Prof. Kuan-Ting Lai

2021/5/16

DeepFake (Intro)

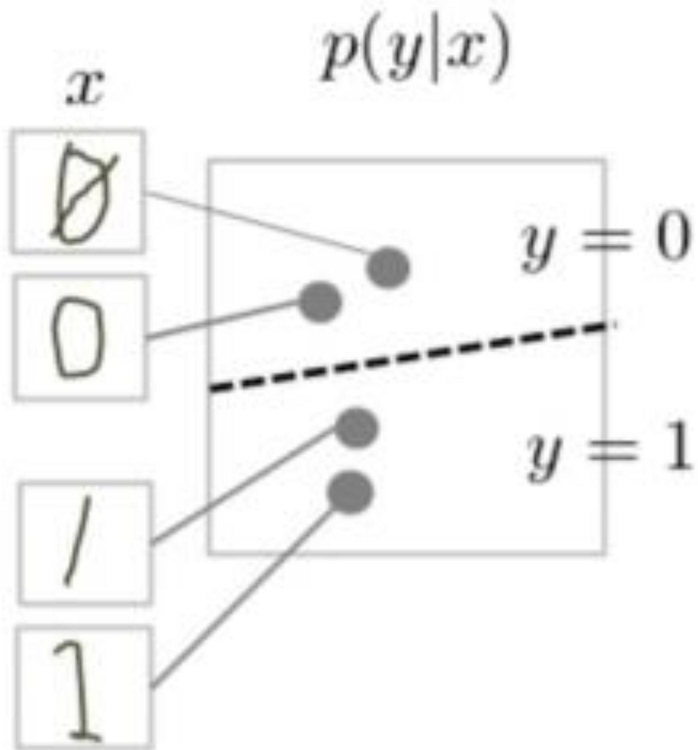
This Person does not Exist (thispersondoesnotexist.com)

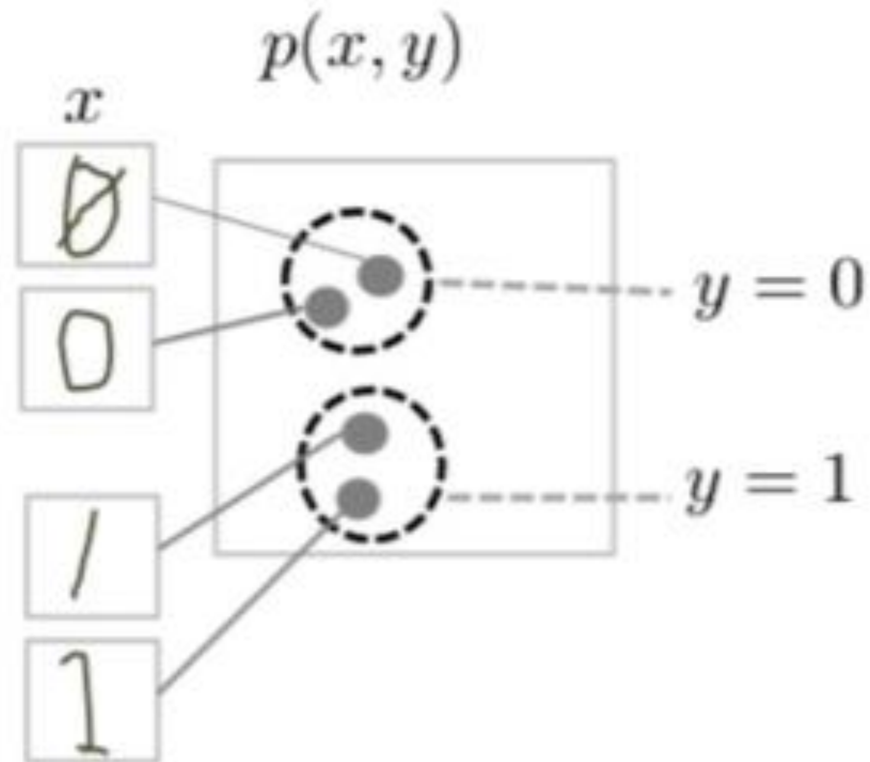
What is a Generative Model?

