Generative Models

December 1, 2022

ddebarr@uw.edu

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

Agenda

- Homework Review
- [DLI] Generative Adversarial Networks
- [DLP] Generative Deep Learning

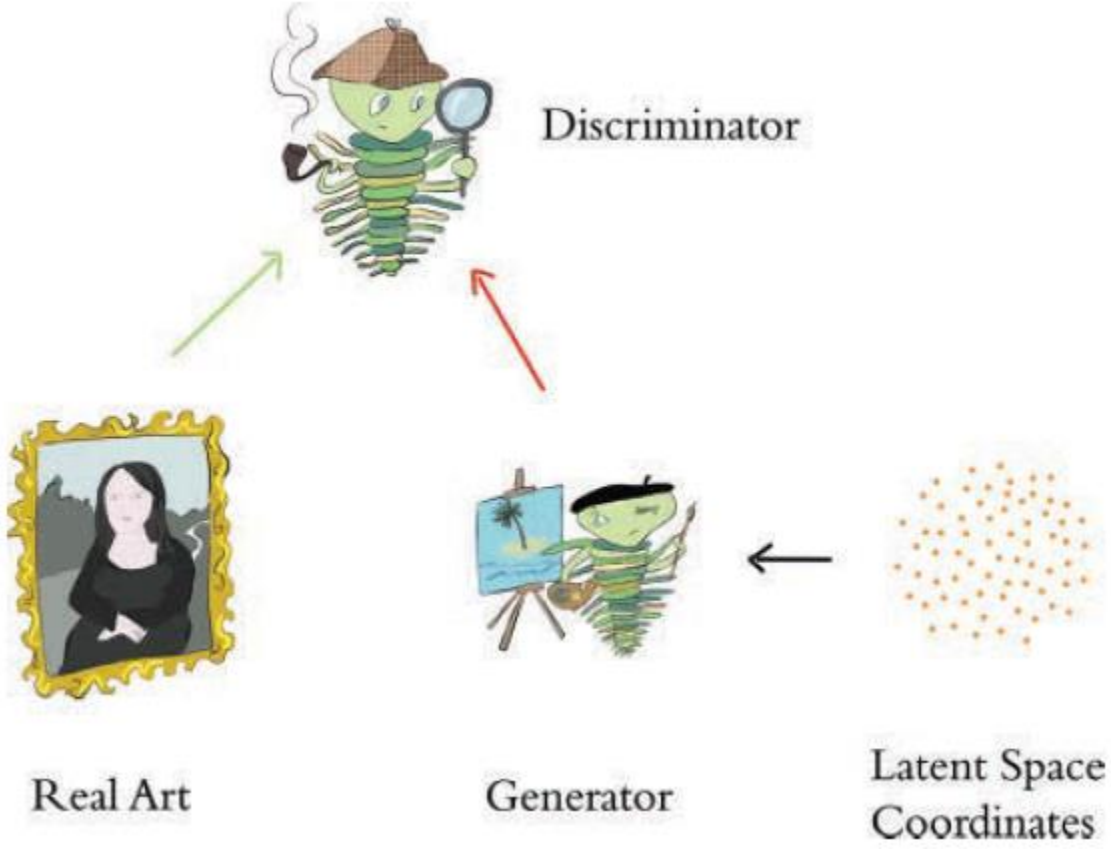
[DLI] Machine Art

- 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)

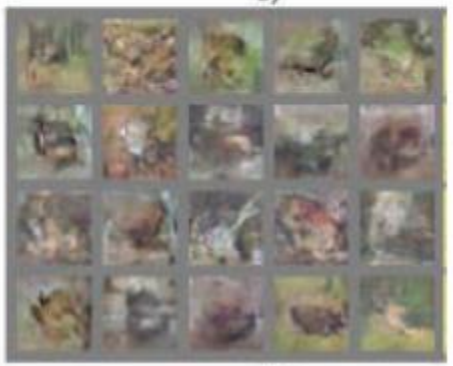
a)

b)

Toronto Face Database (TFD)

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

c)

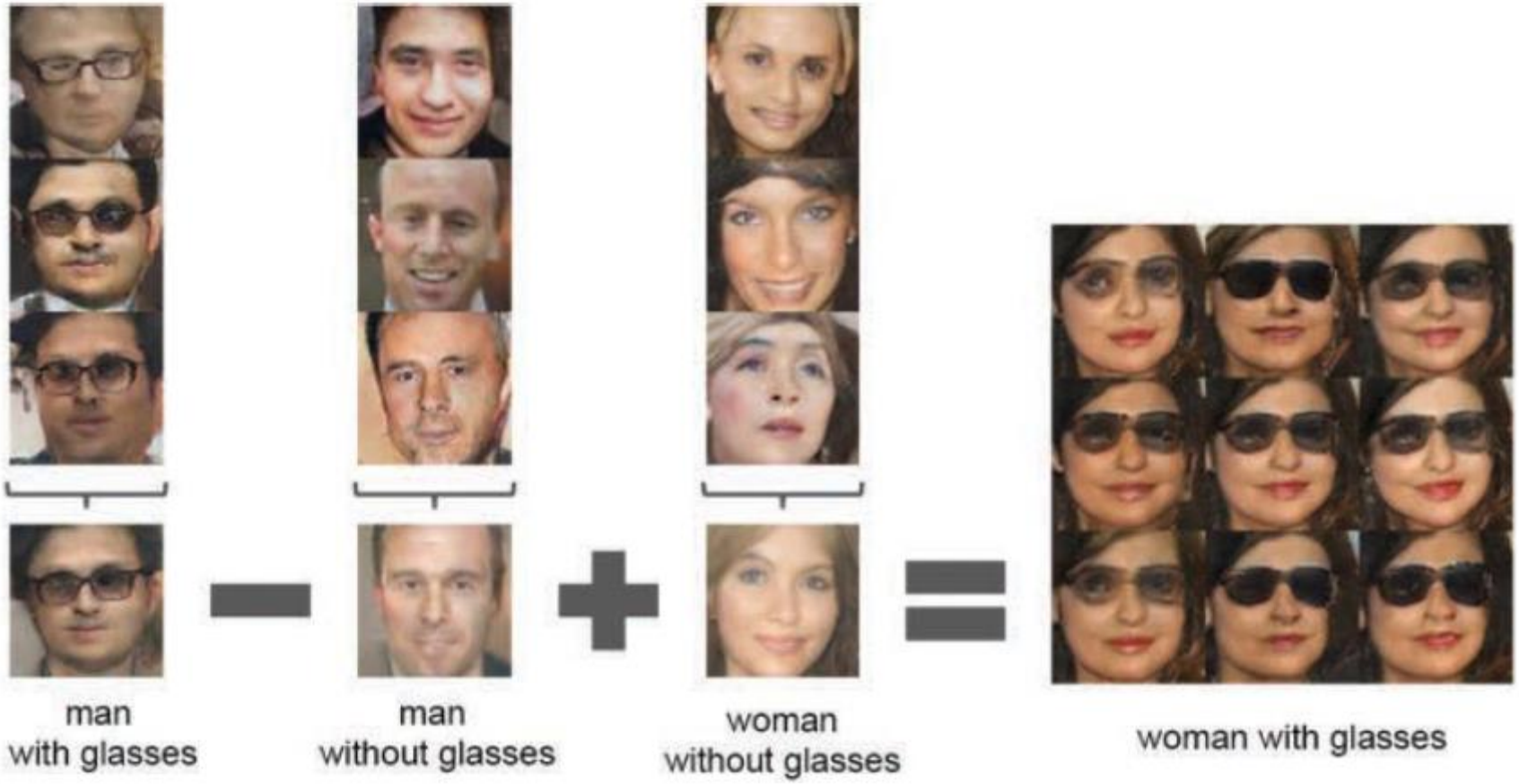


d)

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

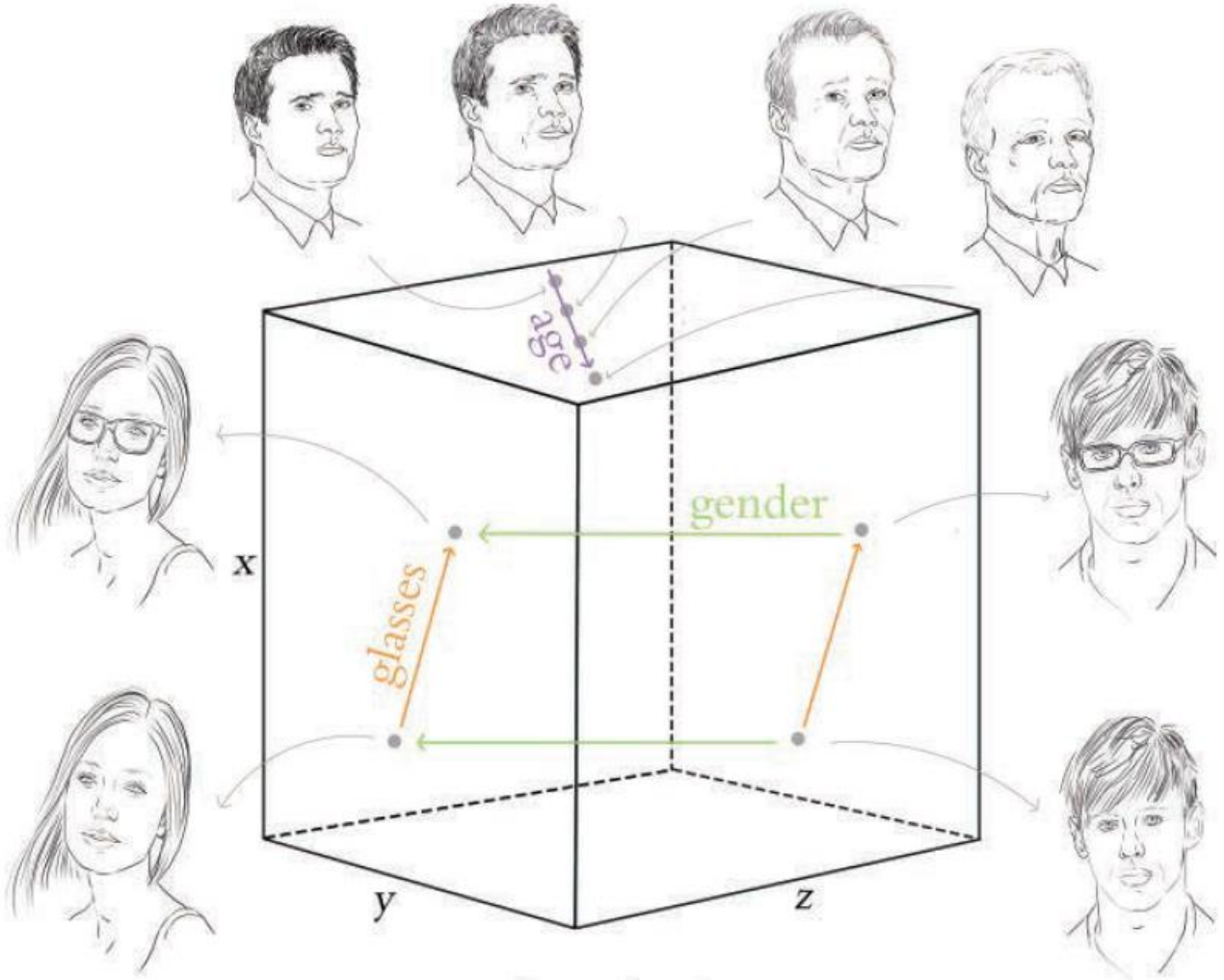
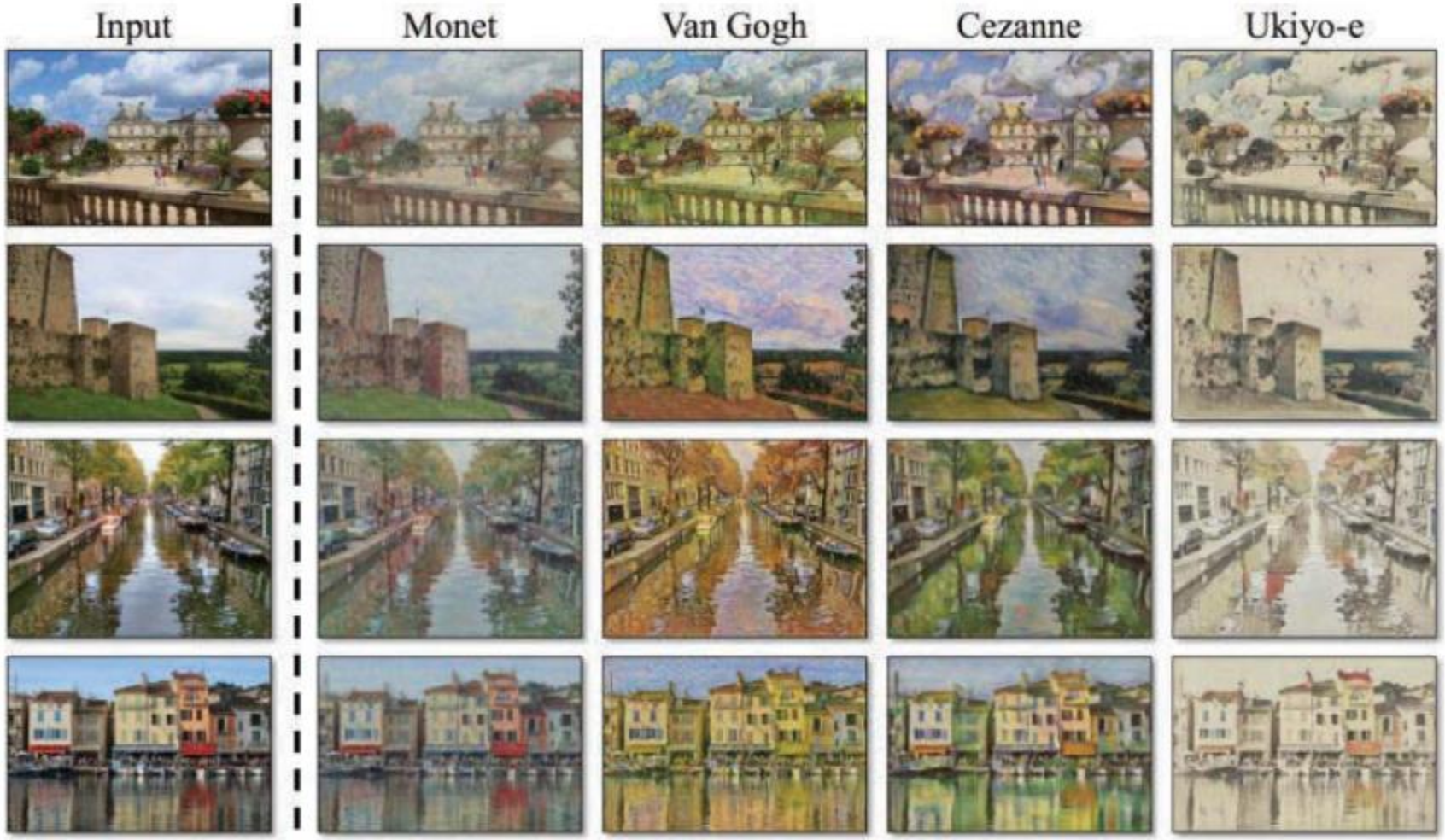
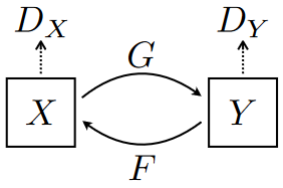


