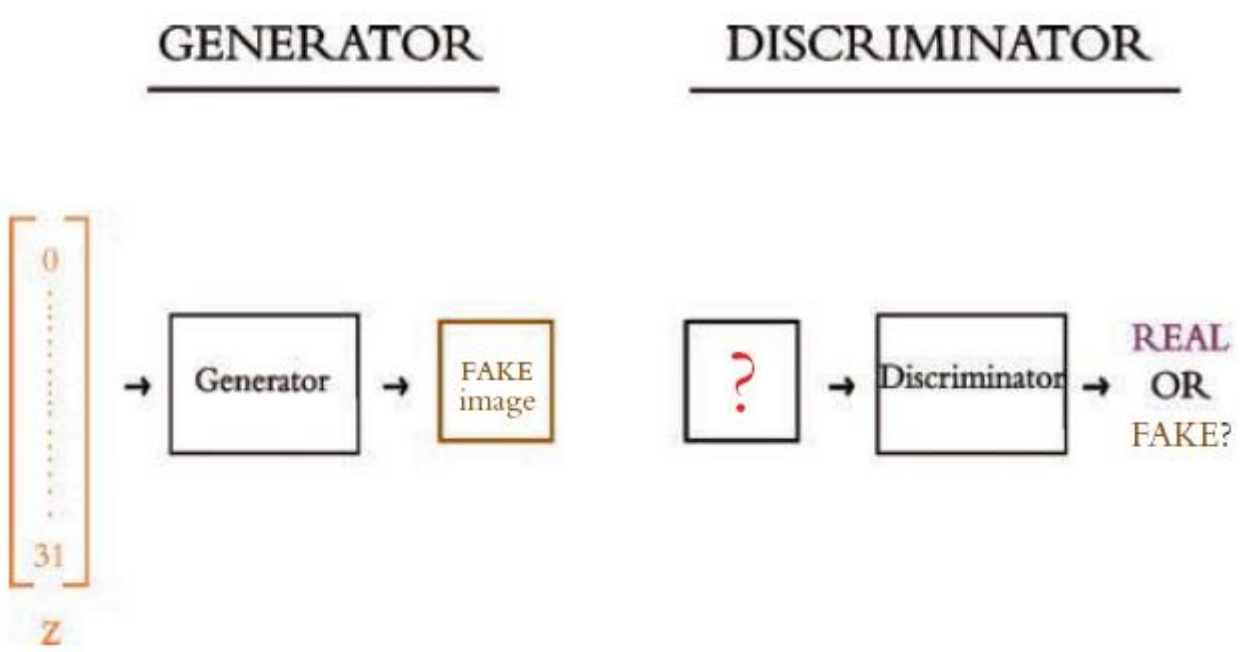


[DLI] Generative Adversarial Networks

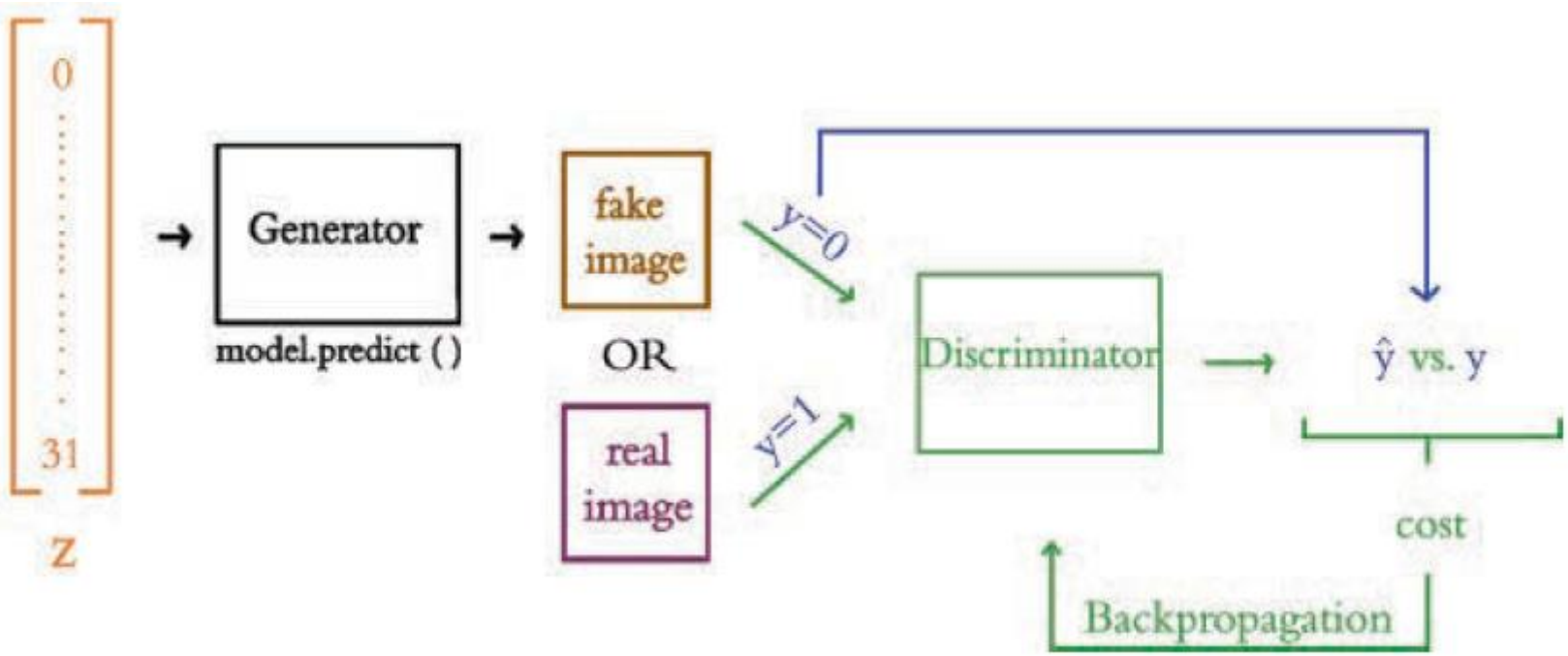
- Essential GAN Theory
- The Quick, Draw! Dataset
- The Discriminator Network
- The Generator Network
- The Adversarial Network
- GAN Training
- Summary