- Informally:
  - **Generative** models can generate new data instances.
  - **Discriminative** models discriminate between different kinds of data instances.
- Formally
  - **Generative** models capture the joint probability p(X, Y), or just p(X) if there are no labels.
  - **Discriminative** models capture the conditional probability p(Y | X).

https://developers.google.com/machine-learning/gan

Generative Models Are Hard



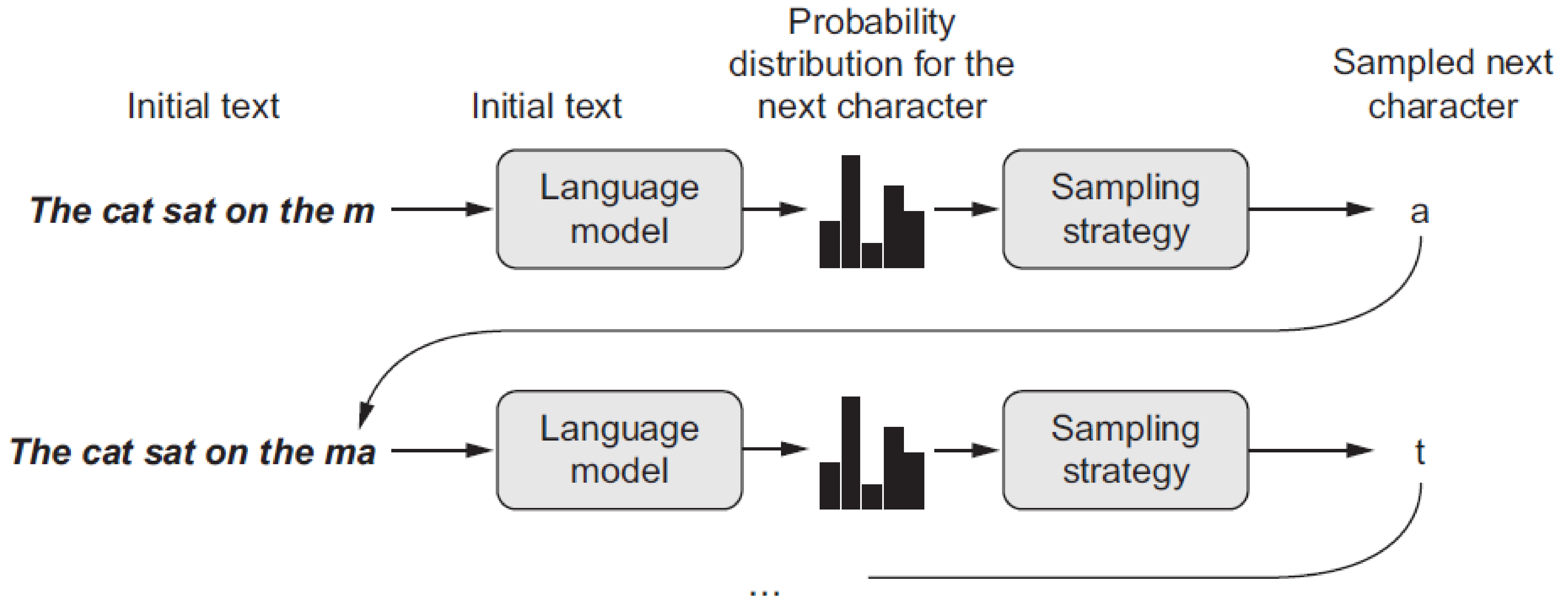


https://developers.google.com/machine-learning/gan/generative

Generative Recurrent Networks