Photo + CycleGAN Image to Generate Output



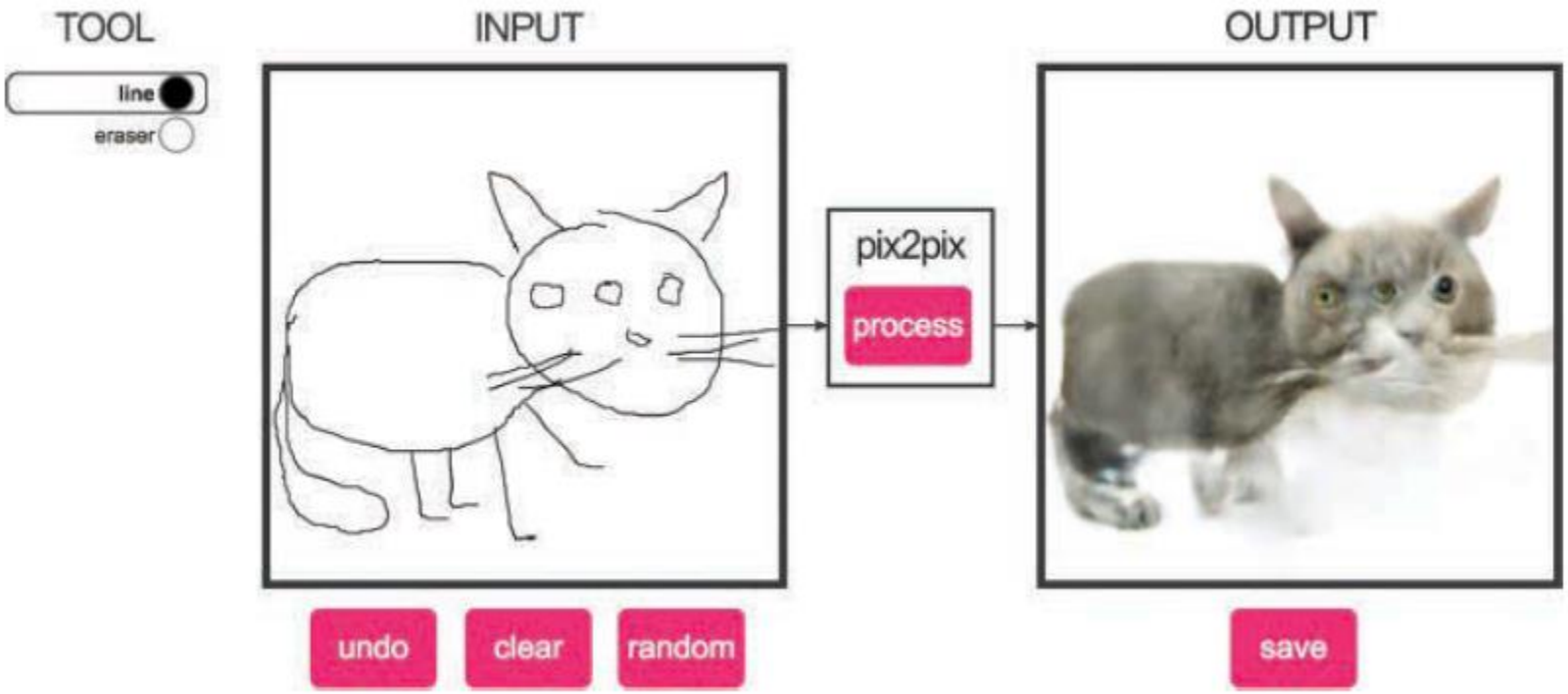
mo nay
van go
say zahn
you key oh ay



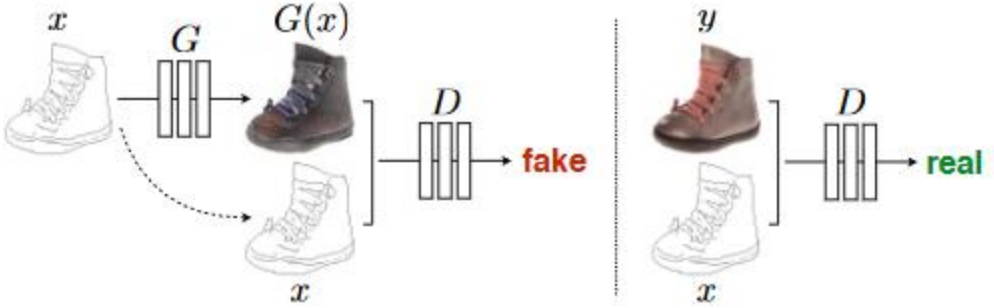
https://arxiv.org/pdf/1703.10593.pdf

Learning to automatically "translate" from one image collection to another

Pix2Pix Demo



Training a conditional GAN to map edges to a photo …

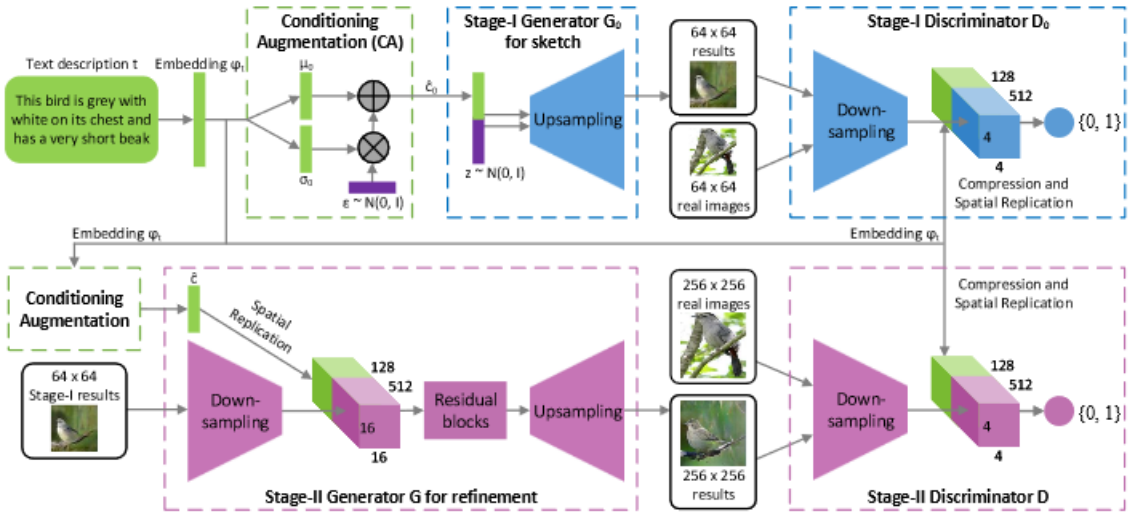


https://arxiv.org/pdf/1611.07004.pdf

StackGAN Examples

First GAN:
text to 64x64 image

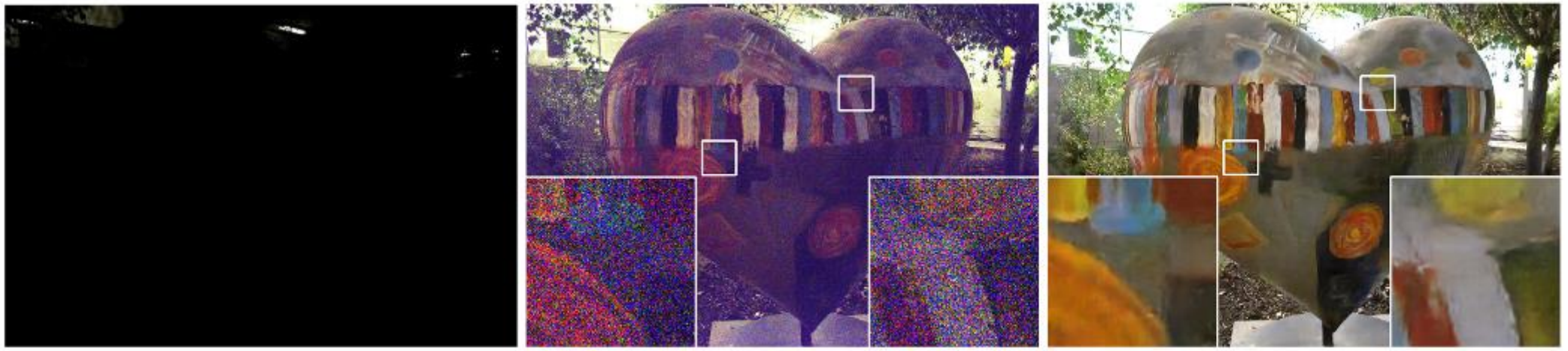
Second GAN:
64x64 image to 256x256 image



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

Image Processing Example

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

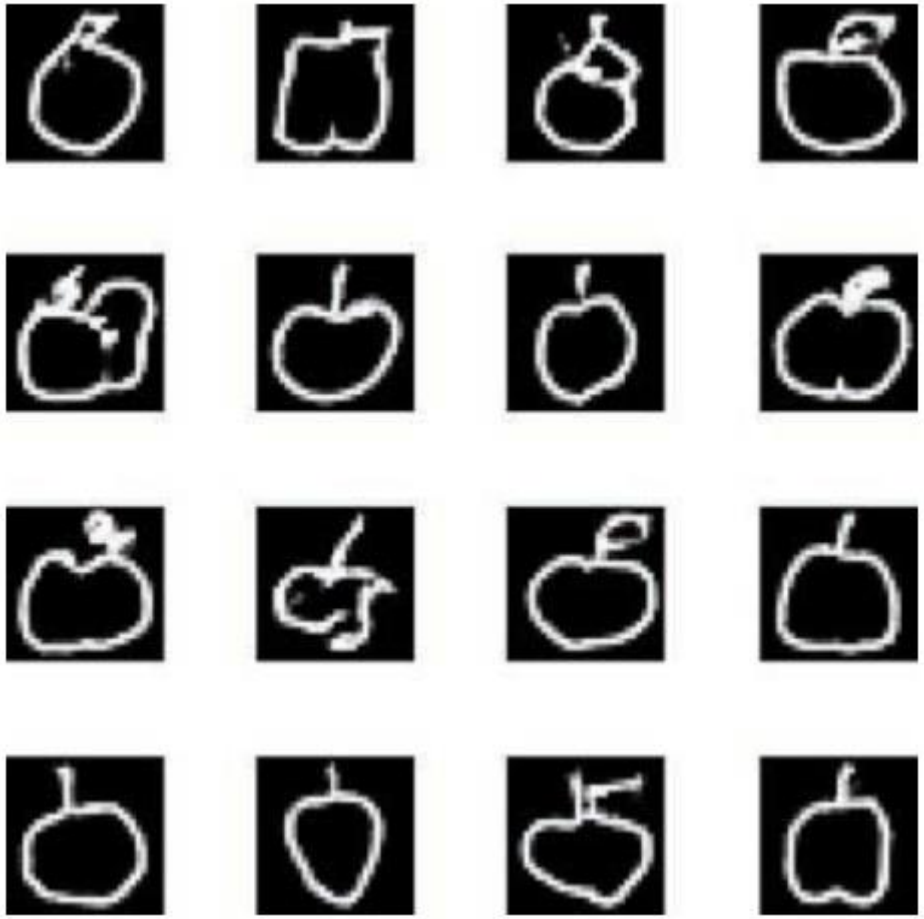


Learning to See in the Dark: trained a fully convolutional network to perform the entire image processing pipeline

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

“Hand” Drawings of Apples

GAN trained on “Quick, Draw!” images

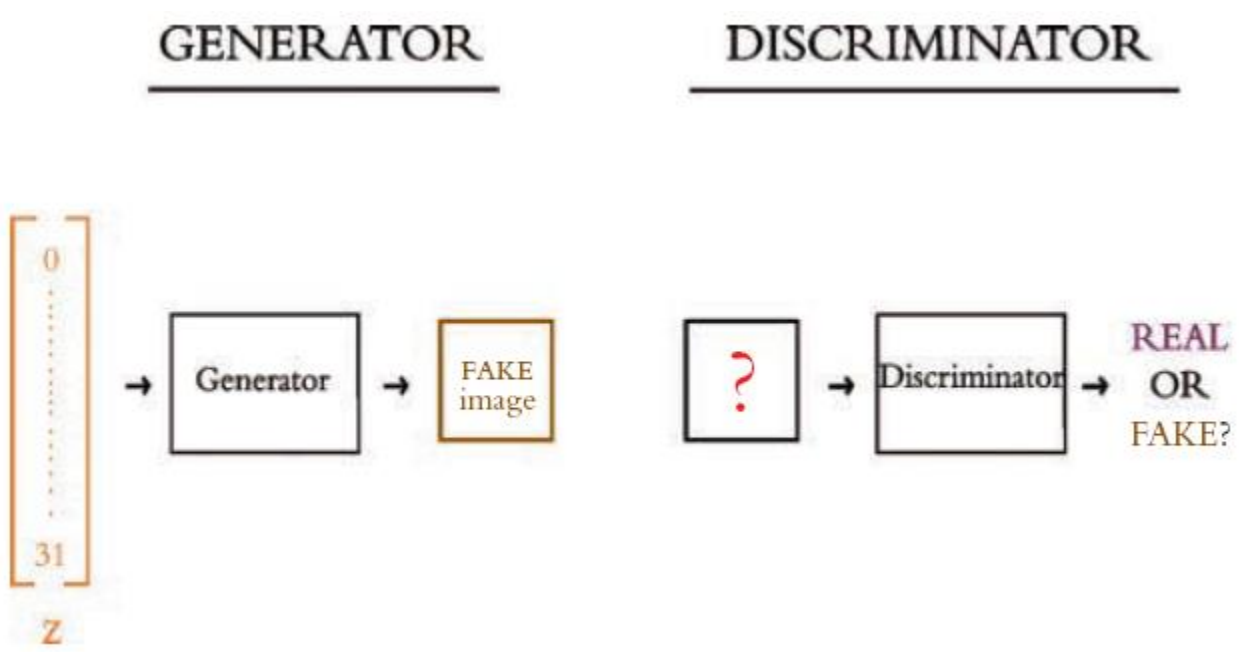


[DLI] Generative Adversarial Networks

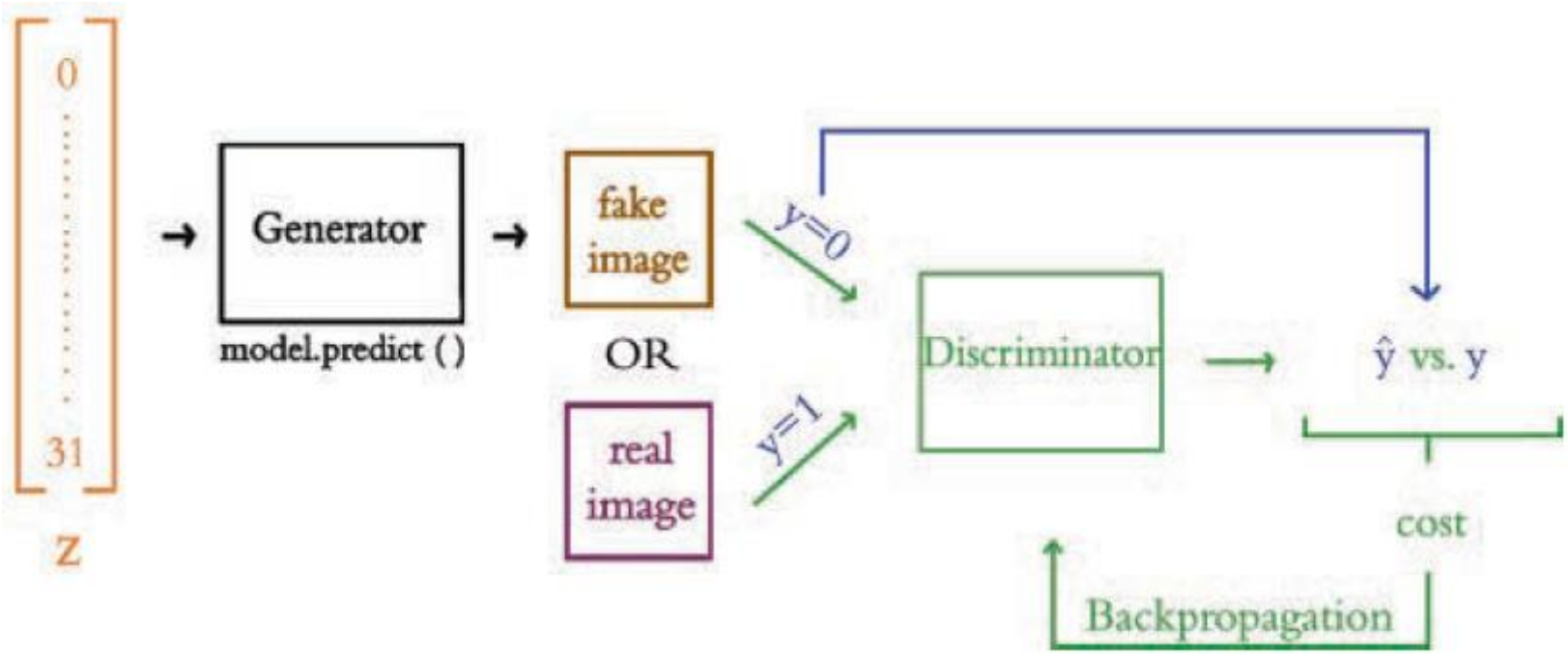
- Essential GAN Theory
- The Quick, Draw! Dataset
- The Discriminator Network
- The Generator Network
- The Adversarial Network
- GAN Training
- Summary

Generator versus Discriminator

The two models that make up a GAN

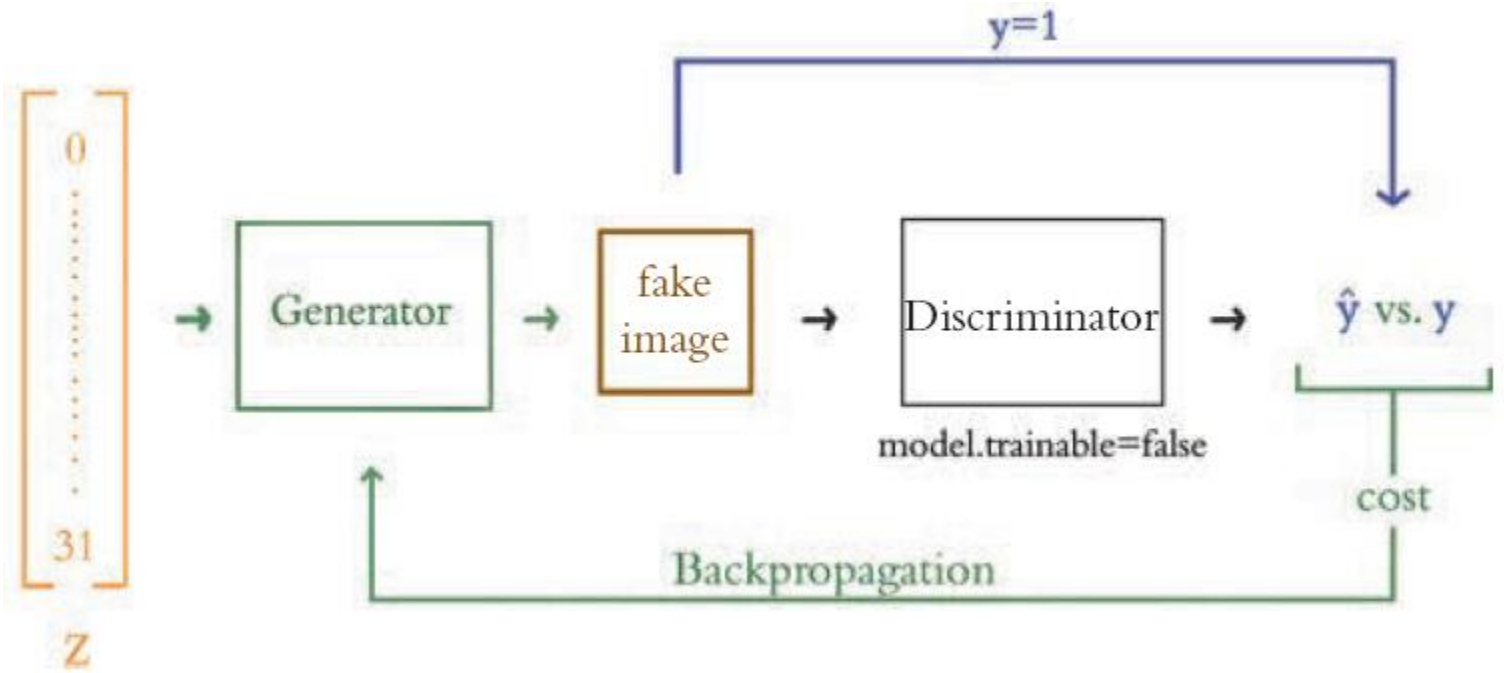


Training the Discriminator



Only the weights of the Discriminator are updated

Training the Generator



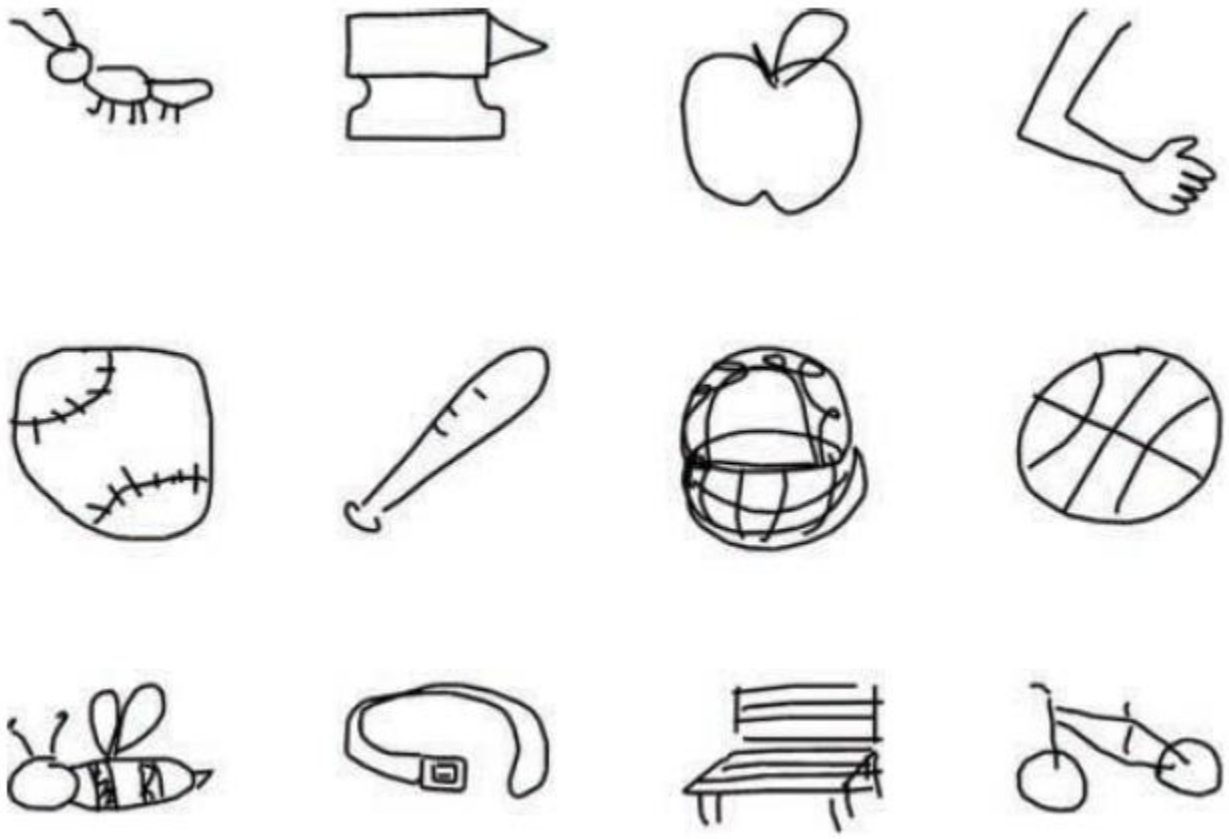
Error is backpropagated across the Discriminator, but only the weights of the Generator are updated