Generator versus Discriminator

The two models that make up a GAN

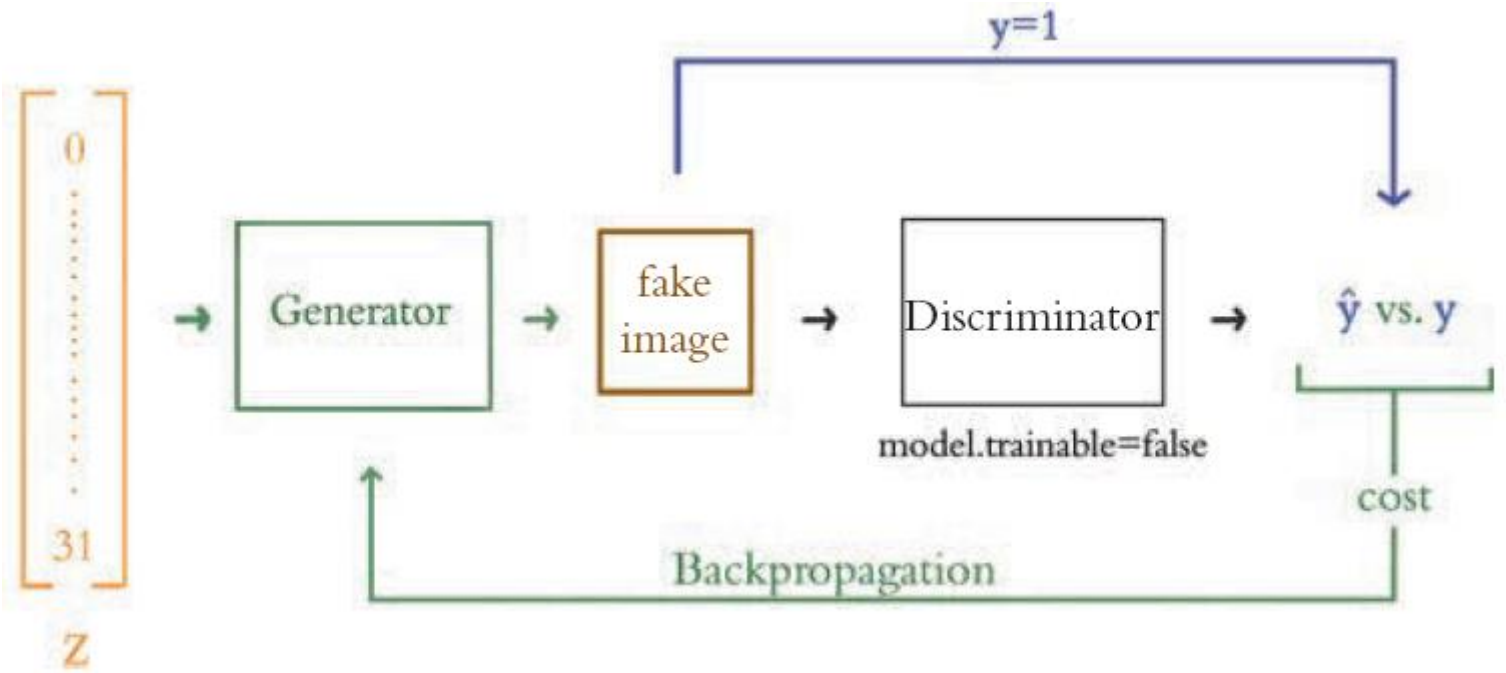


Training the Discriminator



Only the weights of the Discriminator are updated

Training the Generator



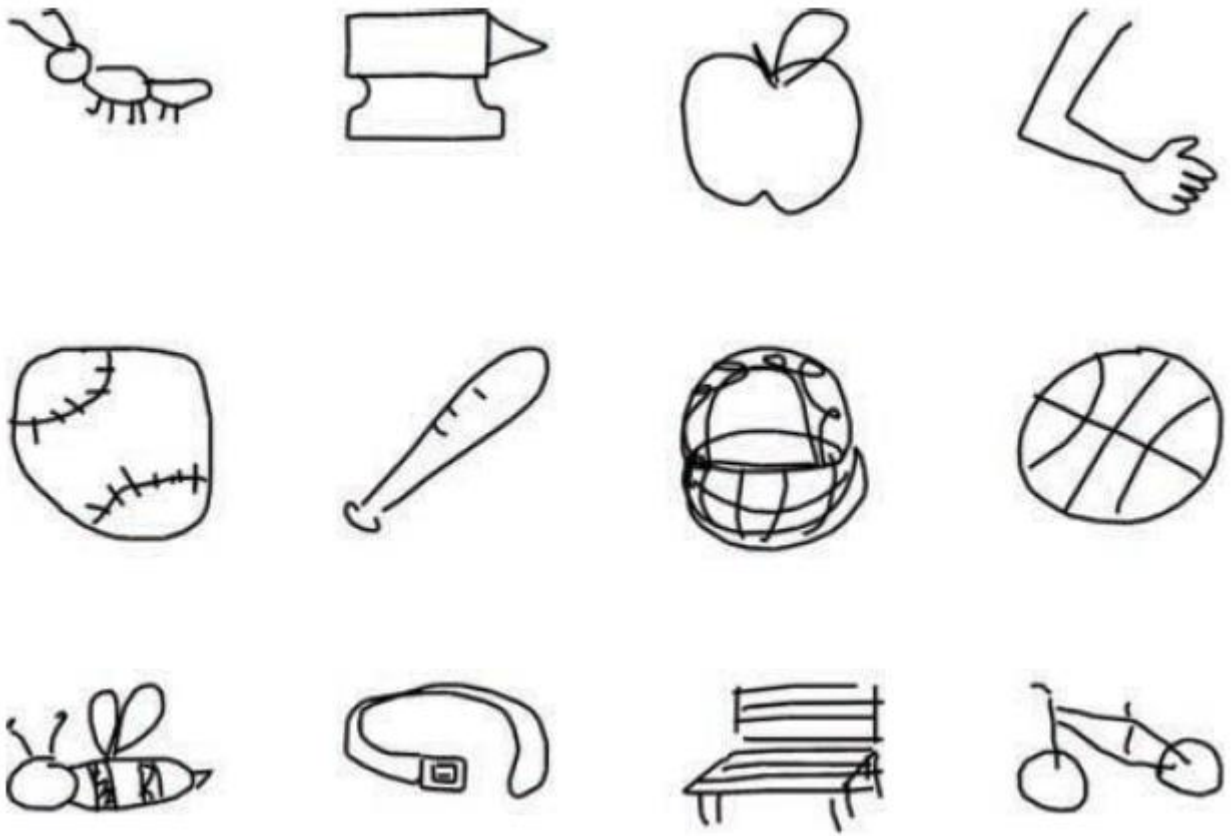
Error is backpropagated across the Discriminator, but only the weights of the Generator are updated