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

Text Generation with LSTM



Sampling Strategy

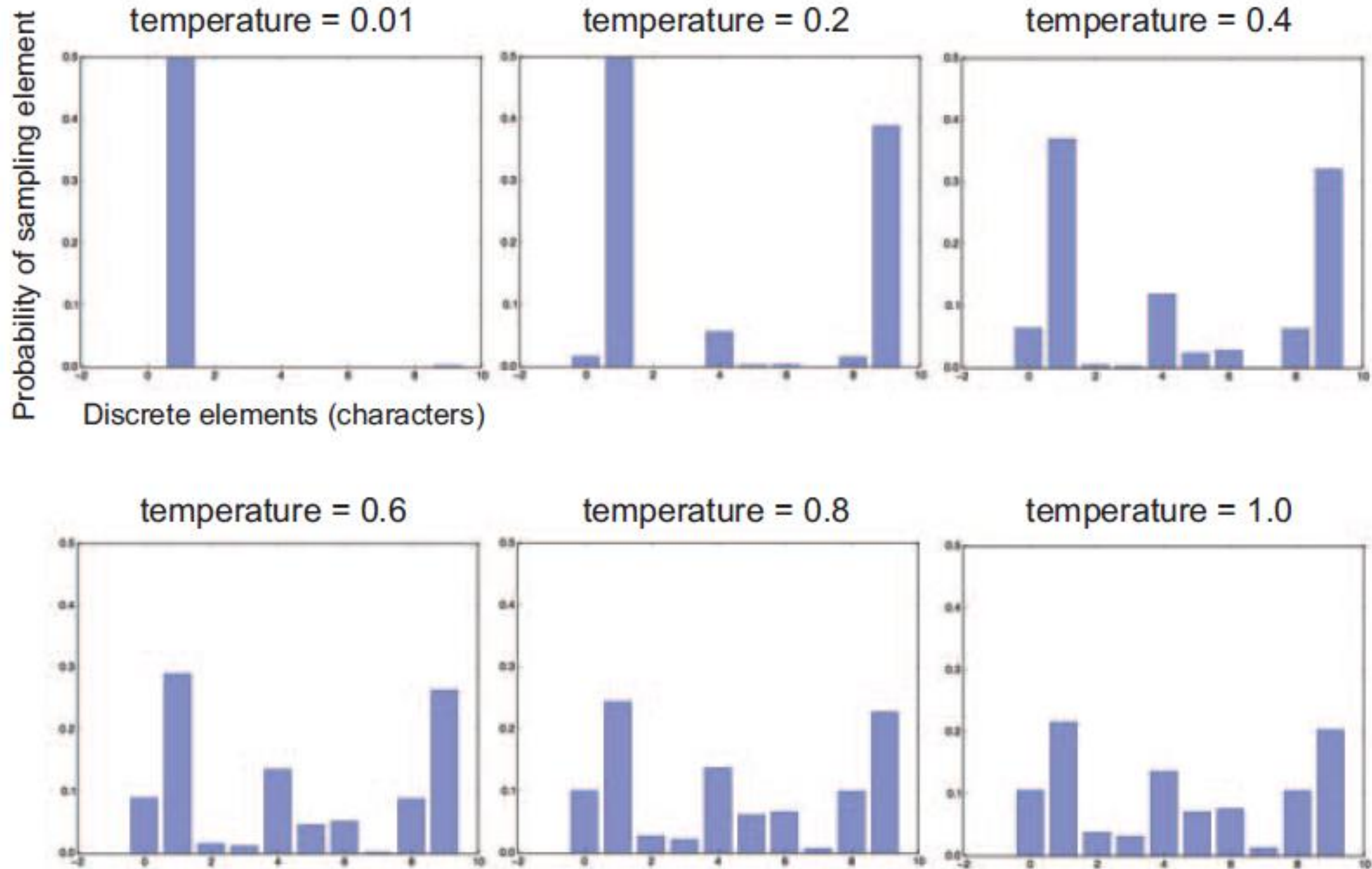
- Greedy sampling: select the one with highest possibility
- Stochastic sampling
- More randomness -> more surprises

Temperature

- Reweighting a probability distribution

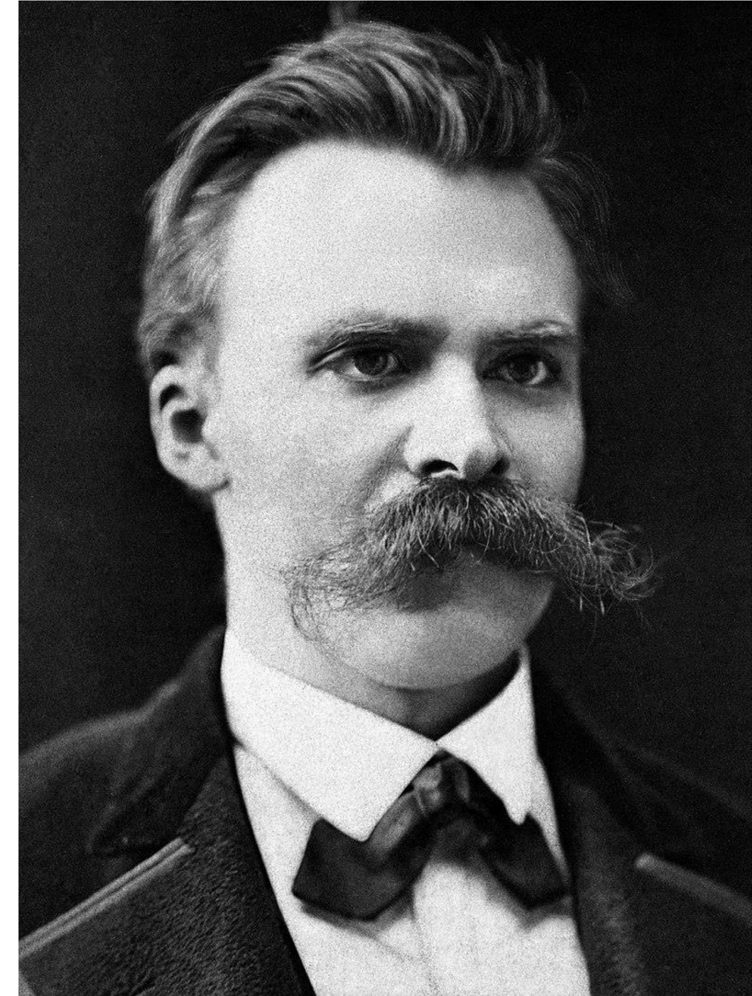
```
import numpy as np
def reweight_distribution(original_distribution, temperature=0.5):
    distribution = np.log(original_distribution) / temperature
    distribution = np.exp(distribution)
    return distribution / np.sum(distribution)
```