Training

while not done:

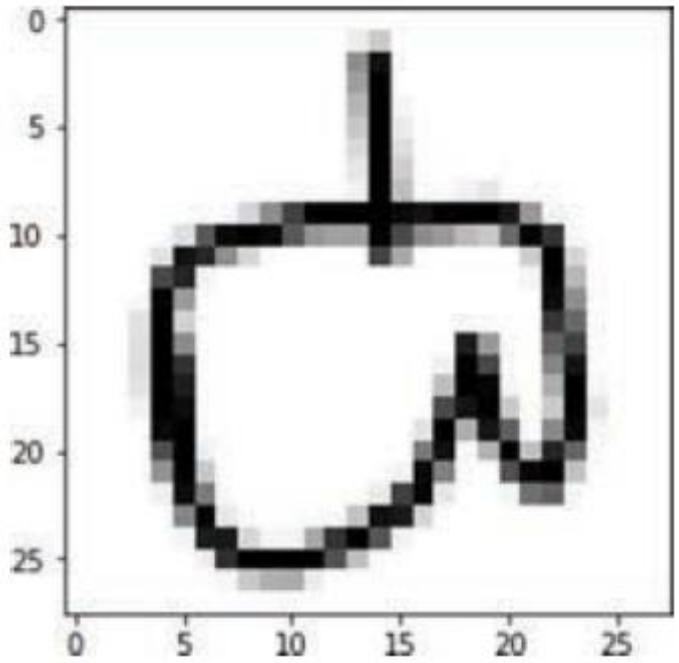
- train the discriminator
  - minimize log loss for real/fake labels
  - only update discriminator weights
- train the generator
  - minimize log loss for “real” labels
  - backprop across both models
  - but only update generator weights

Sketches Drawn by Humans



http://bit.ly/QDdata

Bitmap Example



Shape: (28, 28, 1)

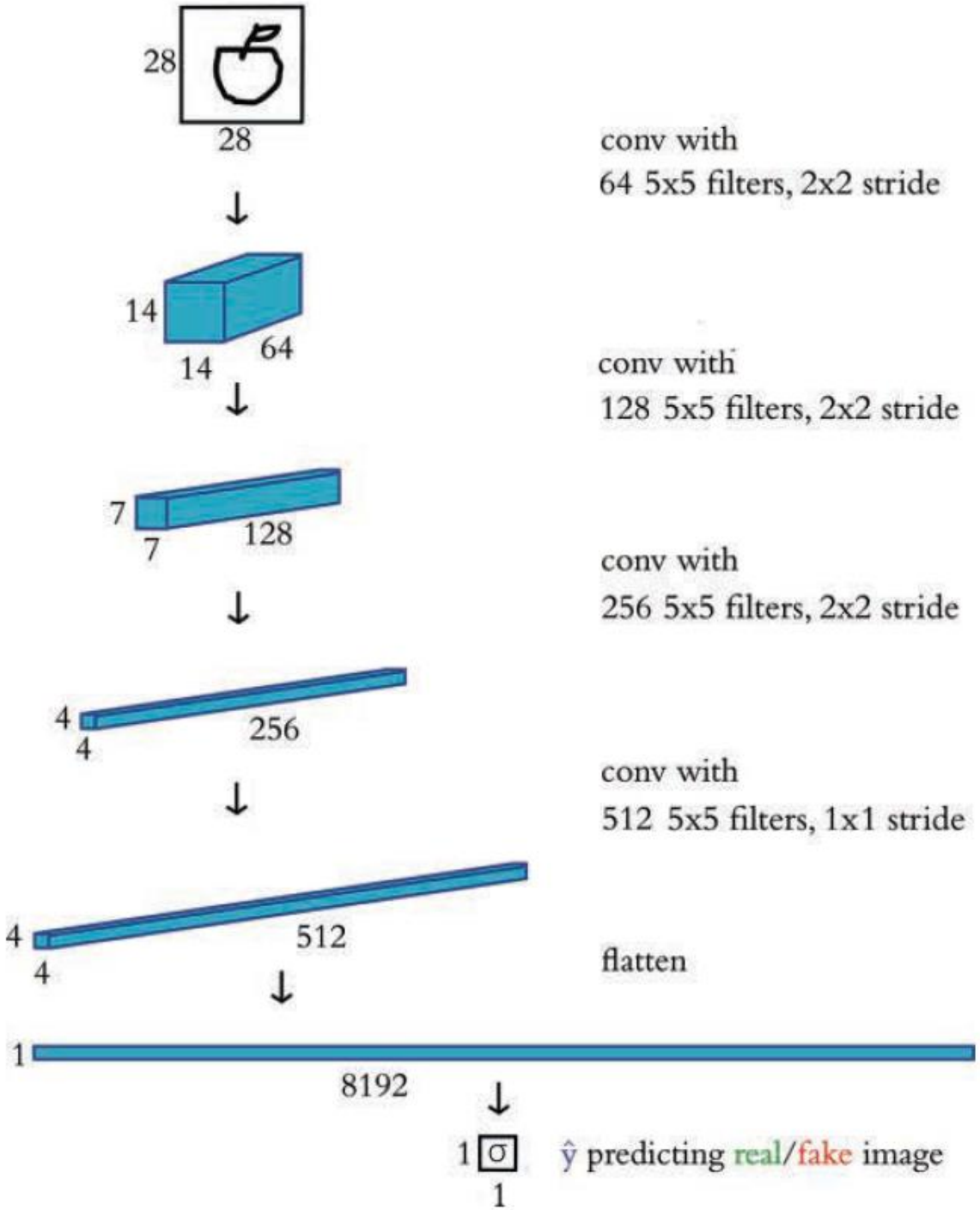
Quick, Draw! GAN Dependencies

```
# 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
```

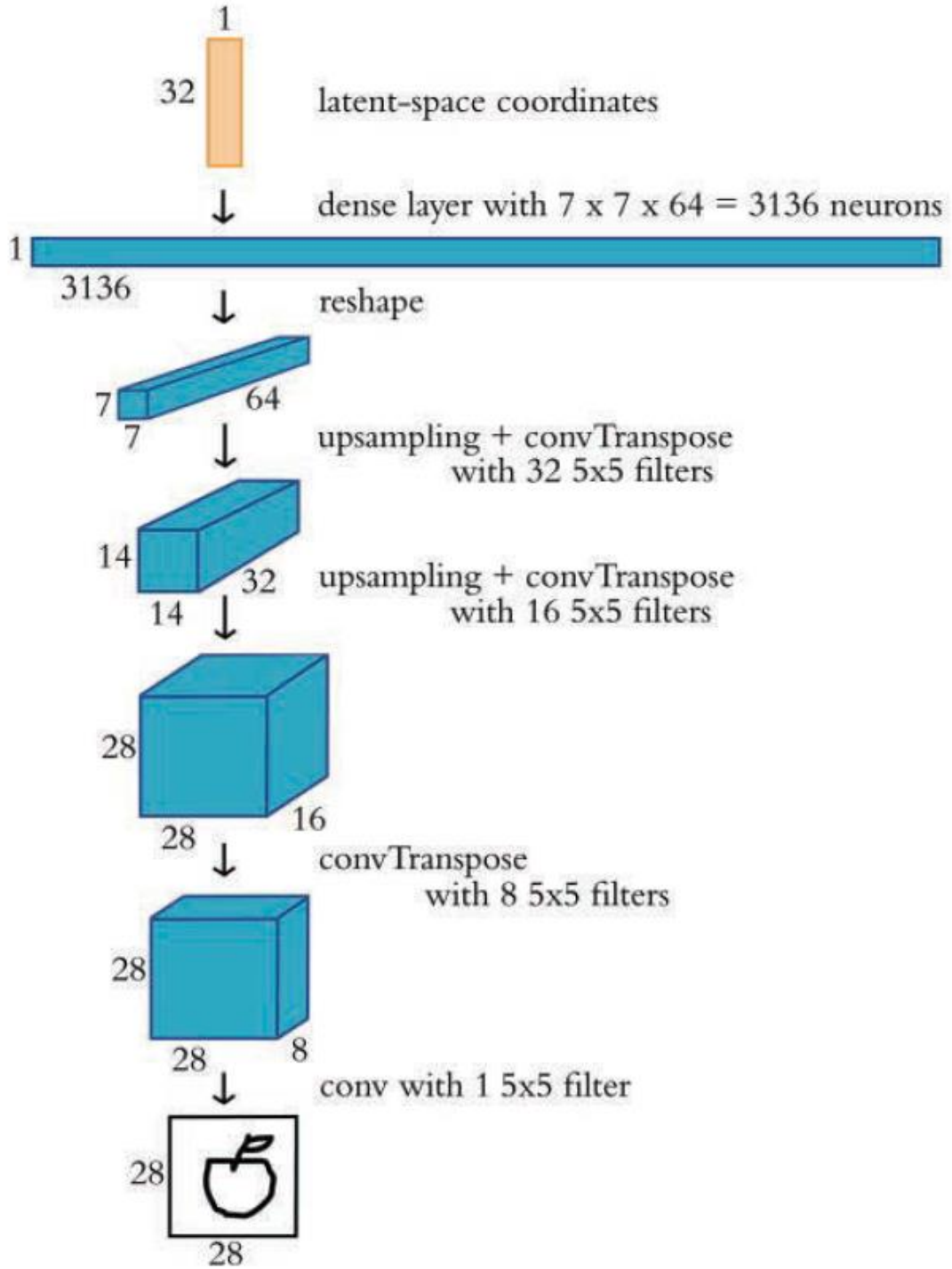
The Discriminator Network



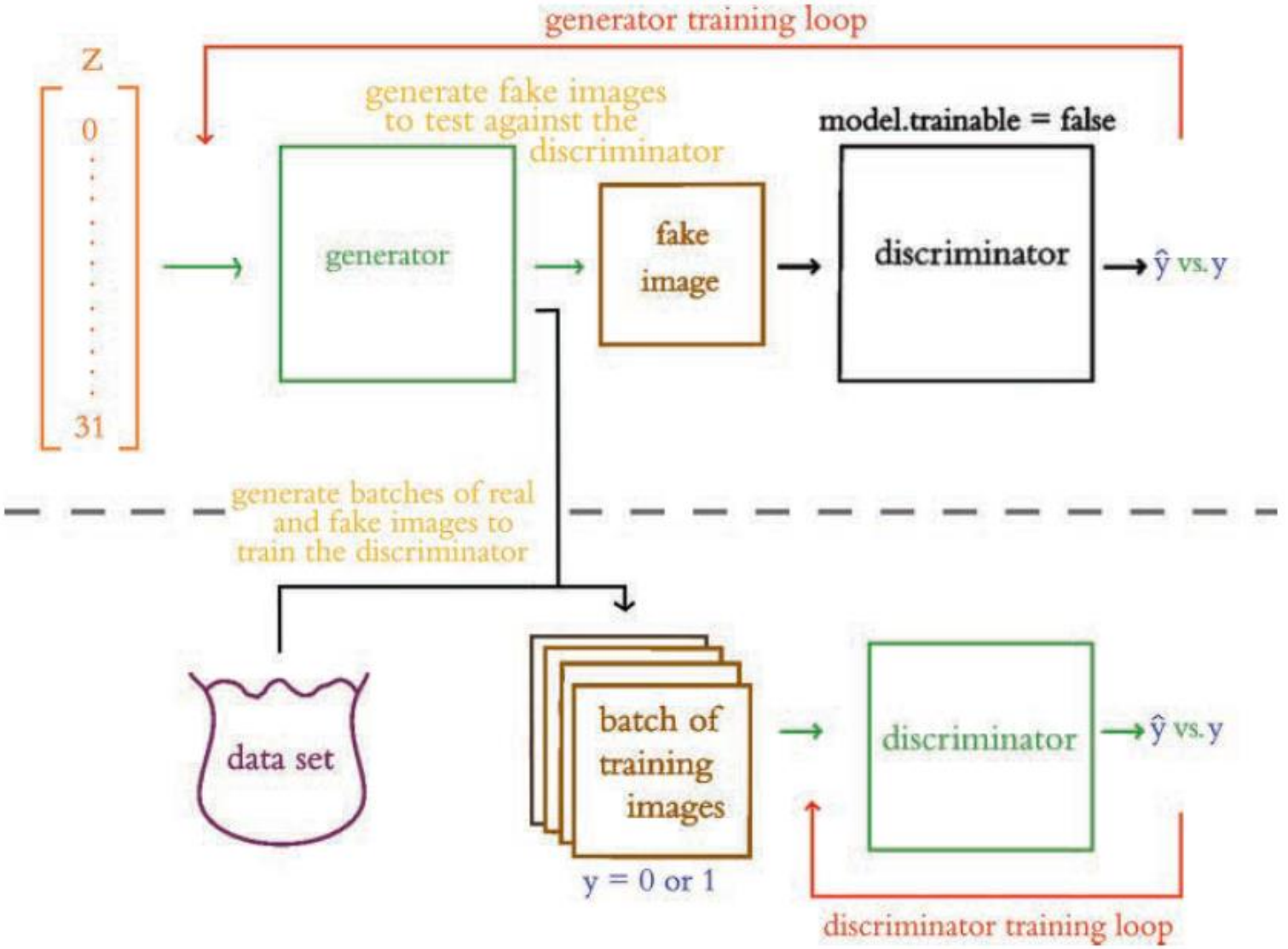
The Generator

```
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



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

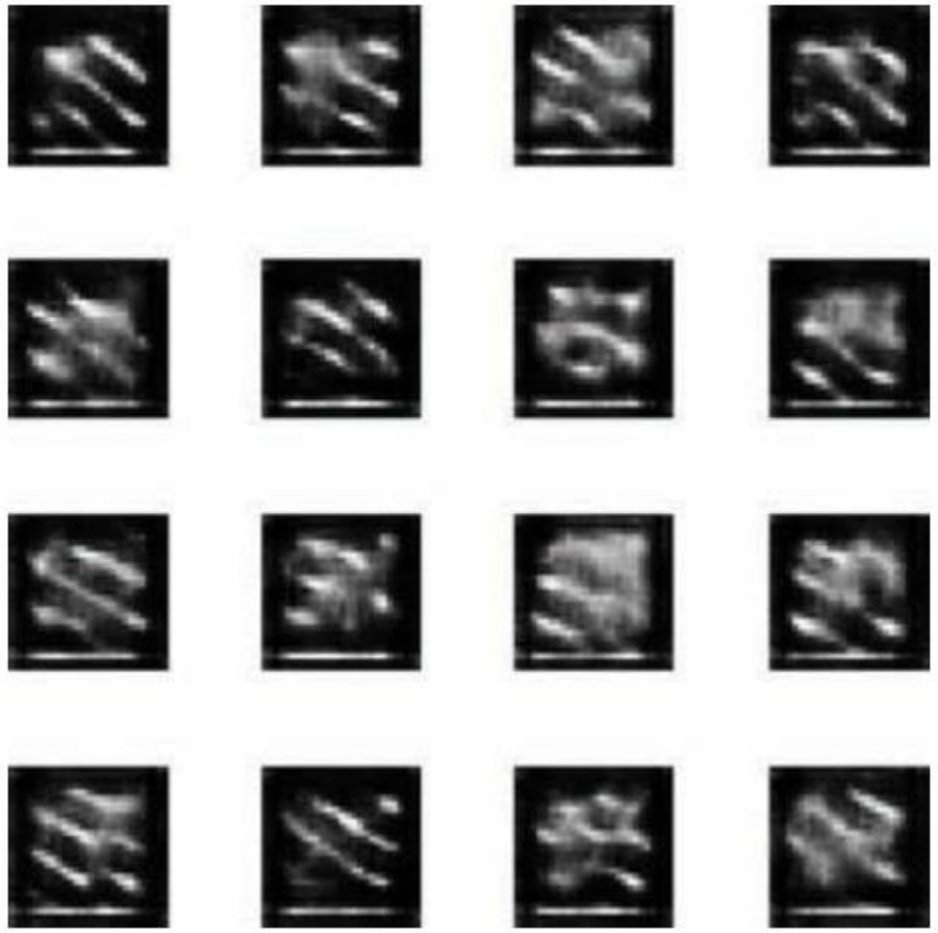
```
def train(epochs=2000, batch=128, z_dim=z_dimensions):
    d_metrics = []
    a_metrics = []
    running_d_loss = 0
    running_d_acc = 0
    running_a_loss = 0
    running_a_acc = 0
    for i in range(epochs):
        # sample real images:
        real_imgs = np.reshape(
            data[np.random.choice(data.shape[0],
            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)
        )
```

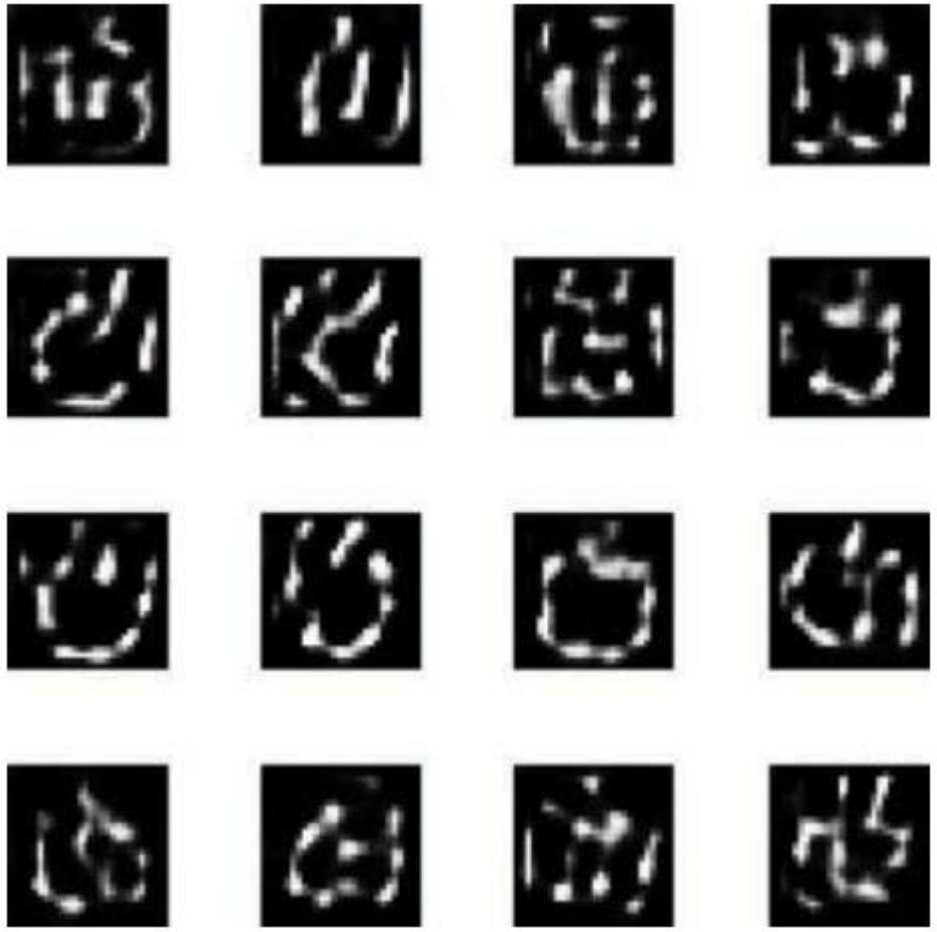
Train GAN: Part 2