Training

while not done:

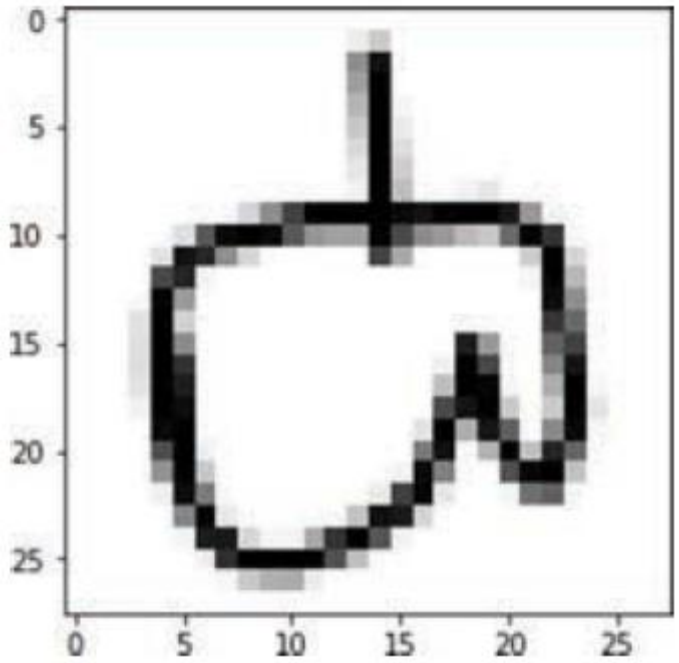
- train the discriminator
  - minimize log loss for real/fake labels
  - only update discriminator weights
- train the generator
  - minimize log loss for “real” labels
  - backprop across both models
  - but only update generator weights

Sketches Drawn by Humans



http://bit.ly/QDdata

Bitmap Example



Shape: (28, 28, 1)

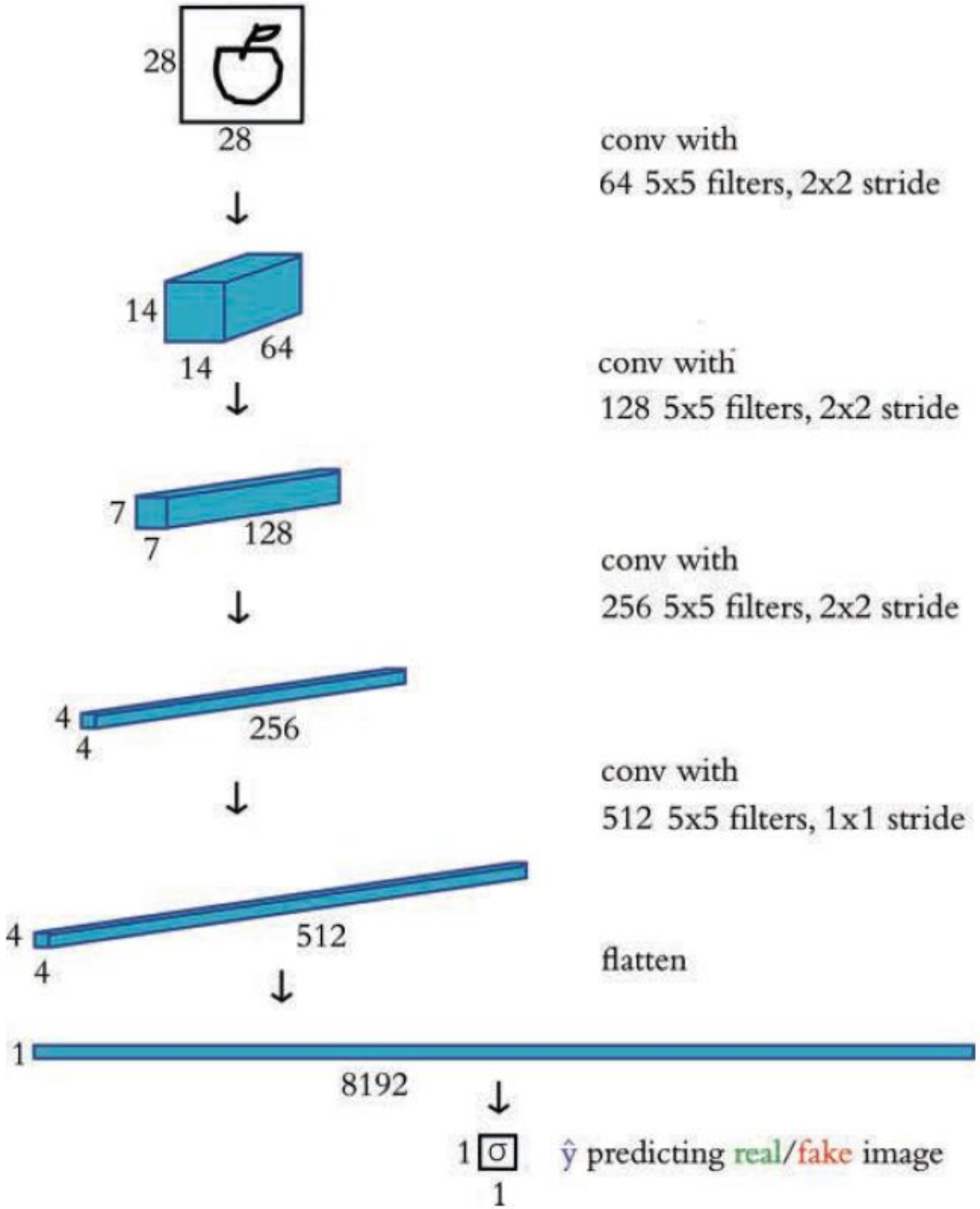
Quick, Draw! GAN Dependencies

```
# for data input and output:
import numpy as np
import os
# for deep learning:
import keras
from keras.models import Model
from keras.layers import Input, Dense, Conv2D, Dropout
from keras.layers import BatchNormalization, Flatten
from keras.layers import Activation
from keras.layers import Reshape # new!
from keras.layers import Conv2DTranspose, UpSampling2D # new!
from keras.optimizers import RMSprop # new!
# for plotting:
import pandas as pd
from matplotlib import pyplot as plt
%matplotlib inline
```