Higher Temperature = More Randomness



Generating Text of Nietzsche

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



Character-level LSTM Text Generation

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

```
import keras
import numpy as np

path = keras.utils.get_file(
    'nietzsche.txt',
    origin='https://s3.amazonaws.com/text-datasets/nietzsche.txt')
text = open(path, encoding="utf-8").read().lower()
print('Corpus length:', len(text))
```

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

Convert Characters into Indices

- 57 unique characters in the data

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

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

Vectorizing Sequences of Characters

```
maxlen = 60                      <- You'll extract sequences of 60 characters.

step = 3                         <- You'll sample a new sequence every three characters.

sentences = []                   <- Holds the extracted sequences

next_chars = []                  <- Holds the targets (the follow-up characters)

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)))  <- List of unique characters in the corpus
print('Unique characters:', len(chars))
char_indices = dict((char, chars.index(char)) for char in chars)  <- Dictionary that maps unique characters to their index in the list "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
```

One-hot encodes the characters into binary arrays

Building the Network

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

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

Training & Sampling the Language Model

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

Sampling Next Characters

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

Text-generation Loop

```
import random
import sys

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

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

Trains the model for 60 epochs

Fits the model for one iteration on the data

Selects a text seed at random

Tries a range of different sampling temperatures

Text-generation Loop (Cont'd)

Generates 400 characters, starting from the seed text

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

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

    generated_text += next_char
    generated_text = generated_text[1:]

    sys.stdout.write(next_char)
```