```
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()
```

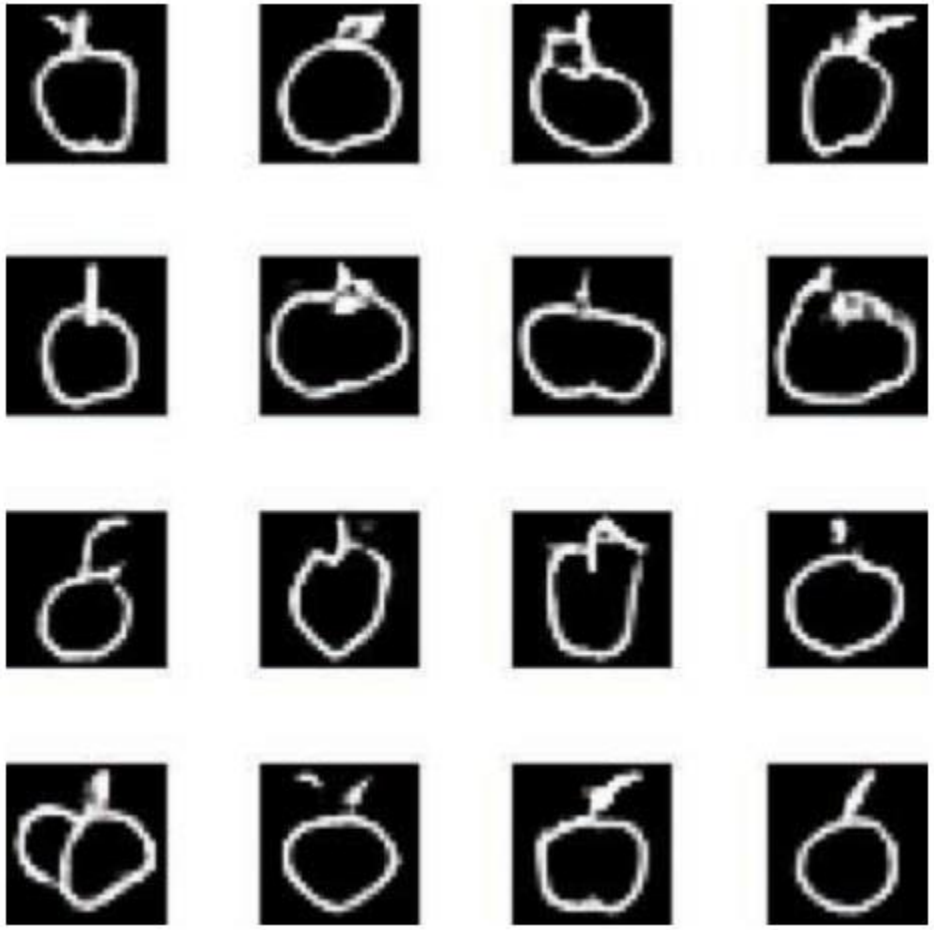
Fake Sketches After 100 Epochs



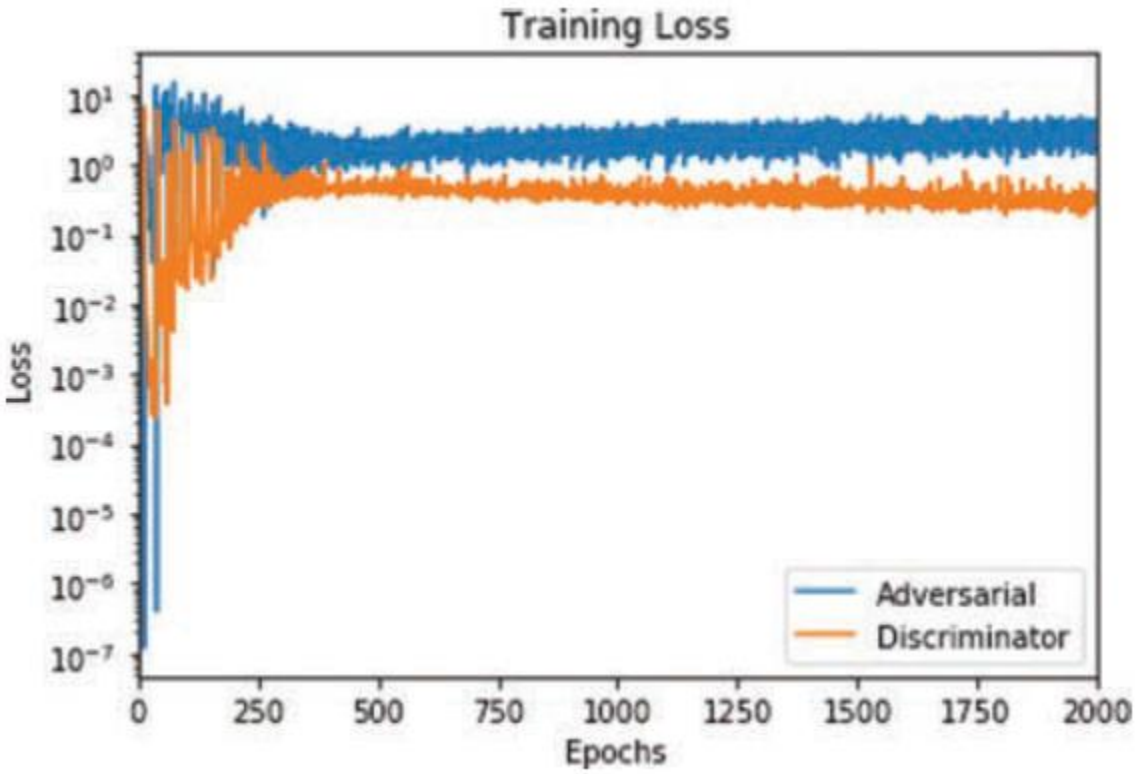
Fake Sketches After 200 Epochs

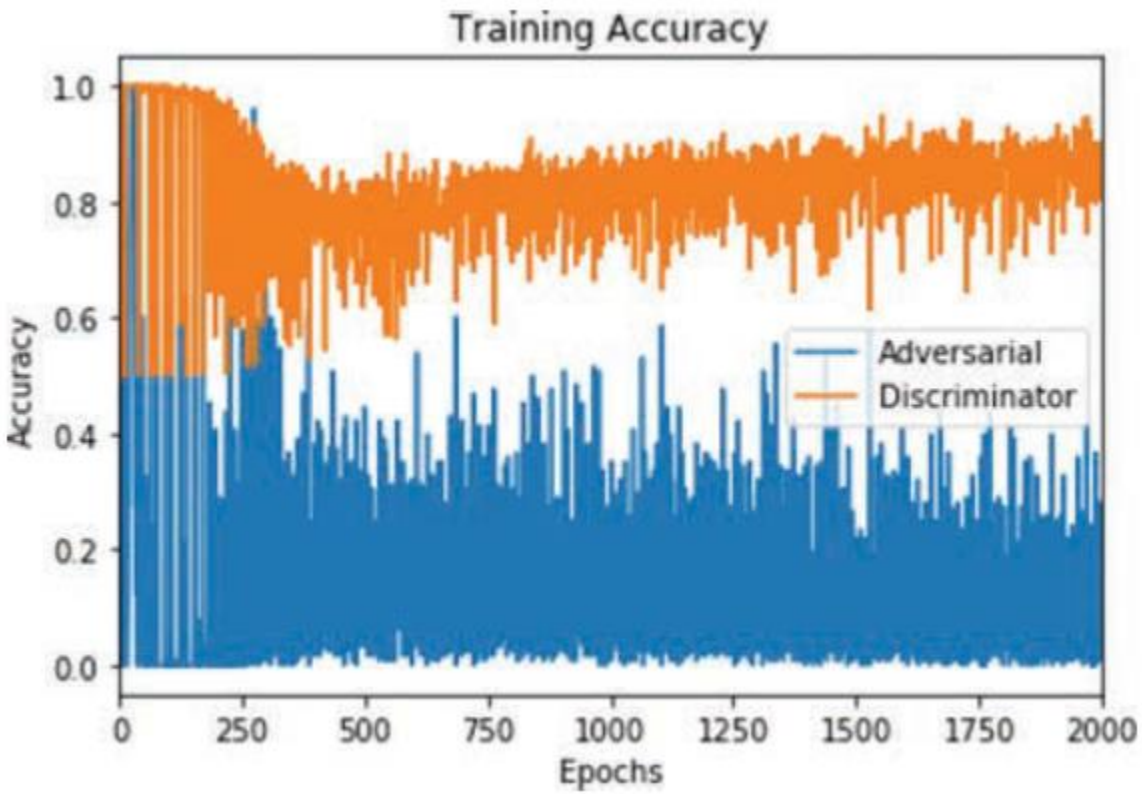


Fake Sketches After 1,000 Epochs



Training Loss and Accuracy



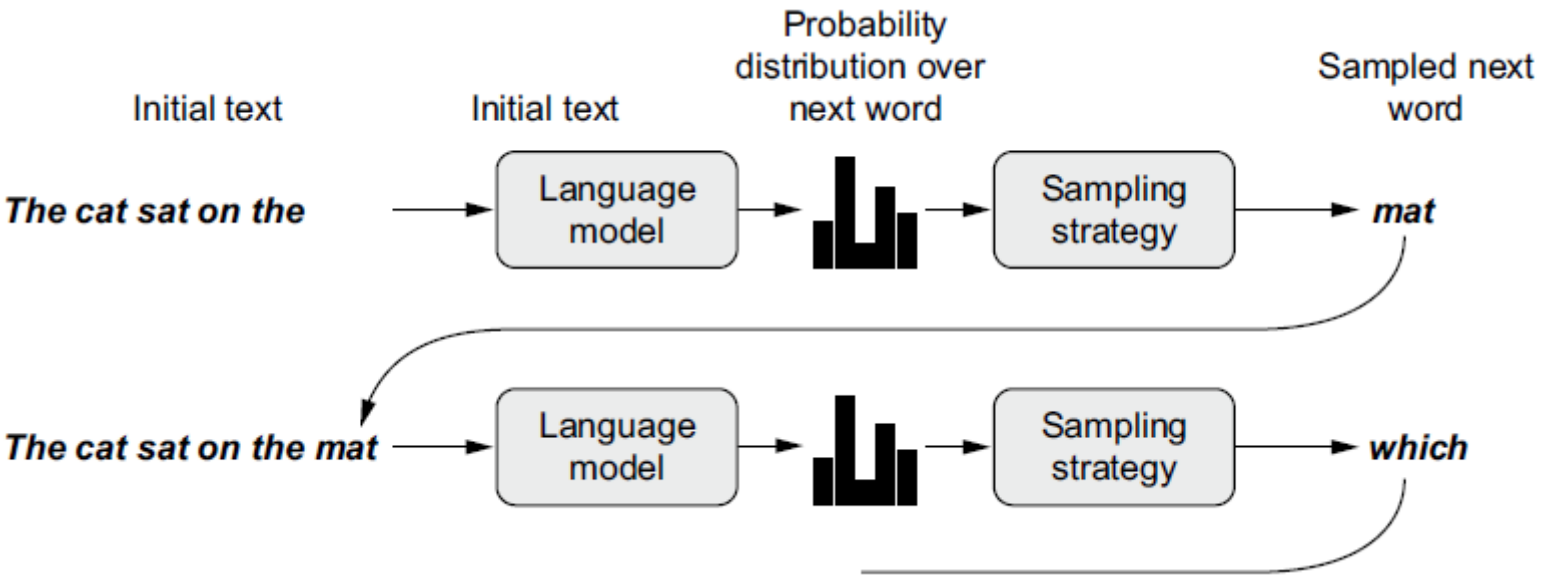


Summary

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.

Generative Deep Learning

Word-by-Word Text Generation



Stochastic Sampling with Temperature

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.

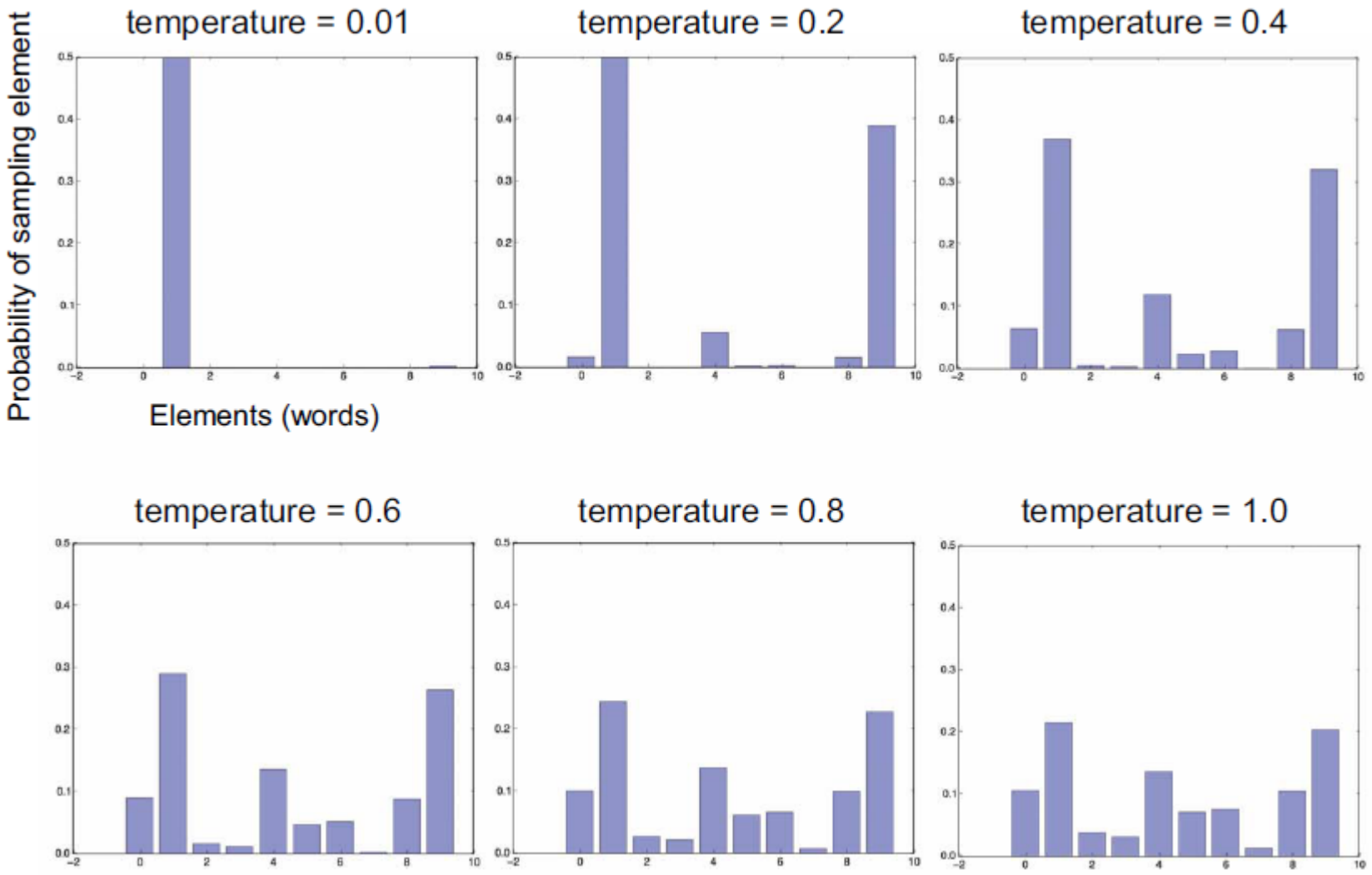
```
import numpy as np
def reweight_distribution(original_distribution, temperature=0.5):
    distribution = np.log(original_distribution) / temperature
    distribution = np.exp(distribution)
    return distribution / np.sum(distribution)
```

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.

Stochastic sampling sometimes called softmax sampling, Monte Carlo sampling, Boltzmann sampling.
Contributions have come from many different fields, which leads to variety in terminology.
Data science is a team sport: always ask when hearing an unfamiliar term!

Affect of Lowering Temperature

[higher temperature yields higher entropy]



Q: What would happen if temperature = 100?

A: Probabilities would be approximately equal!

Preparing IMDB Data for Language Modeling

```
!wget https://ai.stanford.edu/~amaas/data/sentiment/aclImdb_v1.tar.gz
!tar -xf aclImdb_v1.tar.gz
```

```
import tensorflow as tf
from tensorflow import keras
dataset = keras.utils.text_dataset_from_directory(
    directory="aclImdb", label_mode=None, batch_size=256)
dataset = dataset.map(lambda x: tf.strings.regex_replace(x, "<br />", " "))
```

Strip the
 HTML tag that occurs in many of the reviews. This did not matter much for text classification, but we wouldn't want to generate
 tags in this example!