The Discriminator

```
def build_discriminator(depth=64, p=0.4):
    # Define inputs
    image = Input((img_w,img_h,1))
    # Convolutional layers
    conv1 = Conv2D(depth*1, 5, strides=2,
        padding='same',
        activation='relu')(image)
    conv1 = Dropout(p)(conv1)
    conv2 = Conv2D(depth*2, 5, strides=2,
        padding='same',
        activation='relu')(conv1)
    conv2 = Dropout(p)(conv2)
    conv3 = Conv2D(depth*4, 5, strides=2,
        padding='same', activation='relu')(conv2)
    conv3 = Dropout(p)(conv3)
    conv4 = Conv2D(depth*8, 5, strides=1,
        padding='same', activation='relu')(conv3)
    conv4 = Flatten()(Dropout(p)(conv4))
    # Output layer
    prediction = Dense(1,
        activation='sigmoid')(conv4)
    # Model definition
    model = Model(inputs=image,
        outputs=prediction)
    return model
```

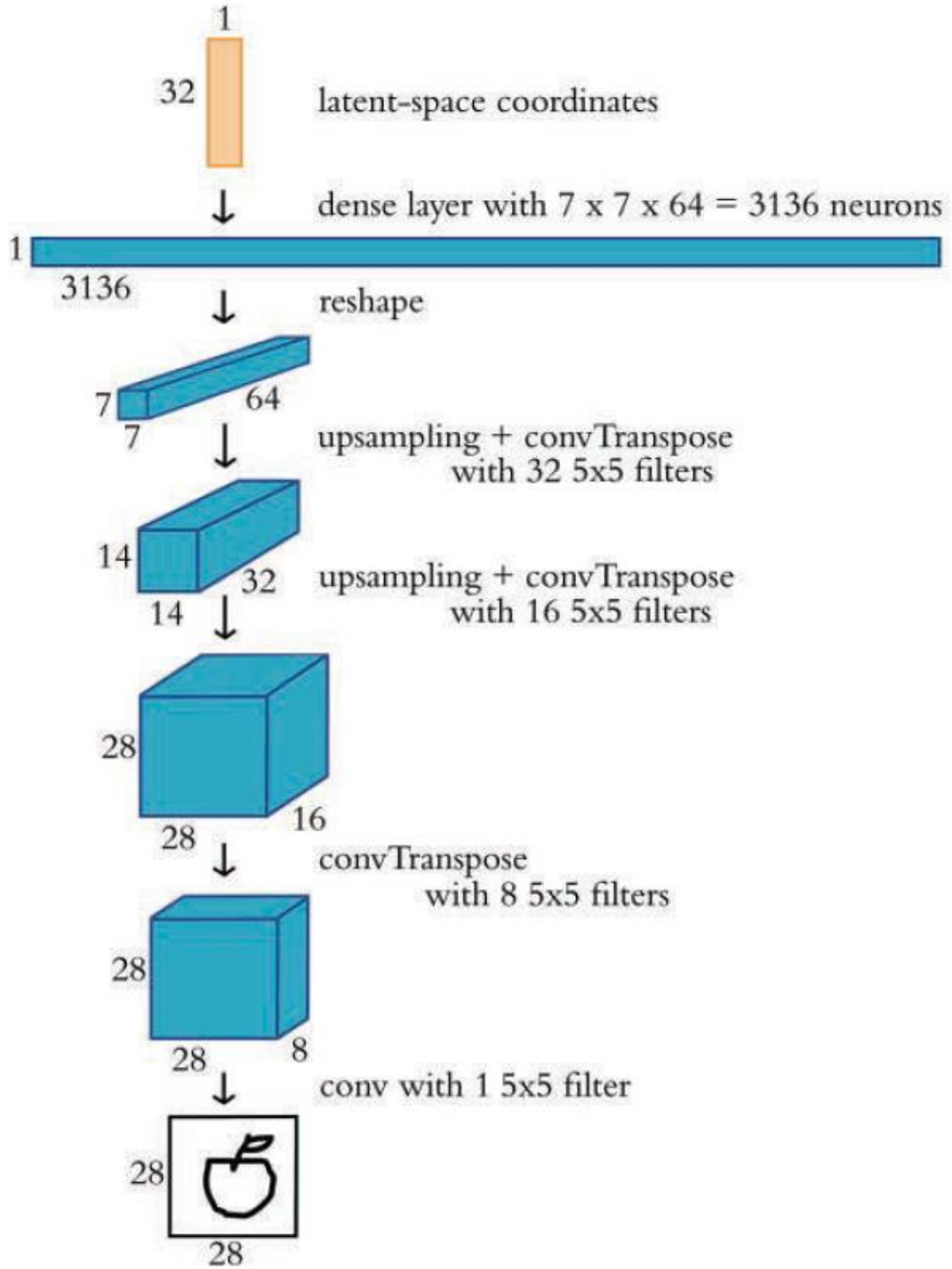
The Discriminator Network



The Generator

```
z_dimensions = 32
def build_generator(latent_dim=z_dimensions,
    depth=64, p=0.4):
    # Define inputs
    noise = Input((latent_dim,))
    # First dense layer
    dense1 = Dense(7*7*depth)(noise)
    dense1 = BatchNormalization(momentum=0.9)(dense1)
    dense1 = Activation(activation='relu')(dense1)
    dense1 = Reshape((7,7,depth))(dense1)
    dense1 = Dropout(p)(dense1)
    # De-Convolutional layers
    conv1 = UpSampling2D()(dense1)
    conv1 = Conv2DTranspose(int(depth/2),
        kernel_size=5, padding='same',
        activation=None,)(conv1)
    conv1 = BatchNormalization(momentum=0.9)(conv1)
    conv1 = Activation(activation='relu')(conv1)
    conv2 = UpSampling2D()(conv1)
    conv2 = Conv2DTranspose(int(depth/4),
        kernel_size=5, padding='same',
        activation=None,)(conv2)
    conv2 = BatchNormalization(momentum=0.9)(conv2)
    conv2 = Activation(activation='relu')(conv2)
    conv3 = Conv2DTranspose(int(depth/8),
        kernel_size=5, padding='same',
        activation=None,)(conv2)
    conv3 = BatchNormalization(momentum=0.9)(conv3)
    conv3 = Activation(activation='relu')(conv3)
    # Output layer
    image = Conv2D(1, kernel_size=5, padding='same',
        activation='sigmoid')(conv3)
    # Model definition
    model = Model(inputs=noise, outputs=image)
    return model
```