One-hot encodes the characters generated so far

Samples the next character

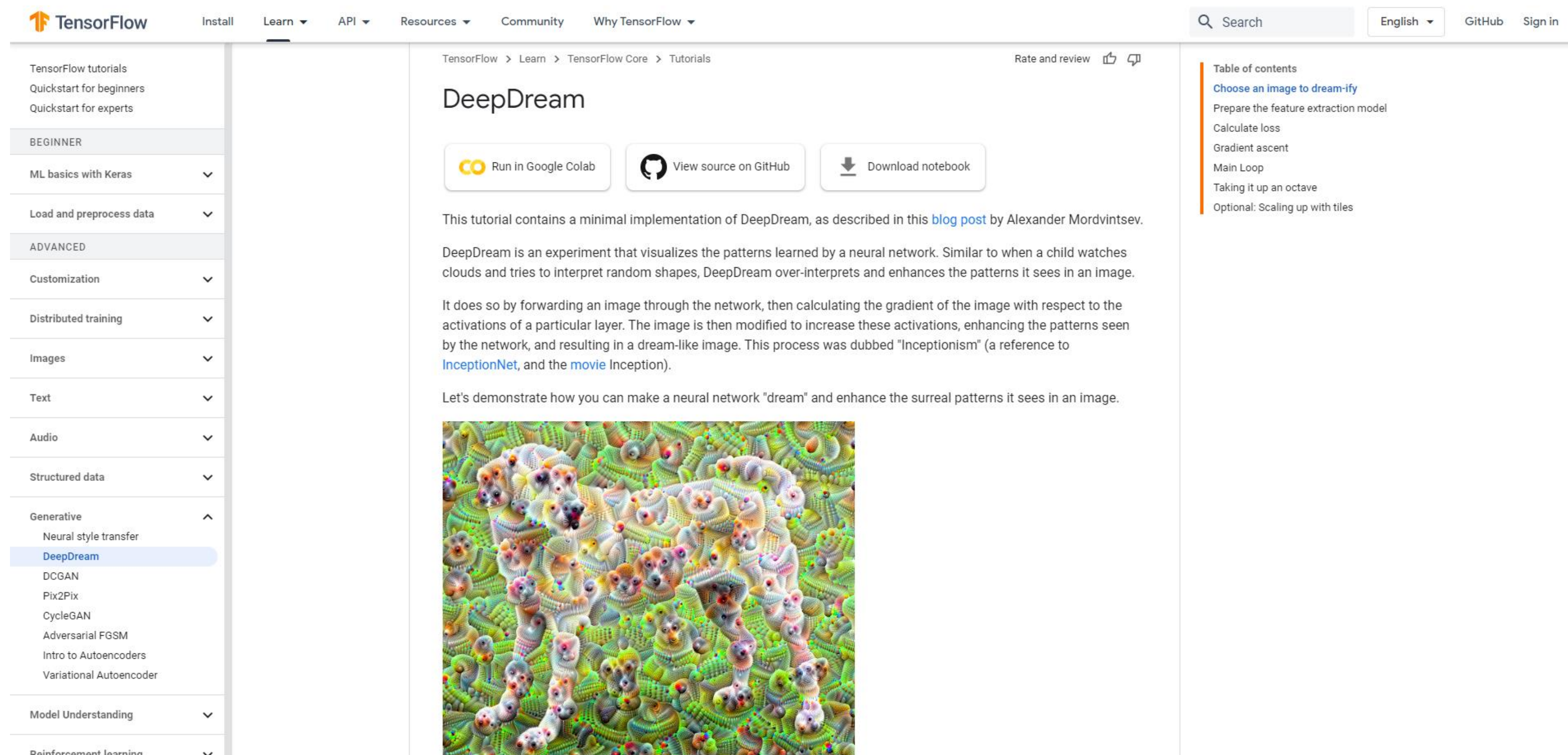
Results of Epoch 60

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

Deep Dream

Use TensorFlow DeepDream Example

- The original keras code are not executable for TensorFlow 2
- https://www.tensorflow.org/tutorials/generative/deepdream



Implementing DeepDream in Keras

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

K.set_learning_phase(0)

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

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

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

Configuring DeepDream

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

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

Define the Loss

- Loss = the weighted sum of the square of the layer activations

```
# Get the symbolic outputs of each "key" layer (we gave them unique names).
layer_dict = dict([(layer.name, layer) for layer in model.layers])

# Define the loss.
loss = K.variable(0.)

for layer_name in layer_contributions:
    # Add the L2 norm of the features of a layer to the loss.
    coeff = layer_contributions[layer_name]
    activation = layer_dict[layer_name].output

    # We avoid border artifacts by only involving non-border pixels in the loss.
    scaling = K.prod(K.cast(K.shape(activation), 'float32'))
    loss.assign_add( coeff * K.sum(K.square(activation[:, 2: -2, 2: -2, :])) / scaling )
    # Note assign_add() is +=. In Keras 2.3.1, Variable += value not supported.
```