```
from tensorflow.keras.layers import TextVectorization
sequence_length = 100
vocab_size = 15000
text_vectorization = TextVectorization(
    max_tokens=vocab_size,
    output_mode="int",
    output_sequence_length=sequence_length,
)
text_vectorization.adapt(dataset)
```

We'll only consider the top 15,000 most common words—anything else will be treated as the out-of-vocabulary token, "[UNK]".

We want to return integer word index sequences.

We'll work with inputs and targets of length 100 (but since we'll offset the targets by 1, the model will actually see sequences of length 99)

```
def prepare_lm_dataset(text_batch):
    vectorized_sequences = text_vectorization(text_batch)
    x = vectorized_sequences[:, :-1]
    y = vectorized_sequences[:, 1:]
    return x, y

lm_dataset = dataset.map(prepare_lm_dataset, num_parallel_calls=4)
```

Convert a batch of texts (strings) to a batch of integer sequences.

Create inputs by cutting off the last word of the sequences.

Create targets by offsetting the sequences by 1.

Next-Word vs Sequence-to-Sequence Prediction

Next-word prediction	the cat sat on the → **mat**
Sequence-to-sequence modeling	the → **cat sat on the mat** the cat → **sat on the mat** the cat sat → **on the mat** the cat sat on → **the mat** the cat sat on the → **mat**

Simple Transformer-based Language Model

```
from tensorflow.keras import layers
embed_dim = 256
latent_dim = 2048
num_heads = 2

inputs = keras.Input(shape=(None,), dtype="int64")
x = PositionalEmbedding(sequence_length, vocab_size, embed_dim)(inputs)
x = TransformerDecoder(embed_dim, latent_dim, num_heads)(x, x)
outputs = layers.Dense(vocab_size, activation="softmax")(x)
model = keras.Model(inputs, outputs)
model.compile(loss="sparse_categorical_crossentropy", optimizer="rmsprop")
```

Softmax over possible vocabulary words, computed for each output sequence timestep.

Text-Generation Callback

```
import numpy as np

tokens_index = dict(enumerate(text_vectorization.get_vocabulary()))

def sample_next(predictions, temperature=1.0):
    predictions = np.asarray(predictions).astype("float64")
    predictions = np.log(predictions) / temperature
    exp_preds = np.exp(predictions)
    predictions = exp_preds / np.sum(exp_preds)
    probas = np.random.multinomial(1, predictions, 1)
    return np.argmax(probas)

class TextGenerator(keras.callbacks.Callback):
    def __init__(self,
                 prompt,
                 generate_length,
                 model_input_length,
                 temperatures=(1.,),
                 print_freq=1):
        self.prompt = prompt
        self.generate_length = generate_length
        self.model_input_length = model_input_length
        self.temperatures = temperatures
        self.print_freq = print_freq

    def on_epoch_end(self, epoch, logs=None):
        if (epoch + 1) % self.print_freq != 0:
            return
        for temperature in self.temperatures:
            print("== Generating with temperature", temperature)
            sentence = self.prompt
            for i in range(self.generate_length):
                tokenized_sentence = text_vectorization([sentence])
                predictions = self.model(tokenized_sentence)
                next_token = sample_next(predictions[0, i, :])
                sampled_token = tokens_index[next_token]
                sentence += " " + sampled_token
            print(sentence)

prompt = "This movie"
text_gen_callback = TextGenerator(
    prompt,
    generate_length=50,
    model_input_length=sequence_length,
    temperatures=(0.2, 0.5, 0.7, 1., 1.5))
```

Dict that maps word indices back to strings, to be used for text decoding

Implements variable-temperature sampling from a probability distribution

Prompt that we use to seed text generation

How many words to generate

Range of temperatures to use for sampling

When generating text, we start from our prompt.

Feed the current sequence into our model.

Append the new word to the current sequence and repeat.

Retrieve the predictions for the last timestep, and use them to sample a new word.

We'll use a diverse range of temperatures to sample text, to demonstrate the effect of temperature on text generation.

Example Output

```
model.fit(lm_dataset, epochs=200, callbacks=[text_gen_callback])
```

With `temperature=0.2`

- "this movie is a [UNK] of the original movie and the first half hour of the movie is pretty good but it is a very good movie it is a good movie for the time period"

With `temperature=1.0`

- "this movie was entertaining i felt the plot line was loud and touching but on a whole watch a stark contrast to the artistic of the original we watched the original version of england however whereas arc was a bit of a little too ordinary the [UNK] were the present parent [UNK]"

With `temperature=1.5`

- "this movie was possibly the worst film about that 80 women its as weird insightful actors like barker movies but in great buddies yes no decorated shield even [UNK] land dinosaur ralph ian was must make a play happened falls after miscast [UNK] bach not really not wrestlemania seriously sam didnt exist"

Perplexity of a Language Model

- Measures how well a language model predicts words
  - Perplexity = 1 is perfect
  - Perplexity = vocab_size is terrible: achieved by always predicting $p_i = 1/\text{vocab\_size}$
- Perplexity is the geometric mean of inverted predicted probabilities, where an inverted predicted probability can be viewed as the number of words considered
- Compare language models using the *same* vocabulary

$$\left(\prod_{i=1}^{N}\frac{1}{p_i}\right)^{\frac{1}{N}} = exp\left(log\left(\left(\prod_{i=1}^{N}\frac{1}{p_i}\right)^{\frac{1}{N}}\right)\right) = exp\left(\frac{1}{N}log\left(\prod_{i=1}^{N}\frac{1}{p_i}\right)\right) = exp\left(\frac{1}{N}\sum_{i=1}^{N}log\left(\frac{1}{p_i}\right)\right)$$

$$= exp\left(\frac{1}{N}\sum_{i=1}^{N}\big(log(1) - log(p_i)\big)\right) = exp\left(\frac{1}{N}\sum_{i=1}^{N}\big(0 - log(p_i)\big)\right) = exp\left(\frac{1}{N}\sum_{i=1}^{N}\big(-log(p_i)\big)\right)$$

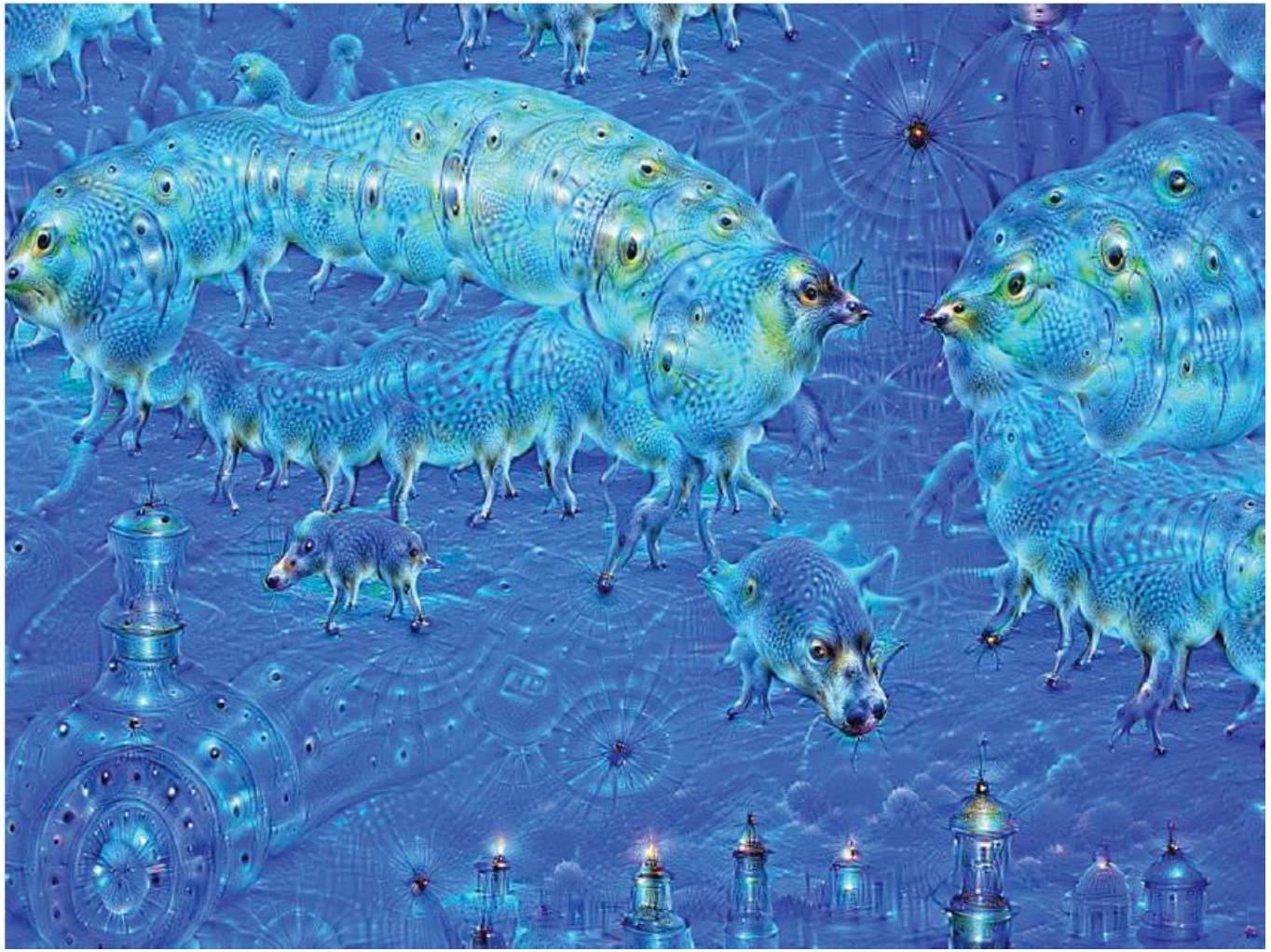
$$= exp\left(\frac{-\sum_{i=1}^{N}\big(log(p_i)\big)}{N}\right)$$

Reminder: arithmetic_mean ≥ geometric_mean ≥ harmonic_mean

Wrapping Up Text Generation

- 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 a 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.

Example DeepDream Output Image



DeepDream vs ConvNet Filter Visualization

- With DeepDream, you try to maximize the activation of entire layers rather than that of a specific filter, thus mixing together visualizations of large numbers of features at once.
- You start not from blank, slightly noisy input, but rather from an existing image—thus the resulting effects latch on to preexisting visual patterns, distorting elements of the image in a somewhat artistic fashion.
- The input images are processed at different scales (called *octaves*), which improves the quality of the visualizations.

Fetching an Input Image for DeepDream

```
from tensorflow import keras
import matplotlib.pyplot as plt

base_image_path = keras.utils.get_file(
    "coast.jpg", origin="https://img-datasets.s3.amazonaws.com/coast.jpg")

plt.axis("off")
plt.imshow(keras.utils.load_img(base_image_path))
```

DeepDream Feature Extractor

```
from tensorflow.keras.applications import inception_v3
model = inception_v3.InceptionV3(weights="imagenet", include_top=False)

layer_settings = {
    "mixed4": 1.0,
    "mixed5": 1.5,
    "mixed6": 2.0,
    "mixed7": 2.5,
}
outputs_dict = dict(
    [
        (layer.name, layer.output)
        for layer in [model.get_layer(name)
                      for name in layer_settings.keys()]
    ]
)
feature_extractor = keras.Model(inputs=model.inputs, outputs=outputs_dict)
```

Layers for which we try to maximize activation, as well as their weight in the total loss. You can tweak these setting to obtain new visual effects.

Symbolic outputs of each layer

Model that returns the activation values for every target layer (as a dict)

DeepDream Loss (mean squared activation)

Extract activations.

Initialize the loss to 0.

```
def compute_loss(input_image):
    features = feature_extractor(input_image)
    loss = tf.zeros(shape=())
    for name in features.keys():
        coeff = layer_settings[name]
        activation = features[name]
        loss += coeff * tf.reduce_mean(tf.square(activation[:, 2:-2, 2:-2, :]))
    return loss
```

We avoid border artifacts by only involving non-border pixels in the loss.

DeepDream Gradient Ascent

```
import tensorflow as tf

@tf.function
def gradient_ascent_step(image, learning_rate):
    with tf.GradientTape() as tape:
        tape.watch(image)
        loss = compute_loss(image)
    grads = tape.gradient(loss, image)
    grads = tf.math.l2_normalize(grads)
    image += learning_rate * grads
    return loss, image
```

We make the training step fast by compiling it as a tf.function.

Compute gradients of DeepDream loss with respect to the current image.

Normalize gradients (the same trick we used in chapter 9).

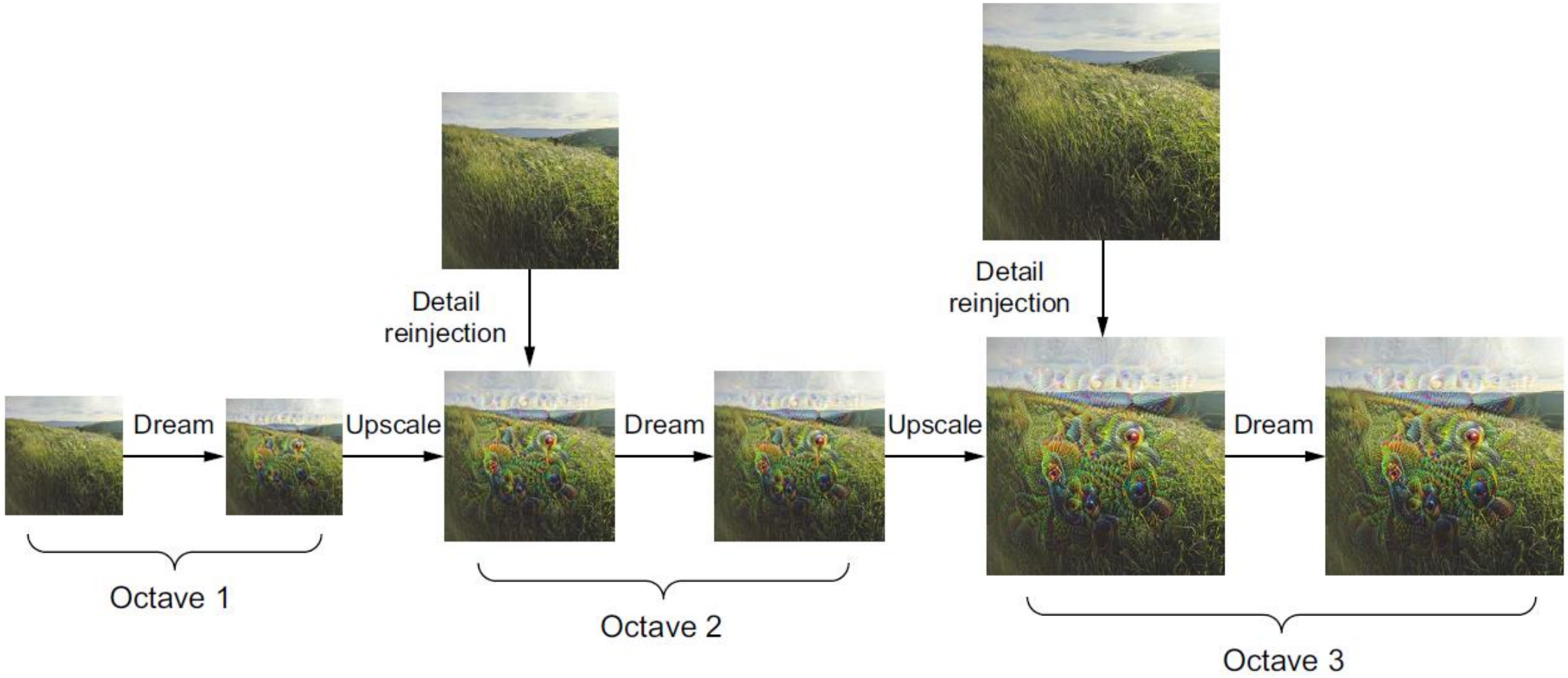
This runs gradient ascent for a given image scale (octave).

```
def gradient_ascent_loop(image, iterations, learning_rate, max_loss=None):
    for i in range(iterations):
        loss, image = gradient_ascent_step(image, learning_rate)
        if max_loss is not None and loss > max_loss:
            break
        print(f"... Loss value at step {i}: {loss:.2f}")
    return image
```

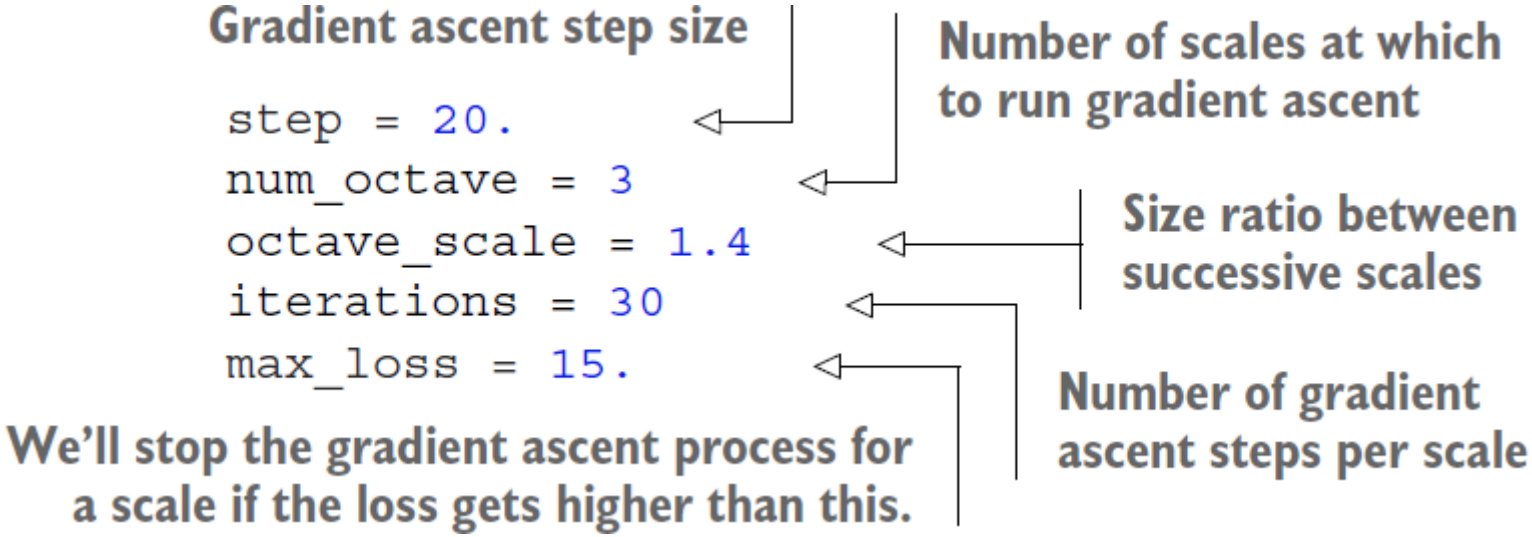
Repeatedly update the image in a way that increases the DeepDream loss.

Break out if the loss crosses a certain threshold (over-optimizing would create unwanted image artifacts).

Multi-Scale Dreaming



DeepDream Hyperparameters



DeepDream Image Processing

```
import numpy as np

def preprocess_image(image_path):
    img = keras.utils.load_img(image_path)
    img = keras.utils.img_to_array(img)
    img = np.expand_dims(img, axis=0)
    img = keras.applications.inception_v3.preprocess_input(img)
    return img

def deprocess_image(img):
    img = img.reshape((img.shape[1], img.shape[2], 3))
    img /= 2.0
    img += 0.5
    img *= 255.
    img = np.clip(img, 0, 255).astype("uint8")
    return img
```

Util function to open, resize, and format pictures into appropriate arrays

Util function to convert a NumPy array into a valid image

Undo inception v3 preprocessing.

Convert to uint8 and clip to the valid range [0, 255].

Gradient Ascent over Multiple Scales