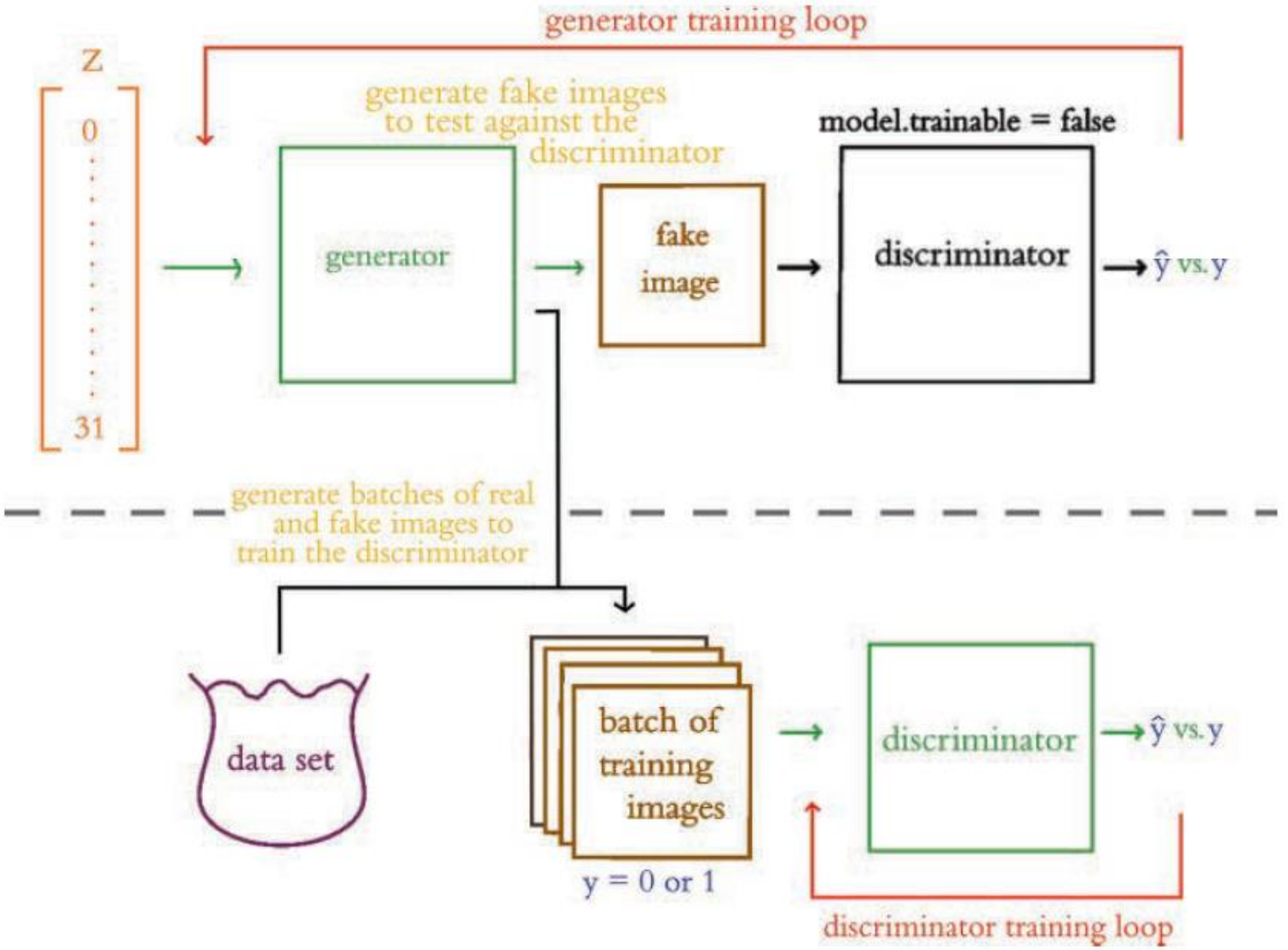
The Generator Network



UpSampling2D and Conv2DTranspose

- UpSampling2D(): each pixel is copied to a 2x2 block of pixels
- Example Conv2DTranspose(filters, kernel_size, padding)
  - Consider a pixel in the middle of an image during convolution
    - First multiplied by the weight in the last row and the last column of the filter
    - Next multiplied by the weight in the last row and the second to last column of the filter
    - Next multiplied by the weight in the last row and the first column of the filter
    - …
    - Finally multiplied by the weight in the first row athe first column of the filter
  - Conv2DTranspose() is effectively convolution with …
    - conv_filter = numpy.rot90(transpose_filter, k = 2)
    - extra padding added for “valid” padding: because this operation reverses “downsizing” due to “valid” padding (the input and output sizes have been exchanged/transposed)

The Two Training Loops



The Two Networks

```
discriminator = build_discriminator()
discriminator.compile(loss='binary_crossentropy',
    optimizer=RMSprop(lr=0.0008,
    decay=6e-8,
    clipvalue=1.0),
    metrics=['accuracy'])

z = Input(shape=(z_dimensions,))
img = generator(z)
discriminator.trainable = False
pred = discriminator(img)
adversarial_model = Model(z, pred)
adversarial_model.compile(loss='binary_crossentropy',
    optimizer=RMSprop(lr=0.0004, decay=3e-8, clipvalue=1.0),
    metrics=['accuracy'])
```

Train GAN: Part 1

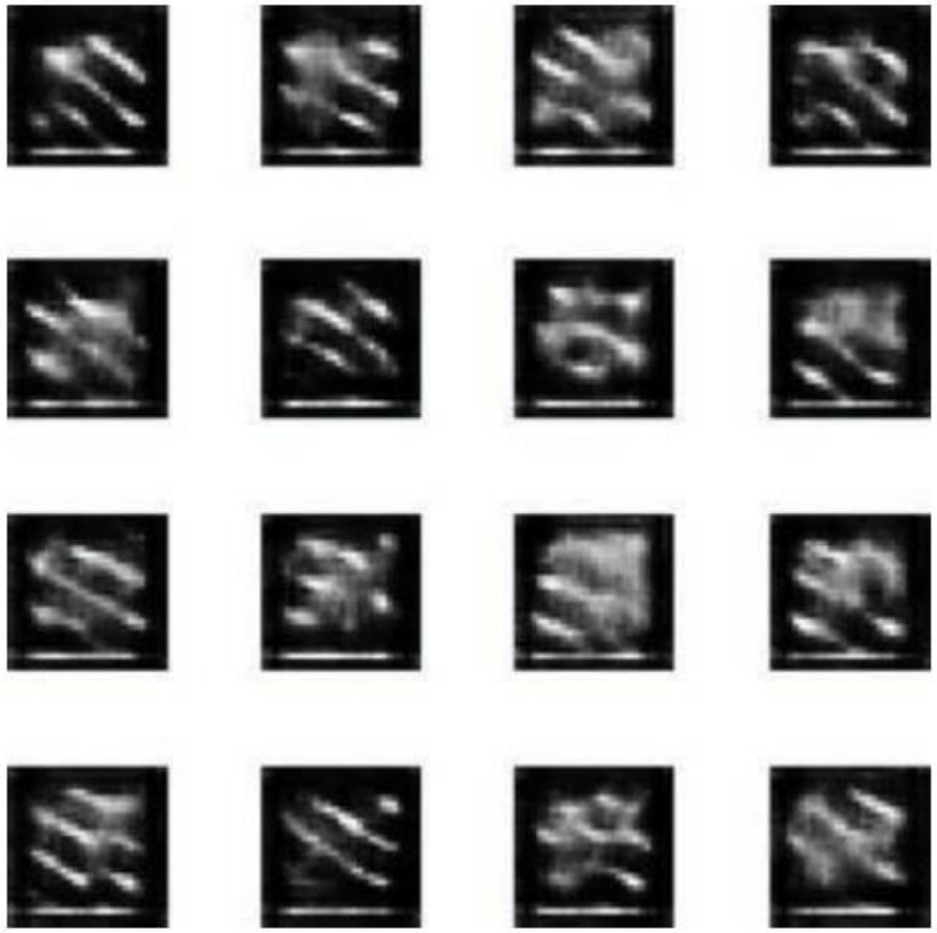
```
def train(epochs=2000, batch=128, z_dim=z_dimensions):
    d_metrics = []
    a_metrics = []
    running_d_loss = 0
    running_d_acc = 0
    running_a_loss = 0
    running_a_acc = 0
    for i in range(epochs):
        # sample real images:
        real_imgs = np.reshape(
            data[np.random.choice(data.shape[0],
            batch,
            replace=False)],
            (batch,28,28,1))
        # generate fake images:
        fake_imgs = generator.predict(
        np.random.uniform(-1.0, 1.0,
        size=[batch, z_dim]))
```

```
        # concatenate images as discriminator inputs:
        x = np.concatenate((real_imgs,fake_imgs))
        # adversarial net's noise input and "real" y:
        noise = np.random.uniform(-1.0, 1.0,
            size=[batch, z_dim])
        y = np.ones([batch,1])
        # train adversarial net:
        a_metrics.append(
            adversarial_model.train_on_batch(noise,y)
        )
        running_a_loss += a_metrics[-1][0]
        running_a_acc += a_metrics[-1][1]
        # assign y labels for discriminator:
        y = np.ones([2*batch,1])
        y[batch:,:] = 0
        # train discriminator:
        d_metrics.append(
            discriminator.train_on_batch(x,y)
        )
```

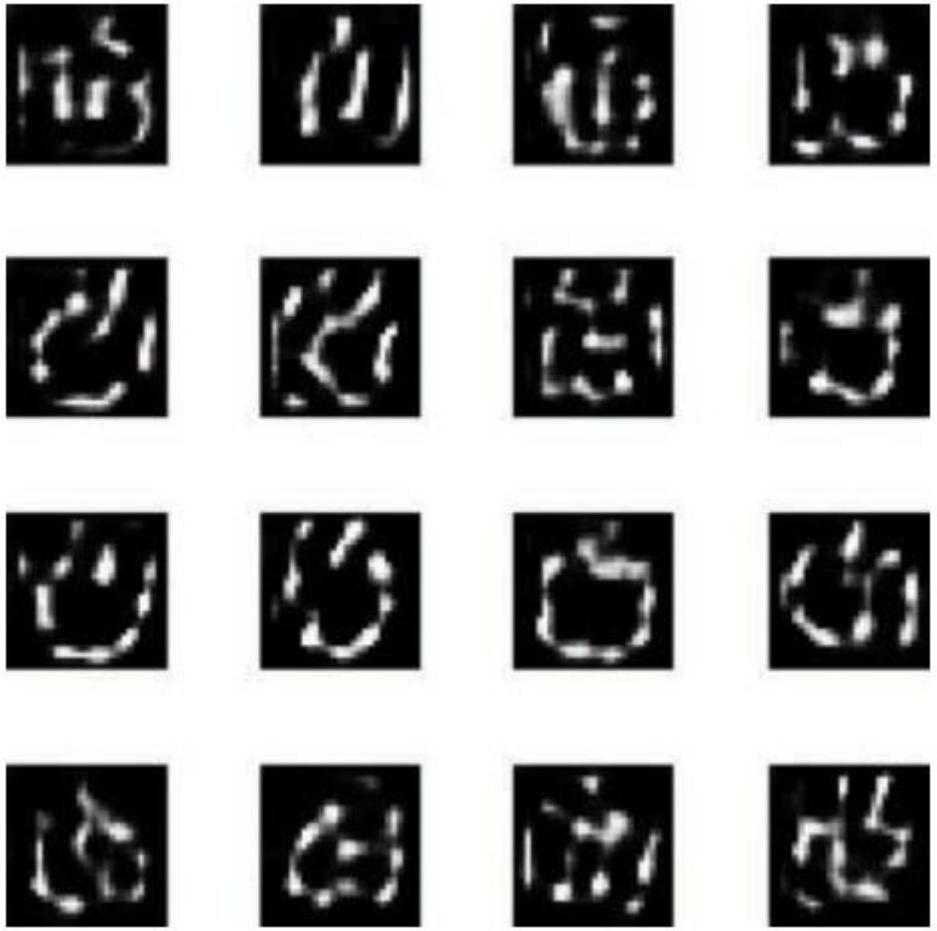
Train GAN: Part 2

```
running_d_loss += d_metrics[-1][0]
running_d_acc += d_metrics[-1][1]
# adversarial net's noise input and "real" y:
noise = np.random.uniform(-1.0, 1.0,
   size=[batch, z_dim])
y = np.ones([batch,1])
# train adversarial net:
a_metrics.append(
   adversarial_model.train_on_batch(noise,y)
)
running_a_loss += a_metrics[-1][0]
running_a_acc += a_metrics[-1][1]
# periodically print progress & fake images:
if (i+1)%100 == 0:
   print('Epoch #{}'.format(i))
   log_mesg = "%d: [D loss: %f, acc: %f]" % \
      (i, running_d_loss/i, running_d_acc/i)
   log_mesg = "%s [A loss: %f, acc: %f]" % \
      (log_mesg, running_a_loss/i, running_a_acc/i)
   print(log_mesg)
   noise = np.random.uniform(-1.0, 1.0,
      size=[16, z_dim])
   gen_imgs = generator.predict(noise)
   plt.figure(figsize=(5,5))
   for k in range(gen_imgs.shape[0]):
      plt.subplot(4, 4, k+1)
      plt.imshow(gen_imgs[k, :, :, 0],
         cmap='gray')
      plt.axis('off')
   plt.tight_layout()
   plt.show()
  return a_metrics, d_metrics
# train the GAN:
a_metrics_complete, d_metrics_complete = train()
```