```
original_img = preprocess_image(base_image_path)
original_shape = original_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]

shrunk_original_img = tf.image.resize(original_img, successive_shapes[0])

img = tf.identity(original_img)
for i, shape in enumerate(successive_shapes):
    print(f"Processing octave {i} with shape {shape}")
    img = tf.image.resize(img, shape)
    img = gradient_ascent_loop(
        img, iterations=iterations, learning_rate=step, max_loss=max_loss
    )
    upscaled_shrunk_original_img = tf.image.resize(shrunk_original_img, shape)
    same_size_original = tf.image.resize(original_img, shape)
    lost_detail = same_size_original - upscaled_shrunk_original_img
    img += lost_detail
    shrunk_original_img = tf.image.resize(original_img, shape)

keras.utils.save_img("dream.png", deprocess_image(img.numpy()))
```

Load the test image.

Compute the target shape of the image at different octaves.

Iterate over the different octaves.

Make a copy of the image (we need to keep the original around).

Scale up the dream image.

Run gradient ascent, altering the dream.

Compute the high-quality version of the original image at this size.

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

Re-inject lost detail into the dream.

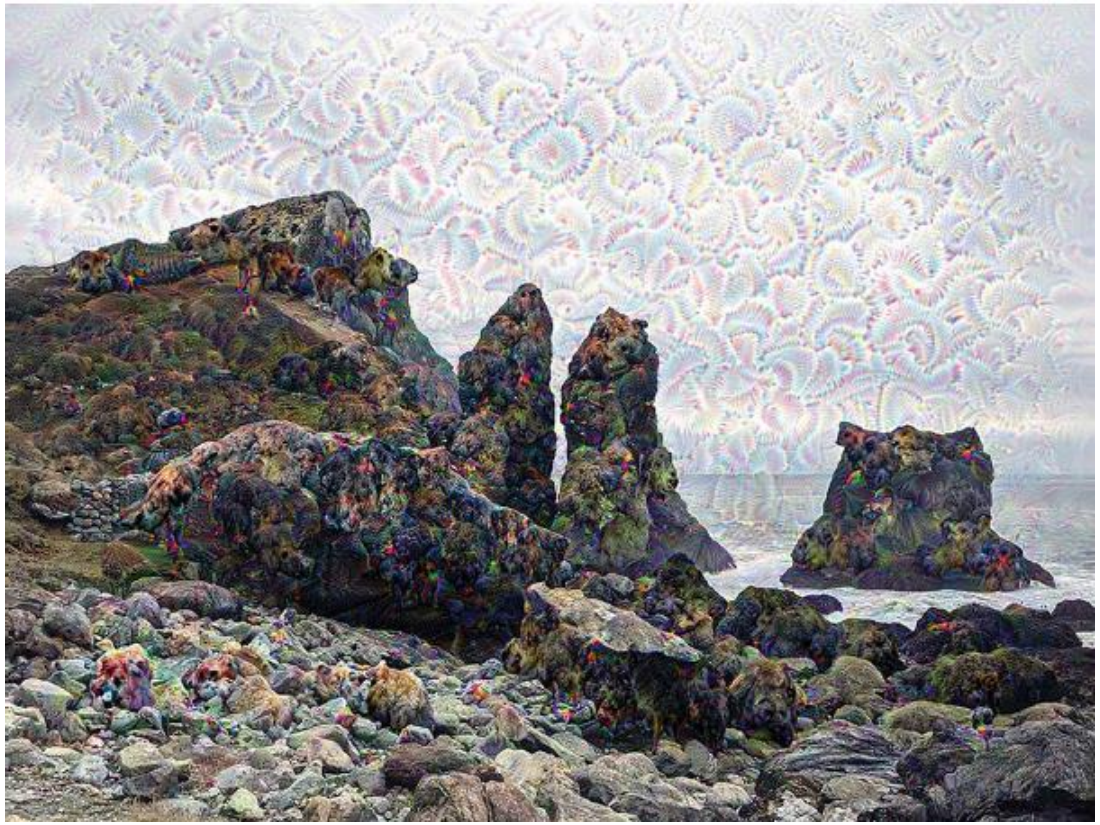
Save the final result.

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

[51%, 71%, 100%]

lost_detail = difference between original_rescaled and upscaled_shrunk

DeepDream Output Images



Varying layer weights

Wrapping Up DeepDream

- 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.

Neural Style Transfer Example

Content target

Tubingen, Germany

+

Style reference

Van Gogh's Starry Night

=

Combination image

Neural Style Transfer Input Images

```
from tensorflow import keras

base_image_path = keras.utils.get_file(
    "sf.jpg", origin="https://img-datasets.s3.amazonaws.com/sf.jpg")
style_reference_image_path = keras.utils.get_file(
    "starry_night.jpg",
    origin="https://img-datasets.s3.amazonaws.com/starry_night.jpg")
original_width, original_height = keras.utils.load_img(base_image_path).size
img_height = 400
img_width = round(original_width * img_height / original_height)
```

Path to the image we want to transform

Path to the style image

Dimensions of the generated picture

San Francisco

Neural Style Transfer Image Processing

```
import numpy as np

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

def deprocess_image(img):
    img = img.reshape((img_height, img_width, 3))
    img[:, :, 0] += 103.939
    img[:, :, 1] += 116.779
    img[:, :, 2] += 123.68
    img = img[:, :, ::-1]
    img = np.clip(img, 0, 255).astype("uint8")
    return img
```

Util function to open, resize, and format pictures into appropriate arrays

Util function to convert a NumPy array into a valid image

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.

Neural Style Transfer Feature Extractor

```
model = keras.applications.vgg19.VGG19(weights="imagenet", include_top=False)

outputs_dict = dict([(layer.name, layer.output) for layer in model.layers])
feature_extractor = keras.Model(inputs=model.inputs, outputs=outputs_dict)
```

Build a VGG19 model loaded with pretrained ImageNet weights.

Model that returns the activation values for every target layer (as a dict)

Neural Style Transfer Loss Functions

```
def content_loss(base_img, combination_img):
    return tf.reduce_sum(tf.square(combination_img - base_img))

def gram_matrix(x):
    x = tf.transpose(x, (2, 0, 1))
    features = tf.reshape(x, (tf.shape(x)[0], -1))
    gram = tf.matmul(features, tf.transpose(features))
    return gram

def style_loss(style_img, combination_img):
    S = gram_matrix(style_img)
    C = gram_matrix(combination_img)
    channels = 3
    size = img_height * img_width
    return tf.reduce_sum(tf.square(S - C)) / (4.0 * (channels ** 2) * (size ** 2))

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

style is represented as dot products between feature map activations

Divide by (2 * |Channels| * |Pixels|) squared

vertical differences

horizontal differences

Configuring Neural Style Transfer Loss

```
style_layer_names = [
    "block1_conv1",
    "block2_conv1",
    "block3_conv1",
    "block4_conv1",
    "block5_conv1",
]
content_layer_name = "block5_conv2"
total_variation_weight = 1e-6
style_weight = 1e-6
content_weight = 2.5e-8

def compute_loss(combination_image, base_image, style_reference_image):
    input_tensor = tf.concat(
        [base_image, style_reference_image, combination_image], axis=0)
    features = feature_extractor(input_tensor)
    loss = tf.zeros(shape=())
    layer_features = features[content_layer_name]
    base_image_features = layer_features[0, :, :, :]
    combination_features = layer_features[2, :, :, :]
    loss = loss + content_weight * content_loss(
        base_image_features, combination_features
    )
    for layer_name in style_layer_names:
        layer_features = features[layer_name]
        style_reference_features = layer_features[1, :, :, :]
        combination_features = layer_features[2, :, :, :]
        style_loss_value = style_loss(
            style_reference_features, combination_features)
        loss += (style_weight / len(style_layer_names)) * style_loss_value

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

List of layers to use for the style loss

The layer to use for the content loss

Contribution weight of the total variation loss

Contribution weight of the style loss

Contribution weight of the content loss

Initialize the loss to 0.

Add the content loss.

Add the style loss.

Add the total variation loss.

Note the really small weights!
Contrast to the learning rate.

Configuring the Gradient Descent Process

```
import tensorflow as tf

@tf.function
def compute_loss_and_grads(
    combination_image, base_image, style_reference_image):
    with tf.GradientTape() as tape:
        loss = compute_loss(
            combination_image, base_image, style_reference_image)
    grads = tape.gradient(loss, combination_image)
    return loss, grads

optimizer = keras.optimizers.SGD(
    keras.optimizers.schedules.ExponentialDecay(
        initial_learning_rate=100.0, decay_steps=100, decay_rate=0.96
    )
)
base_image = preprocess_image(base_image_path)
style_reference_image = preprocess_image(style_reference_image_path)
combination_image = tf.Variable(preprocess_image(base_image_path))

iterations = 4000
for i in range(1, iterations + 1):
    loss, grads = compute_loss_and_grads(
        combination_image, base_image, style_reference_image
    )
    optimizer.apply_gradients([(grads, combination_image)])
    if i % 100 == 0:
        print(f"Iteration {i}: loss={loss:.2f}")
        img = deprocess_image(combination_image.numpy())
        fname = f"combination_image_at_iteration_{i}.png"
        keras.utils.save_img(fname, img)
```

We make the training step fast by compiling it as a tf.function.

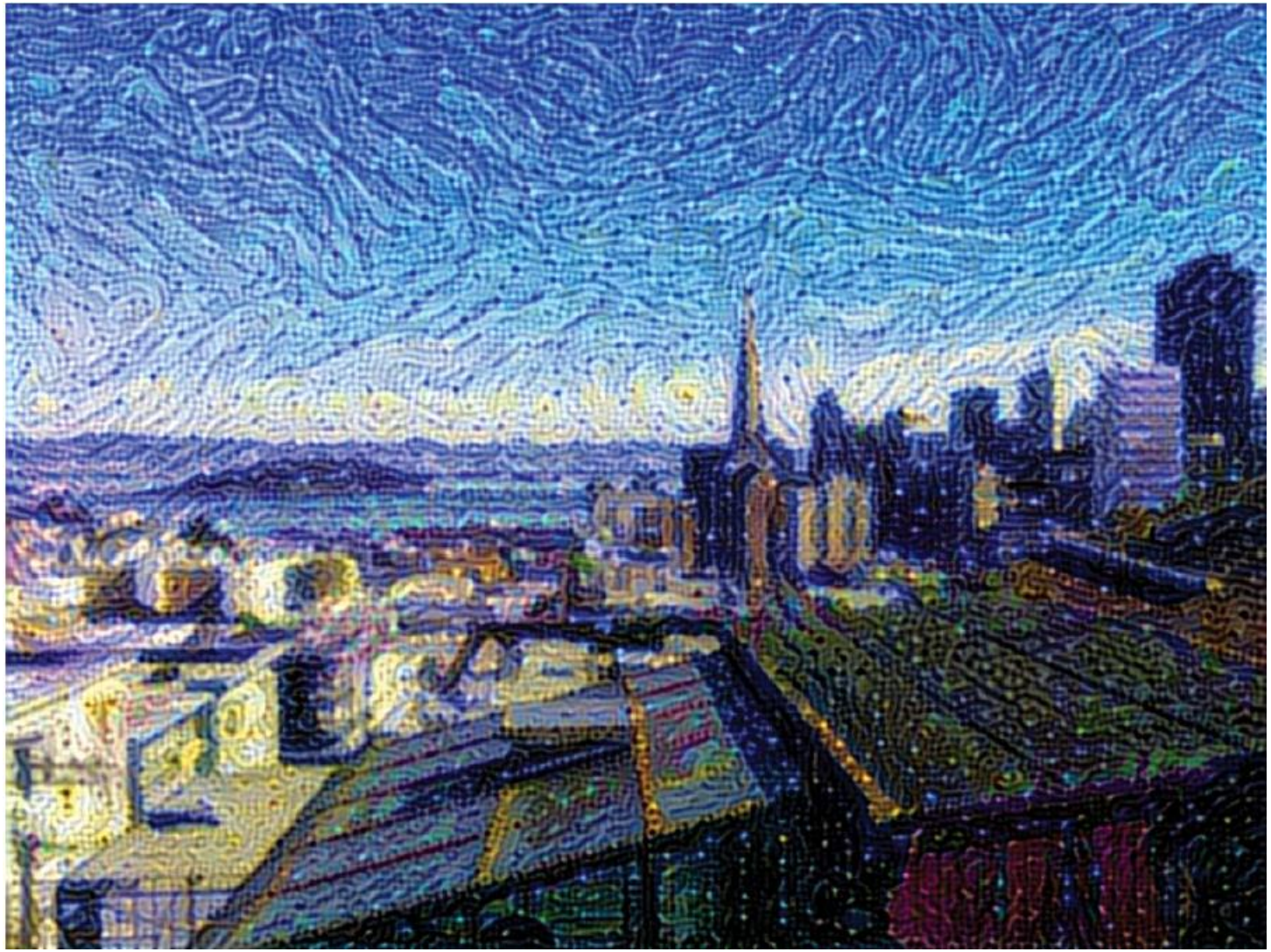
We'll start with a learning rate of 100 and decrease it by 4% every 100 steps.

Use a Variable to store the combination image since we'll be updating it during training.

Update the combination image in a direction that reduces the style transfer loss.

Save the combination image at regular intervals.

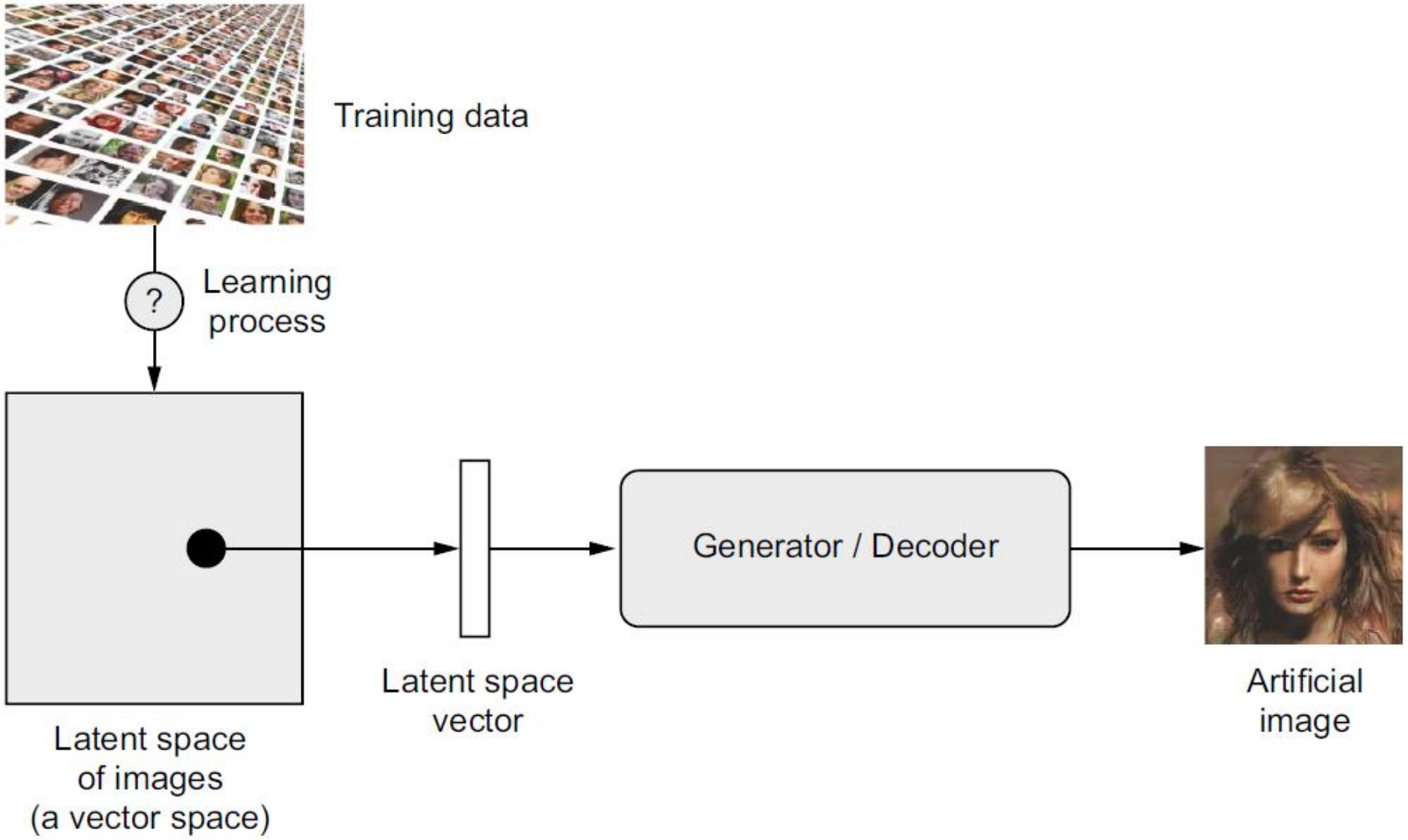
Neural Style Transfer Output Image



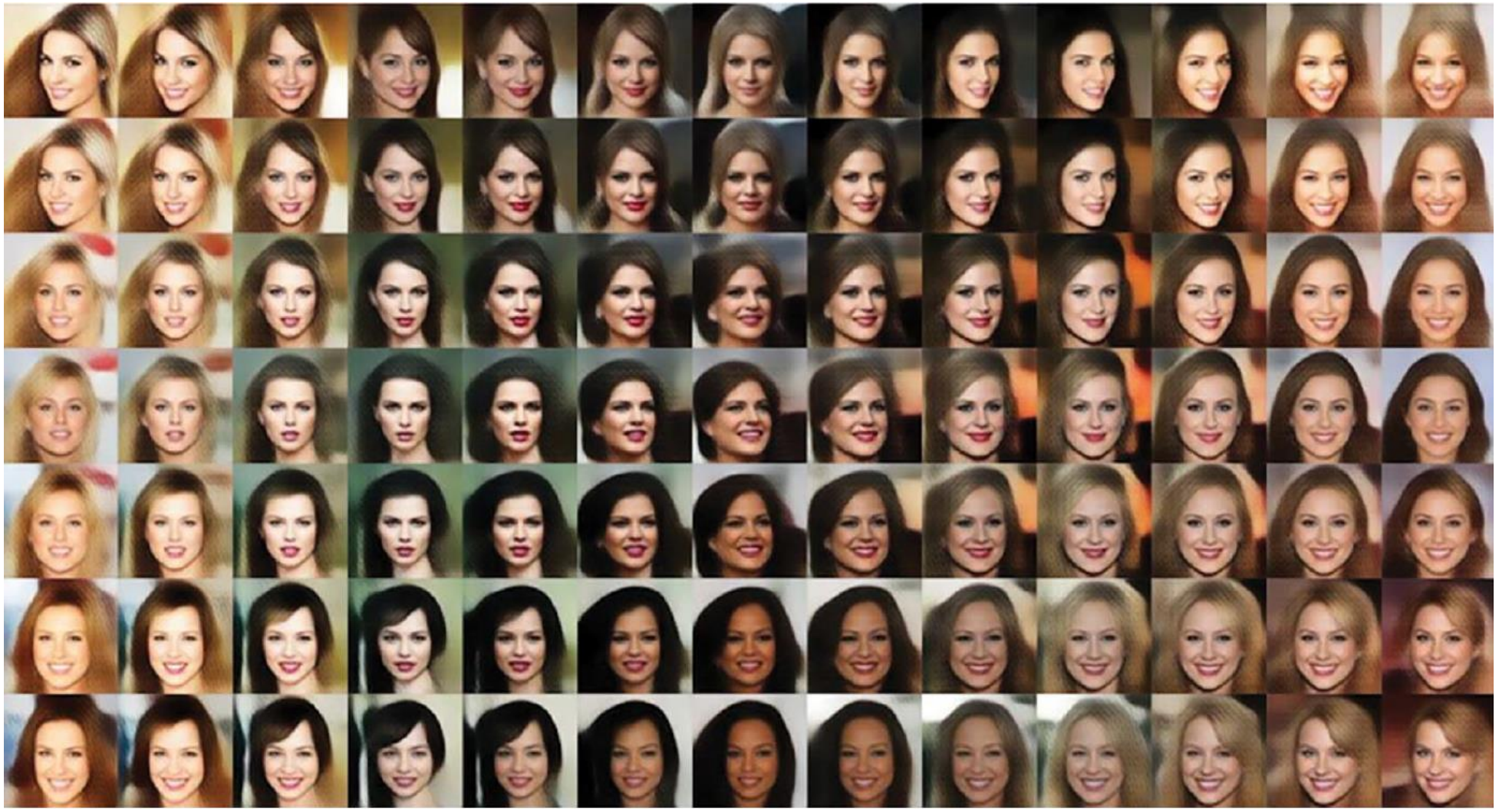
Wrapping Up Neural Style Transfer

- 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.

Generative Modeling for Images



Continuous Space of Faces Generated by VAE



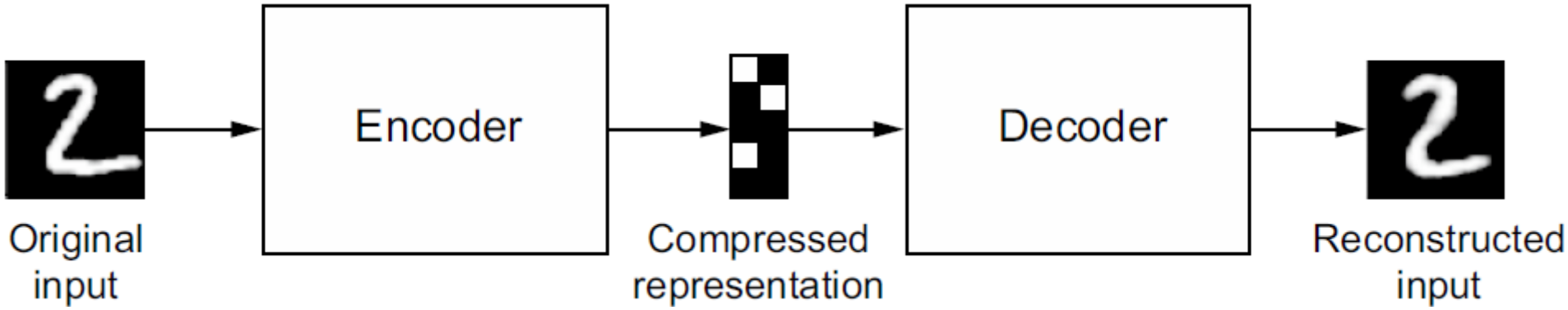
The Smile Vector

From the paper …

- the smile vector can be computed by simply subtracting the mean vector for images without the smile attribute from the mean vector for images with the smile attribute
- the smile vector can then be applied to in a positive or negative direction to manipulate this visual attribute

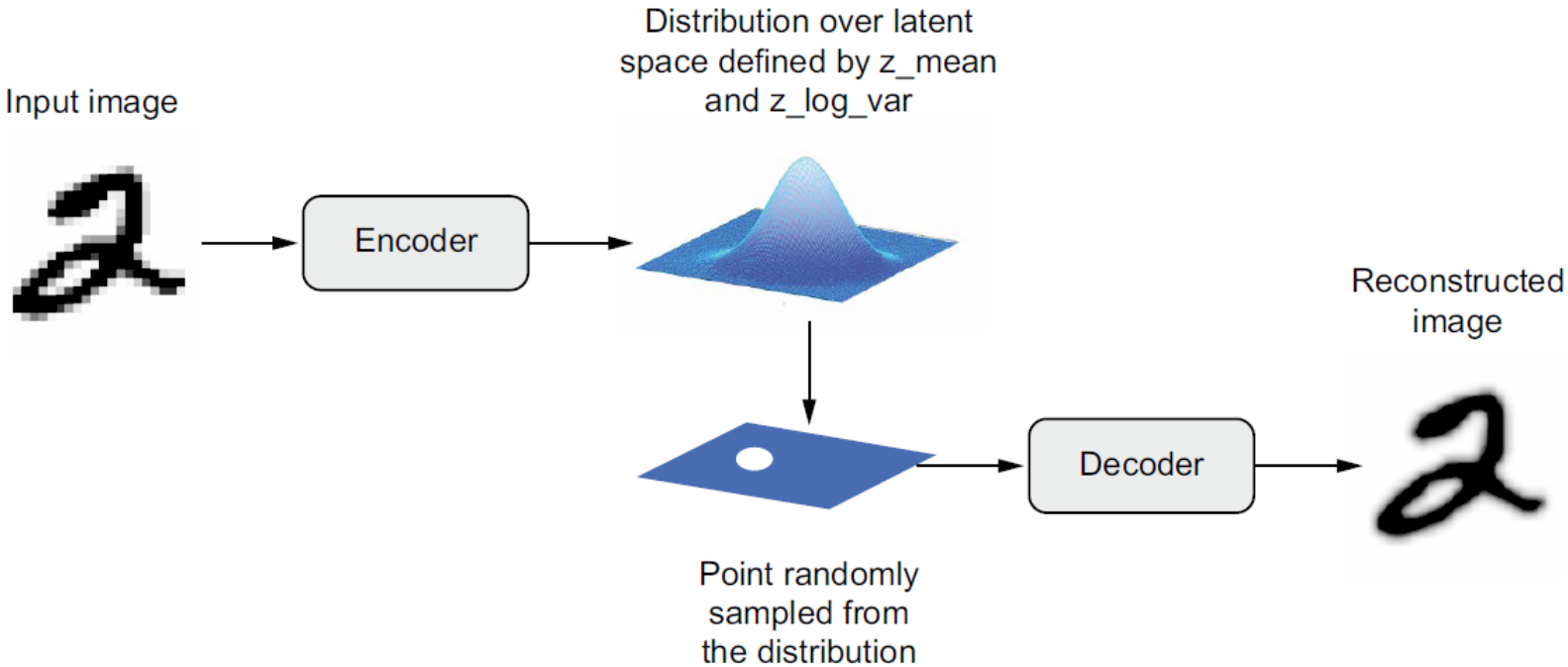
Autoencoder

For a denoising autoencoder, we can apply dropout to the input image pixels and train the model to predict the original input image …



Variational Autoencoder (VAE)

1. An encoder module turns the input sample, `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.



VAE in Action

```
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 mean and variance parameters

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

VAE: Encoder Model

```
from tensorflow import keras
from tensorflow.keras import layers

latent_dim = 2

encoder_inputs = keras.Input(shape=(28, 28, 1))
x = layers.Conv2D(
    32, 3, activation="relu", strides=2, padding="same")(encoder_inputs)
x = layers.Conv2D(64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Flatten()(x)
x = layers.Dense(16, activation="relu")(x)
z_mean = layers.Dense(latent_dim, name="z_mean")(x)
z_log_var = layers.Dense(latent_dim, name="z_log_var")(x)
encoder = keras.Model(encoder_inputs, [z_mean, z_log_var], name="encoder")
```

Dimensionality of the latent space: a 2D plane

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

VAE: Encoder Model Summary

```
>>> encoder.summary()
Model: "encoder"
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
input_1 (InputLayer)            [(None, 28, 28, 1)]  0
__________________________________________________________________________________________________
conv2d (Conv2D)                 (None, 14, 14, 32)   320         input_1[0][0]
__________________________________________________________________________________________________
conv2d_1 (Conv2D)               (None, 7, 7, 64)     18496       conv2d[0][0]
__________________________________________________________________________________________________
flatten (Flatten)               (None, 3136)         0           conv2d_1[0][0]
__________________________________________________________________________________________________
dense (Dense)                   (None, 16)           50192       flatten[0][0]
__________________________________________________________________________________________________
z_mean (Dense)                  (None, 2)            34          dense[0][0]
__________________________________________________________________________________________________
z_log_var (Dense)               (None, 2)            34          dense[0][0]
==================================================================================================
Total params: 69,076
Trainable params: 69,076
Non-trainable params: 0
```

VAE: Sampler

```
import tensorflow as tf

class Sampler(layers.Layer):
    def call(self, z_mean, z_log_var):
        batch_size = tf.shape(z_mean)[0]
        z_size = tf.shape(z_mean)[1]
        epsilon = tf.random.normal(shape=(batch_size, z_size))
        return z_mean + tf.exp(0.5 * z_log_var) * epsilon
```

Apply the VAE sampling formula.

Draw a batch of random normal vectors.

log((sigma**2)**0.5) = 0.5 * log(sigma**2)
… so …
exp(log((sigma**2)**0.5)) = (sigma**2)**0.5 = sigma

VAE: Decoder Model

Input where we'll feed z

Produce the same number of coefficients that we had at the level of the Flatten layer in the encoder.

Revert the Flatten layer of the encoder.

Revert the Conv2D layers of the encoder.

```
latent_inputs = keras.Input(shape=(latent_dim,))
x = layers.Dense(7 * 7 * 64, activation="relu")(latent_inputs)
x = layers.Reshape((7, 7, 64))(x)
x = layers.Conv2DTranspose(64, 3, activation="relu", strides=2, padding="same")(x)
x = layers.Conv2DTranspose(32, 3, activation="relu", strides=2, padding="same")(x)
decoder_outputs = layers.Conv2D(1, 3, activation="sigmoid", padding="same")(x)
decoder = keras.Model(latent_inputs, decoder_outputs, name="decoder")
```

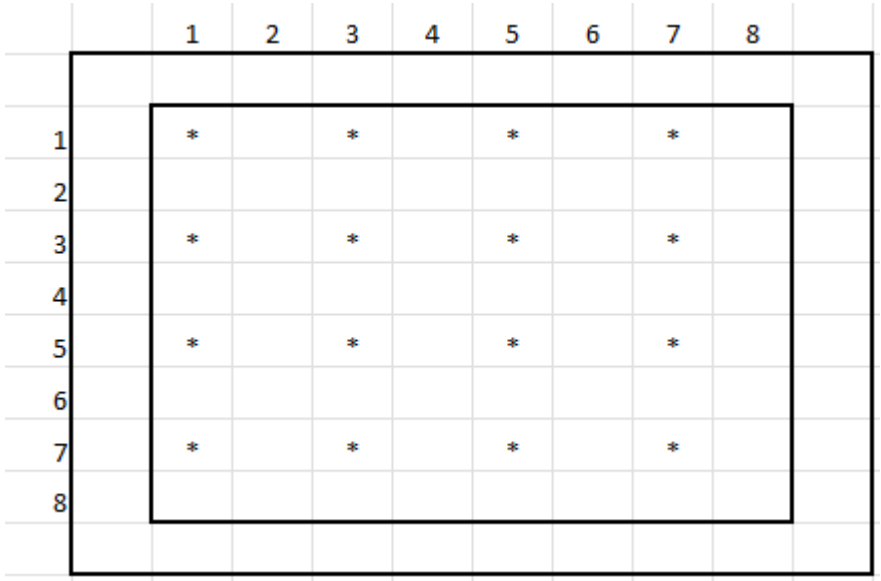
The output ends up with shape (28, 28, 1).

VAE: Decoder Model Summary

```
>>> decoder.summary()
Model: "decoder"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 2)]               0
_________________________________________________________________
dense_1 (Dense)              (None, 3136)              9408
_________________________________________________________________
reshape (Reshape)            (None, 7, 7, 64)          0
_________________________________________________________________
conv2d_transpose (Conv2DTran (None, 14, 14, 64)        36928
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 28, 28, 32)        18464
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 28, 28, 1)         289
=================================================================
Total params: 65,089
Trainable params: 65,089
Non-trainable params: 0
```

Transposed Convolution with Stride = 2

- Convolution:
  - input_size: (8, 8)
  - kernel_size: (3, 3)
  - padding_size: “same”
  - strides = 2
  - output_size: (4, 4)
- Transposed Convolution:
  - input_size: (4, 4)
  - kernel_size: (3, 3)
  - padding_size: “same”
  - strides: 2
  - output_size: (8, 8)



asterisks mark …

- center pixel of (3,3) filter for convolution
- input pixel for transposed convolution

VAE Model Class (with custom train step)

```
class VAE(keras.Model):
    def __init__(self, encoder, decoder, **kwargs):
        super().__init__(**kwargs)
        self.encoder = encoder
        self.decoder = decoder
        self.sampler = Sampler()
        self.total_loss_tracker = keras.metrics.Mean(name="total_loss")
        self.reconstruction_loss_tracker = keras.metrics.Mean(
            name="reconstruction_loss")
        self.kl_loss_tracker = keras.metrics.Mean(name="kl_loss")

    @property
    def metrics(self):
        return [self.total_loss_tracker,
                self.reconstruction_loss_tracker,
                self.kl_loss_tracker]

    def train_step(self, data):
        with tf.GradientTape() as tape:
            z_mean, z_log_var = self.encoder(data)
            z = self.sampler(z_mean, z_log_var)
            reconstruction = decoder(z)
            reconstruction_loss = tf.reduce_mean(
                tf.reduce_sum(
                    keras.losses.binary_crossentropy(data, reconstruction),
                    axis=(1, 2)
                )
            )
            kl_loss = -0.5 * (1 + z_log_var - tf.square(z_mean) -
                tf.exp(z_log_var))
            total_loss = reconstruction_loss + tf.reduce_mean(kl_loss)
        grads = tape.gradient(total_loss, self.trainable_weights)
        self.optimizer.apply_gradients(zip(grads, self.trainable_weights))
        self.total_loss_tracker.update_state(total_loss)
        self.reconstruction_loss_tracker.update_state(reconstruction_loss)
        self.kl_loss_tracker.update_state(kl_loss)
        return {
            "total_loss": self.total_loss_tracker.result(),
            "reconstruction_loss": self.reconstruction_loss_tracker.result(),
            "kl_loss": self.kl_loss_tracker.result(),
        }
```

We use these metrics to keep track of the loss averages over each epoch.

We list the metrics in the metrics property to enable the model to reset them after each epoch (or between multiple calls to fit()/evaluate()).

We sum the reconstruction loss over the spatial dimensions (axes 1 and 2) and take its mean over the batch dimension.

Add the regularization term (Kullback–Leibler divergence).

gradient(reconstruction_loss, prediction)
= prediction - actual

Kullback-Leibler (KL) Divergence

- Pronounced “cool back – lie blur”
- For a VAE, we’re comparing a Gaussian distribution (p) to a standard Gaussian distribution (q), where a standard Gaussian distribution has …
  - mean = 0
  - standard_deviation = 1
- Think of KL Divergence as a regularization term for the VAE

Kullback-Leibler Divergence

- $P \sim N(\mu_1, \sigma_1) \Rightarrow p(x) = \frac{1}{\sqrt{2\pi\sigma_1^2}} \exp\left(-\frac{1}{2}\frac{(x-\mu_1)^2}{\sigma_1^2}\right)$
- $Q \sim N(\mu_2, \sigma_2) \Rightarrow q(x) = \frac{1}{\sqrt{2\pi\sigma_2^2}} \exp\left(-\frac{1}{2}\frac{(x-\mu_2)^2}{\sigma_2^2}\right)$

For standard Gaussian: $\mu_2 = 0, \sigma_2 = 1$

- $D_{KL}(P \parallel Q) = \int_{-\infty}^{\infty} p(x) \log \frac{p(x)}{q(x)} dx$

$$= \int_{-\infty}^{\infty} p(x) \left(log\big(p(x)\big) - log\big(q(x)\big)\right) dx$$

$$= \int_{-\infty}^{\infty} p(x) log\big(p(x)\big)\, dx - \int_{-\infty}^{\infty} p(x) log\big(q(x)\big) dx$$

$$= \left(-\frac{1}{2}(log(2\pi\sigma_1^2) + 1)\right) - \left(-\frac{1}{2}(log(2\pi) + \mu_1^2 + \sigma_1^2)\right)$$

$$= -\frac{1}{2}(1 + log(\sigma_1^2) - \mu_1^2 - \sigma_1^2)$$

VAE

$$\int_{-\infty}^{\infty} p(x)log\big(p(x)\big)\,dx$$

$$= \int_{-\infty}^{\infty} p(x)log\left(\frac{1}{\sqrt{2\pi\sigma_1^2}}\exp\left(-\frac{1}{2}\frac{(x-\mu_1)^2}{\sigma_1^2}\right)\right)dx$$

$$= \int_{-\infty}^{\infty} p(x)\left(log\left(\frac{1}{\sqrt{2\pi\sigma_1^2}}\right)+log\left(\exp\left(-\frac{1}{2}\frac{(x-\mu_1)^2}{\sigma_1^2}\right)\right)\right)dx$$

$$= -\frac{1}{2}log(2\pi\sigma_1^2)\int_{-\infty}^{\infty} p(x)\,dx - \frac{1}{2\sigma_1^2}\int_{-\infty}^{\infty} p(x)(x-\mu_1)^2dx$$

$$= -\frac{1}{2}log(2\pi\sigma_1^2) * 1 - \frac{1}{2\sigma_1^2} * \sigma_1^2$$

$$= -\frac{1}{2}(log(2\pi\sigma_1^2)+1)$$

VAE

$$\int_{-\infty}^{\infty} p(x) log\big(q(x)\big)\, dx$$

$$= \int_{-\infty}^{\infty} p(x) log\left(\frac{1}{\sqrt{2\pi\sigma_2^2}} \exp\left(-\frac{1}{2}\frac{(x-\mu_2)^2}{\sigma_2^2}\right)\right) dx$$

$$= \int_{-\infty}^{\infty} p(x)\left(log\left(\frac{1}{\sqrt{2\pi\sigma_2^2}}\right) + log\left(\exp\left(-\frac{1}{2}\frac{(x-\mu_2)^2}{\sigma_2^2}\right)\right)\right) dx$$

$$= -\frac{1}{2} log(2\pi\sigma_2^2) \int_{-\infty}^{\infty} p(x)\, dx - \frac{1}{2\sigma_2^2} \int_{-\infty}^{\infty} p(x)(x-\mu_2)^2 dx$$

$$= -\frac{1}{2} log(2\pi\sigma_2^2) * 1 - \frac{1}{2\sigma_2^2} * (\mu_1^2 + \sigma_1^2 - 2\mu_1\mu_2 + \mu_2^2)$$

$$= -\frac{1}{2}(log(2\pi) + \mu_1^2 + \sigma_1^2)$$

Reminder:
$\sigma^2 = E(X^2) - E(X)^2$

Training a VAE for MNIST

```
import numpy as np

(x_train, _), (x_test, _) = keras.datasets.mnist.load_data()
mnist_digits = np.concatenate([x_train, x_test], axis=0)
mnist_digits = np.expand_dims(mnist_digits, -1).astype("float32") / 255

vae = VAE(encoder, decoder)
vae.compile(optimizer=keras.optimizers.Adam(), run_eagerly=True)
vae.fit(mnist_digits, epochs=30, batch_size=128)
```

We train on all MNIST digits, so we concatenate the training and test samples.

Note that we don't pass targets in fit(), since train_step() doesn't expect any.