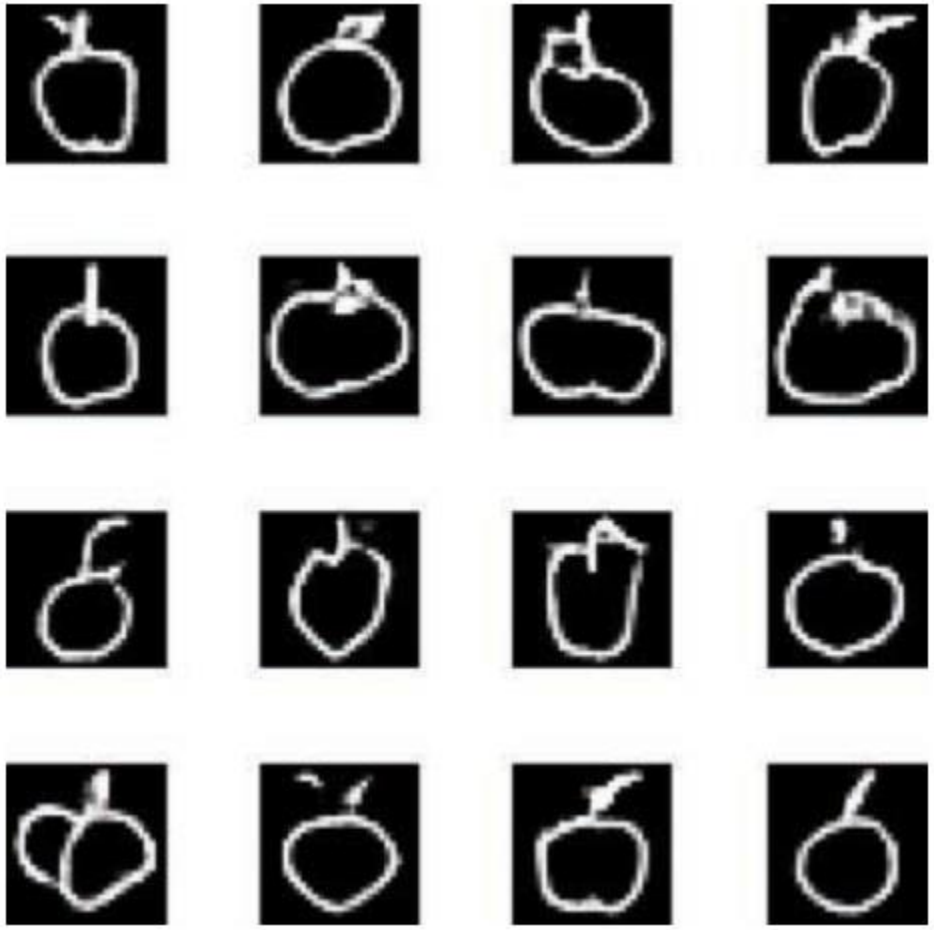
Fake Sketches After 100 Epochs



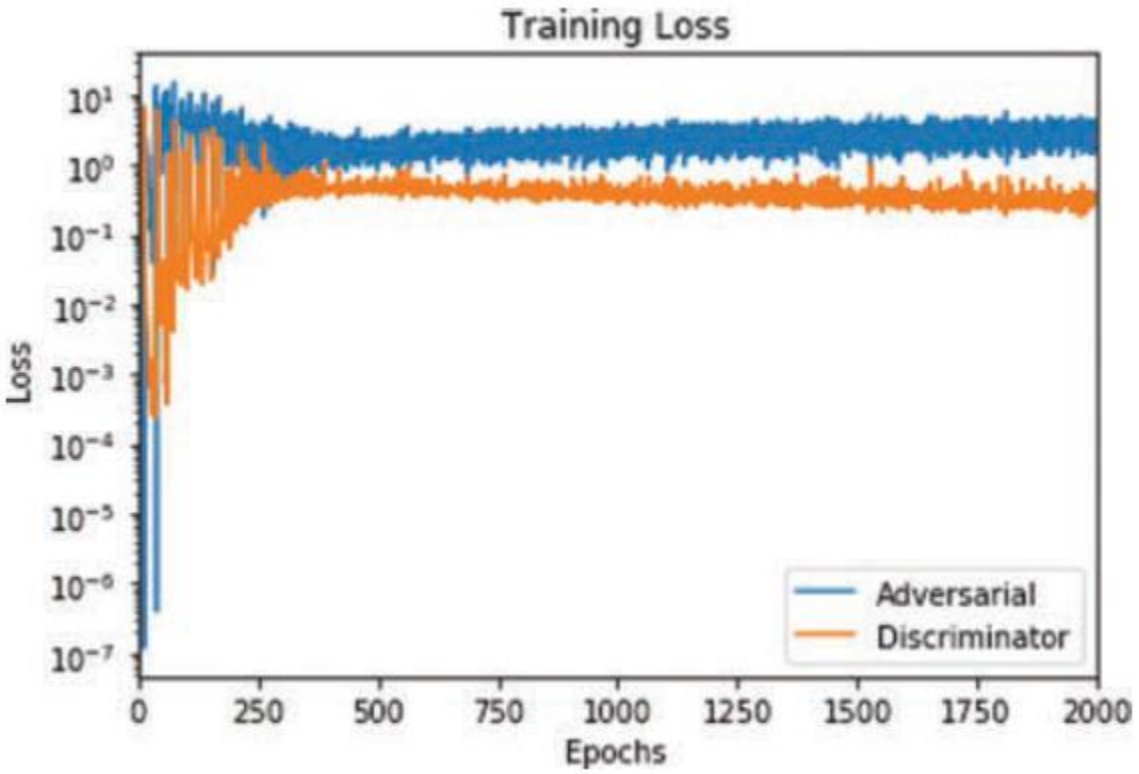
Fake Sketches After 200 Epochs

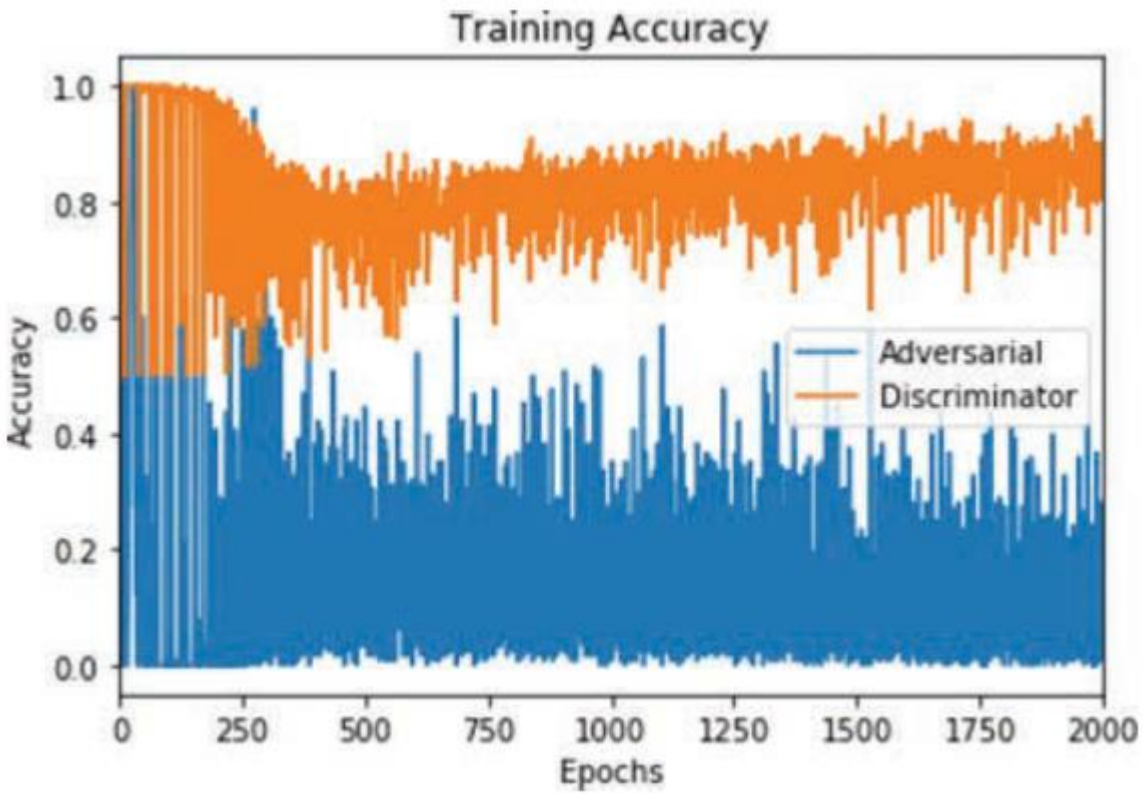


Fake Sketches After 1,000 Epochs



Training Loss and Accuracy





Summary

In this chapter, we covered the essential theory of GANs, including a couple of new layer types (de-convolution and upsampling). We constructed discriminator and generator networks and then combined them to form an adversarial network. Through alternately training a discriminator model and the generator component of the adversarial model, the GAN learned how to create novel “sketches” of apples.

https://www.ling.upenn.edu/courses/Fall_2003/ling001/penn_treebank_pos.html

Example Article: wsj_0001.mrg

```
( (S
   (NP-SBJ
    (NP (NNP Pierre) (NNP Vinken) )
    (, ,)
    (ADJP
     (NP (CD 61) (NNS years) )
     (JJ old) )
    (, ,) )
   (VP (MD will)
    (VP (VB join)
     (NP (DT the) (NN board) )
     (PP-CLR (IN as)
      (NP (DT a) (JJ nonexecutive) (NN director) ))
     (NP-TMP (NNP Nov.) (CD 29) )))
   (. .) ))
```

```
( (S
   (NP-SBJ (NNP Mr.) (NNP Vinken) )
   (VP (VBZ is)
    (NP-PRD
     (NP (NN chairman) )
     (PP (IN of)
      (NP
       (NP (NNP Elsevier) (NNP N.V.) )
       (, ,)
       (NP (DT the) (NNP Dutch) (VBG publishing) (NN group)
)))))
   (. .) ))
```

Pierre Vinken, 61 years old, will join the board as a nonexecutive director Nov. 29.
Mr. Vinken is chairman of Elsevier N.V., the Dutch publishing group.