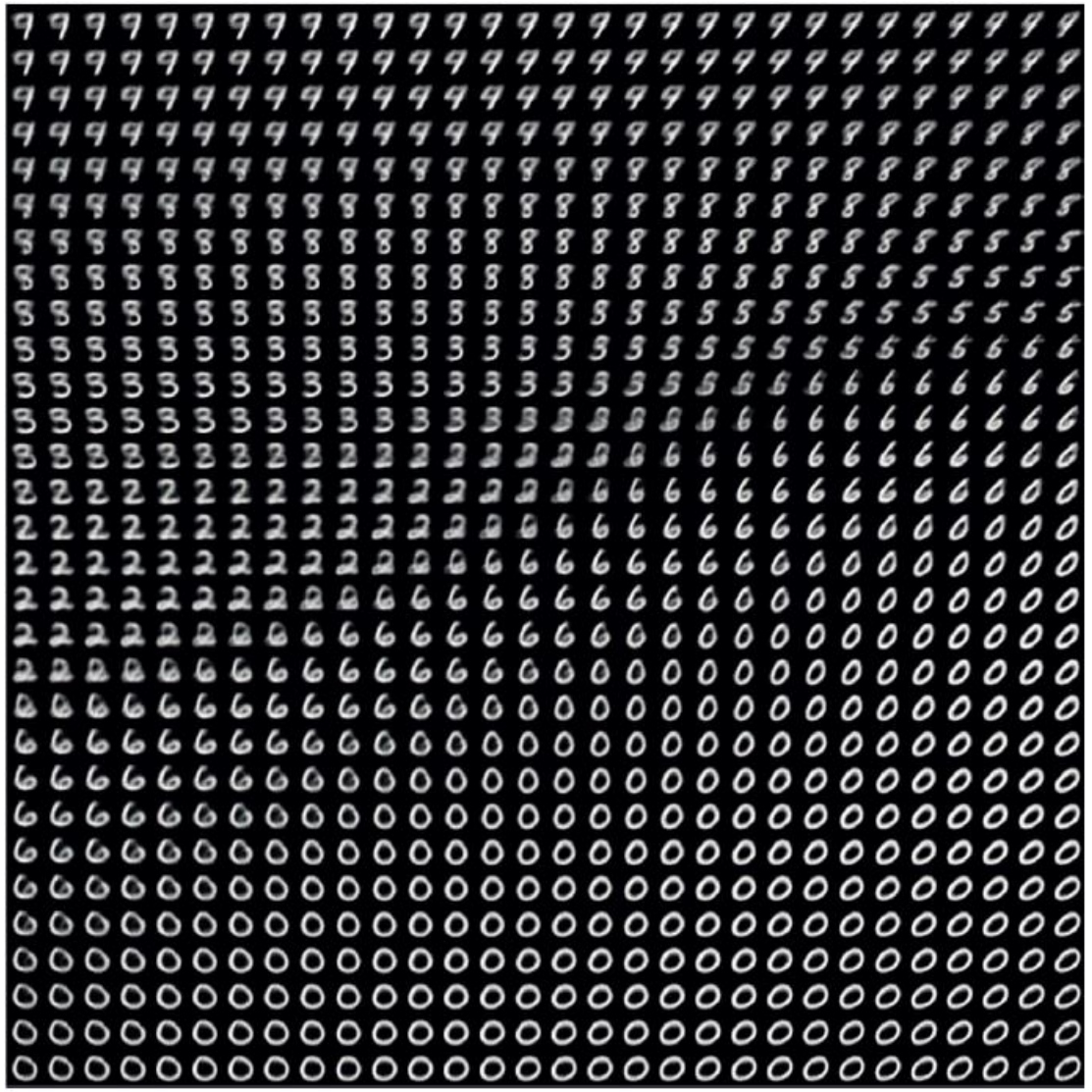
Note that we don't pass a loss argument in compile(), since the loss is already part of the train_step().

Sampling Images from the MNIST Latent Space

```
import matplotlib.pyplot as plt

n = 30                                  # We'll display a grid of 30 × 30 digits (900 digits total).
digit_size = 28
figure = np.zeros((digit_size * n, digit_size * n))

grid_x = np.linspace(-1, 1, n)          # Sample points linearly on a 2D grid.
grid_y = np.linspace(-1, 1, n)[::-1]

for i, yi in enumerate(grid_y):         # Iterate over grid locations.
    for j, xi in enumerate(grid_x):
        z_sample = np.array([[xi, yi]])
        x_decoded = vae.decoder.predict(z_sample)
        digit = x_decoded[0].reshape(digit_size, digit_size)
        figure[                         # For each location, sample a digit and add it to our figure.
            i * digit_size : (i + 1) * digit_size,
                j * digit_size : (j + 1) * digit_size,
            ] = digit

    plt.figure(figsize=(15, 15))
    start_range = digit_size // 2
    end_range = n * digit_size + start_range
    pixel_range = np.arange(start_range, end_range, digit_size)
    sample_range_x = np.round(grid_x, 1)
    sample_range_y = np.round(grid_y, 1)
    plt.xticks(pixel_range, sample_range_x)
    plt.yticks(pixel_range, sample_range_y)
    plt.xlabel("z[0]")
    plt.ylabel("z[1]")
    plt.axis("off")
    plt.imshow(figure, cmap="Greys_r")
```

Images from the MNIST Latent Space

2 morphs into 6 then 0 >>>

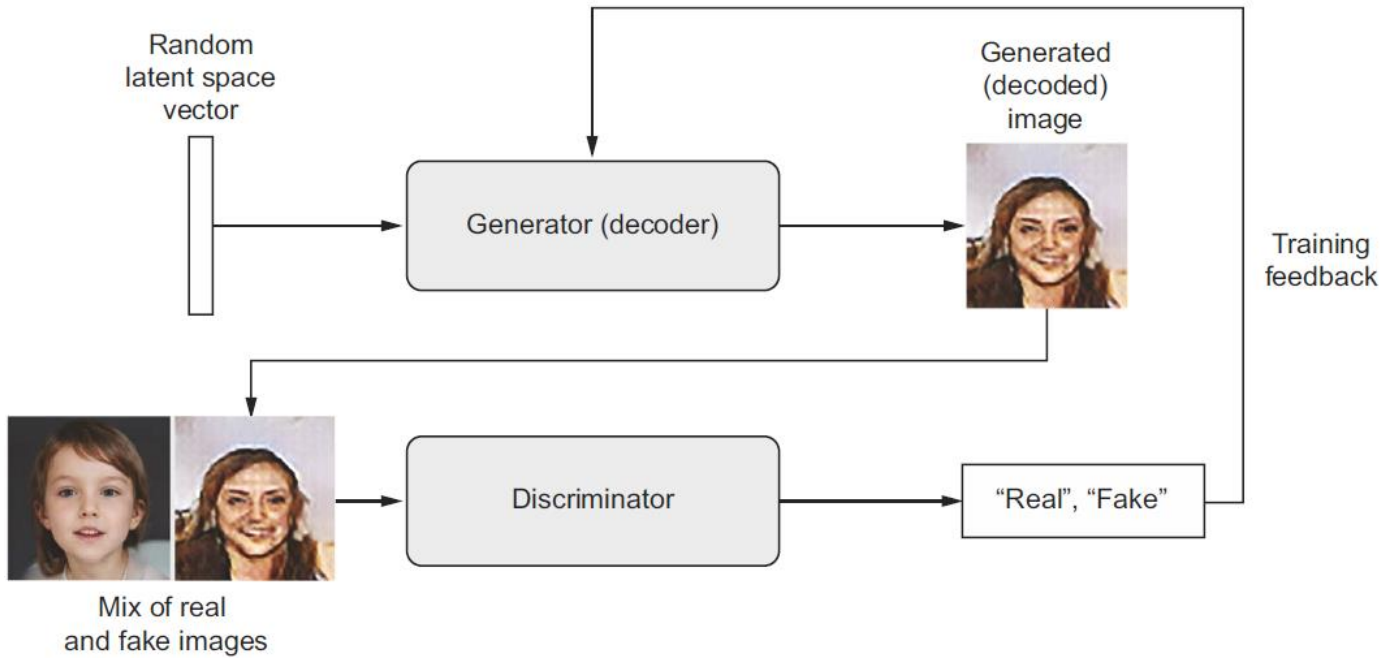


Wrapping Up the Variational Autoencoder

- 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 or 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.

Generative Adversarial Network (GAN) Models

- *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



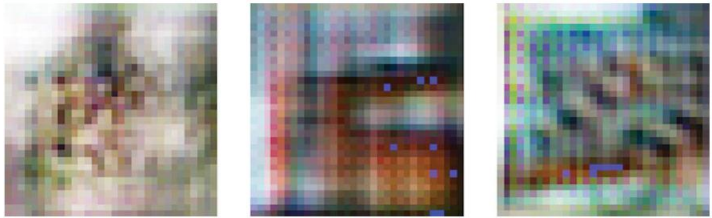
https://thispersondoesnotexist.com

Example GAN Operation

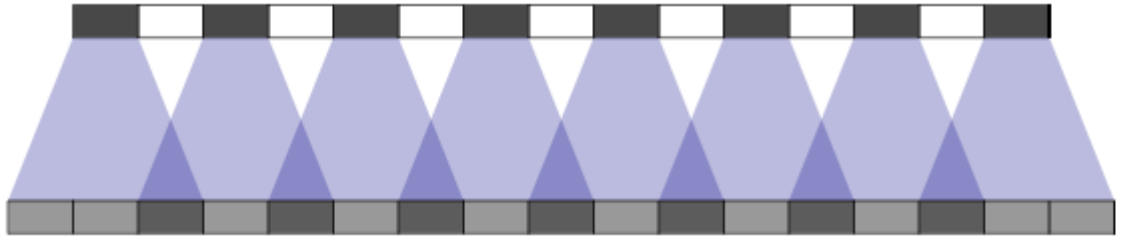
- A `generator` network maps vectors of shape `(latent_dim,)` to images of shape `(64, 64, 3)`.
- A `discriminator` network maps images of shape `(64, 64, 3)` to a binary score estimating the probability that the image is real.
- A `gan` network chains the generator and the discriminator together: `gan(x) = 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.
- We train the discriminator using examples of real and fake images along with "real"/"fake" labels, just as we train any regular image-classification model.
- To train the generator, we use the gradients of the generator's weights with regard to the loss of the `gan` model. This means that at every step, we 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, we train the generator to fool the discriminator.

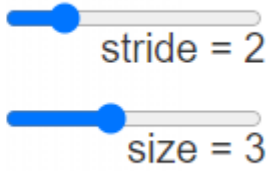
“A Bag of Tricks” for GANs

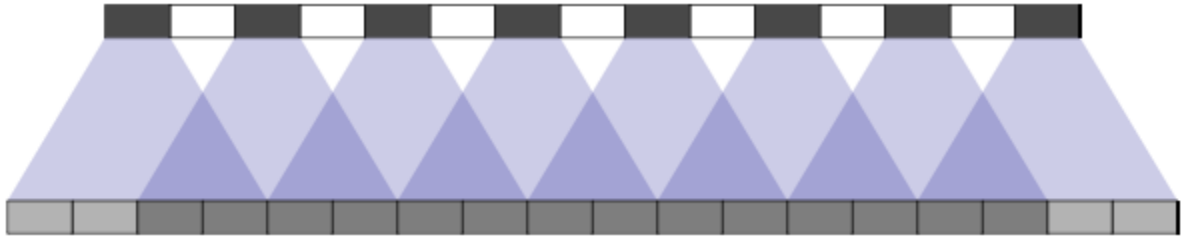
- We use strides instead of pooling for downsampling feature maps in the discriminator, just like we did in our VAE encoder.
- We sample points from the latent space using a *normal distribution* (Gaussian distribution), not a uniform distribution.
- Stochasticity is good for inducing robustness. Because 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 by adding random noise to the labels for the discriminator.
- 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 12.21). To fix this, we use a kernel size that’s divisible by the stride size whenever we use a strided `Conv2DTranspose` or `Conv2D` in both the generator and the discriminator.

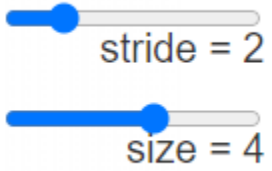


Checkerboarding









https://distill.pub/2016/deconv-checkerboard/

CelebrityFaces Attributes (CelebA)

```
!mkdir celeba_gan
!gdown --id 1O7m1010EJjLE5QxLZiM9Fpjs7Oj6e684 -O celeba_gan/data.zip
!unzip -qq celeba_gan/data.zip -d celeba_gan
```

Create a working directory.

Download the compressed data using gdown (available by default in Colab; install it otherwise).

Uncompress the data.

Once you've got the uncompressed images in a directory, you can use image_dataset_from_directory to turn it into a dataset. Since we just need the images—there are no labels—we'll specify label_mode=None.

```
from tensorflow import keras
dataset = keras.utils.image_dataset_from_directory(
    "celeba_gan",
    label_mode=None,
    image_size=(64, 64),
    batch_size=32,
    smart_resize=True)
dataset = dataset.map(lambda x: x / 255.)
import matplotlib.pyplot as plt
for x in dataset:
    plt.axis("off")
    plt.imshow((x.numpy() * 255).astype("int32")[0])
    break
```

Only the images will be returned—no labels.

We will resize the images to 64 × 64 by using a smart combination of cropping and resizing to preserve aspect ratio. We don't want face proportions to get distorted!

GAN: Discriminator Model

```
from tensorflow.keras import layers

discriminator = keras.Sequential(
    [
        keras.Input(shape=(64, 64, 3)),
        layers.Conv2D(64, kernel_size=4, strides=2, padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.Conv2D(128, kernel_size=4, strides=2, padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.Conv2D(128, kernel_size=4, strides=2, padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.Flatten(),
        layers.Dropout(0.2),
        layers.Dense(1, activation="sigmoid"),
    ],
    name="discriminator",
)
```

One dropout layer: an important trick!

GAN: Discriminator Model Summary

```
>>> discriminator.summary()
Model: "discriminator"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 32, 32, 64)        3136
_________________________________________________________________
leaky_re_lu (LeakyReLU)      (None, 32, 32, 64)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 16, 16, 128)       131200
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 16, 16, 128)       0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 8, 8, 128)         262272
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 8, 8, 128)         0
_________________________________________________________________
flatten (Flatten)            (None, 8192)              0
_________________________________________________________________
dropout (Dropout)            (None, 8192)              0
_________________________________________________________________
dense (Dense)                (None, 1)                 8193
=================================================================
Total params: 404,801
Trainable params: 404,801
Non-trainable params: 0
```

GAN: Generator Model

```
latent_dim = 128

generator = keras.Sequential(
    [
        keras.Input(shape=(latent_dim,)),
        layers.Dense(8 * 8 * 128),
        layers.Reshape((8, 8, 128)),
        layers.Conv2DTranspose(128, kernel_size=4, strides=2, padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.Conv2DTranspose(256, kernel_size=4, strides=2, padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.Conv2DTranspose(512, kernel_size=4, strides=2, padding="same"),
        layers.LeakyReLU(alpha=0.2),
        layers.Conv2D(3, kernel_size=5, padding="same", activation="sigmoid"),
    ],
    name="generator",
)
```

The latent space will be made of 128-dimensional vectors.

Produce the same number of coefficients we had at the level of the Flatten layer in the encoder.

Revert the Flatten layer of the encoder.

Revert the Conv2D layers of the encoder.

The output ends up with shape (28, 28, 1).

We use LeakyReLU as our activation.

GAN: Generator Model Summary

```
>>> generator.summary()
Model: "generator"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_1 (Dense)              (None, 8192)              1056768
_________________________________________________________________
reshape (Reshape)            (None, 8, 8, 128)         0
_________________________________________________________________
conv2d_transpose (Conv2DTran (None, 16, 16, 128)       262272
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 16, 16, 128)       0
_________________________________________________________________
conv2d_transpose_1 (Conv2DTr (None, 32, 32, 256)       524544
_________________________________________________________________
leaky_re_lu_4 (LeakyReLU)    (None, 32, 32, 256)       0
_________________________________________________________________
conv2d_transpose_2 (Conv2DTr (None, 64, 64, 512)       2097664
_________________________________________________________________
leaky_re_lu_5 (LeakyReLU)    (None, 64, 64, 512)       0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 64, 64, 3)         38403
=================================================================
Total params: 3,979,651
Trainable params: 3,979,651
Non-trainable params: 0
```

GAN Training Loop Recap

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 `generator` using these random vectors, with targets that all say "these are real images." This updates the weights of the generator to move them toward getting the discriminator to predict "these are real images" for generated images: this trains the generator to fool the discriminator.

GAN Class

```
import tensorflow as tf
class GAN(keras.Model):
    def __init__(self, discriminator, generator, latent_dim):
        super().__init__()
        self.discriminator = discriminator
        self.generator = generator
        self.latent_dim = latent_dim
        self.d_loss_metric = keras.metrics.Mean(name="d_loss")
        self.g_loss_metric = keras.metrics.Mean(name="g_loss")

    def compile(self, d_optimizer, g_optimizer, loss_fn):
        super(GAN, self).compile()
        self.d_optimizer = d_optimizer
        self.g_optimizer = g_optimizer
        self.loss_fn = loss_fn

    @property
    def metrics(self):
        return [self.d_loss_metric, self.g_loss_metric]
```

Sets up metrics to track the two losses over each training epoch

GAN Class: Train Step

```
def train_step(self, real_images):
    batch_size = tf.shape(real_images)[0]
    random_latent_vectors = tf.random.normal(
        shape=(batch_size, self.latent_dim))
generated_images = self.generator(random_latent_vectors)
combined_images = tf.concat([generated_images, real_images], axis=0)
labels = tf.concat(
    [tf.ones((batch_size, 1)), tf.zeros((batch_size, 1))],
    axis=0
)
labels += 0.05 * tf.random.uniform(tf.shape(labels))

with tf.GradientTape() as tape:
    predictions = self.discriminator(combined_images)
    d_loss = self.loss_fn(labels, predictions)
grads = tape.gradient(d_loss, self.discriminator.trainable_weights)
self.d_optimizer.apply_gradients(
    zip(grads, self.discriminator.trainable_weights)
)

random_latent_vectors = tf.random.normal(
    shape=(batch_size, self.latent_dim))

misleading_labels = tf.zeros((batch_size, 1))

with tf.GradientTape() as tape:
    predictions = self.discriminator(
        self.generator(random_latent_vectors))
    g_loss = self.loss_fn(misleading_labels, predictions)
grads = tape.gradient(g_loss, self.generator.trainable_weights)
self.g_optimizer.apply_gradients(
    zip(grads, self.generator.trainable_weights))

self.d_loss_metric.update_state(d_loss)
self.g_loss_metric.update_state(g_loss)
return {"d_loss": self.d_loss_metric.result(),
        "g_loss": self.g_loss_metric.result()}
```

Decodes them to fake images

Combines them with real images

Samples random points in the latent space

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

GAN Monitor Callback

```
class GANMonitor(keras.callbacks.Callback):
    def __init__(self, num_img=3, latent_dim=128):
        self.num_img = num_img
        self.latent_dim = latent_dim

    def on_epoch_end(self, epoch, logs=None):
        random_latent_vectors = tf.random.normal(
            shape=(self.num_img, self.latent_dim))
        generated_images = self.model.generator(random_latent_vectors)
        generated_images *= 255
        generated_images.numpy()
        for i in range(self.num_img):
            img = keras.utils.array_to_img(generated_images[i])
            img.save(f"generated_img_{epoch:03d}_{i}.png")
```

Configuring the GAN and Calling .fit()

```
epochs = 100

gan = GAN(discriminator=discriminator, generator=generator,
          latent_dim=latent_dim)
gan.compile(
    d_optimizer=keras.optimizers.Adam(learning_rate=0.0001),
    g_optimizer=keras.optimizers.Adam(learning_rate=0.0001),
    loss_fn=keras.losses.BinaryCrossentropy(),
)

gan.fit(
    dataset, epochs=epochs,
    callbacks=[GANMonitor(num_img=10, latent_dim=latent_dim)]
)
```

You'll start getting interesting results after epoch 20.

Example GAN Outputs After 30 Epochs

Wrapping Up GANs

- 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.

Summary

- You can use a sequence-to-sequence model to generate sequence data, one step at a time. This is applicable to text generation, but also to note-by-note music generation or any other type of timeseries data.
- DeepDream works by maximizing convnet layer activations through gradient ascent in input space.
- In the style-transfer algorithm, a content image and a style image are combined together via gradient descent to produce an image with the high-level features of the content image and the local characteristics of the style image.
- VAEs and GANs are models that learn a latent space of images and can then dream up entirely new images by sampling from the latent space. *Concept vectors* in the latent space can even be used for image editing.

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.