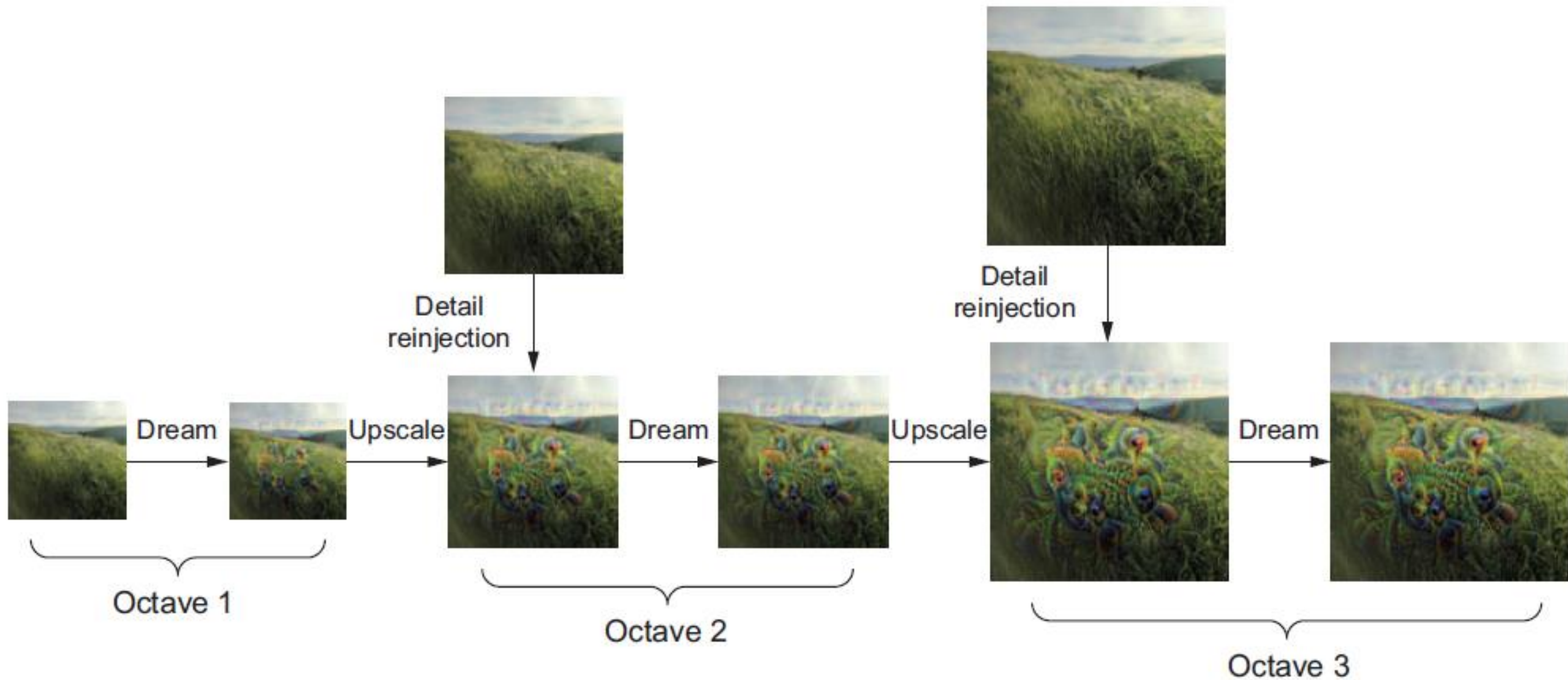
Create Gradients

```
# This holds our generated image
dream = model.input

# Compute the gradients of the dream with regard to the loss.
grads = K.gradients(loss, dream)[0]

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

# Set up function to retrieve the value
# of the loss and gradients given an input image.
outputs = [loss, grads]
fetch_loss_and_grads = K.function([dream], outputs)
```

Gradient-ascent Process

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

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

DeepDream Process: Scaling and Detail Reinjection



Running Gradient Ascent over Different Successive Scales

Playing with these hyperparameters will let you achieve new effects.

```
import numpy as np

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

max_loss = 10.

base_image_path = '...'

img = preprocess_image(base_image_path)
```

- `step = 0.01` — Gradient ascent step size
- `num_octave = 3` — Number of scales at which to run gradient ascent
- `octave_scale = 1.4` — Size ratio between scales
- `iterations = 20` — Number of ascent steps to run at each scale
- `max_loss = 10.` — If the loss grows larger than 10, you'll interrupt the gradient-ascent process to avoid ugly artifacts.
- `base_image_path = '...'` — Fill this with the path to the image you want to use.
- `img = preprocess_image(base_image_path)` — Loads the base image into a Numpy array (function is defined in listing 8.13)

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

successive_shapes = successive_shapes[::-1]

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

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

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

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

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

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

Resizes the Numpy array of the image to the smallest scale

Scales up the dream image

Runs gradient ascent, altering the dream

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

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

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

Reinjects lost detail into the dream

Neural Style Transfer

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

Content target

+

Style reference

=

Combination image

Prisma Photo Editor 12+

Art Filters & Photo Effects

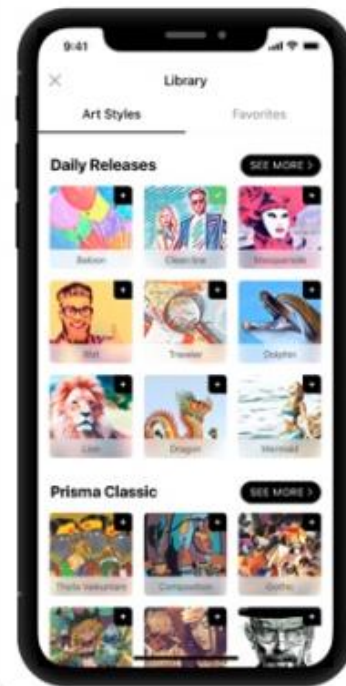
Prisma labs, inc.

★★★★★ 4.7, 95.5K Ratings

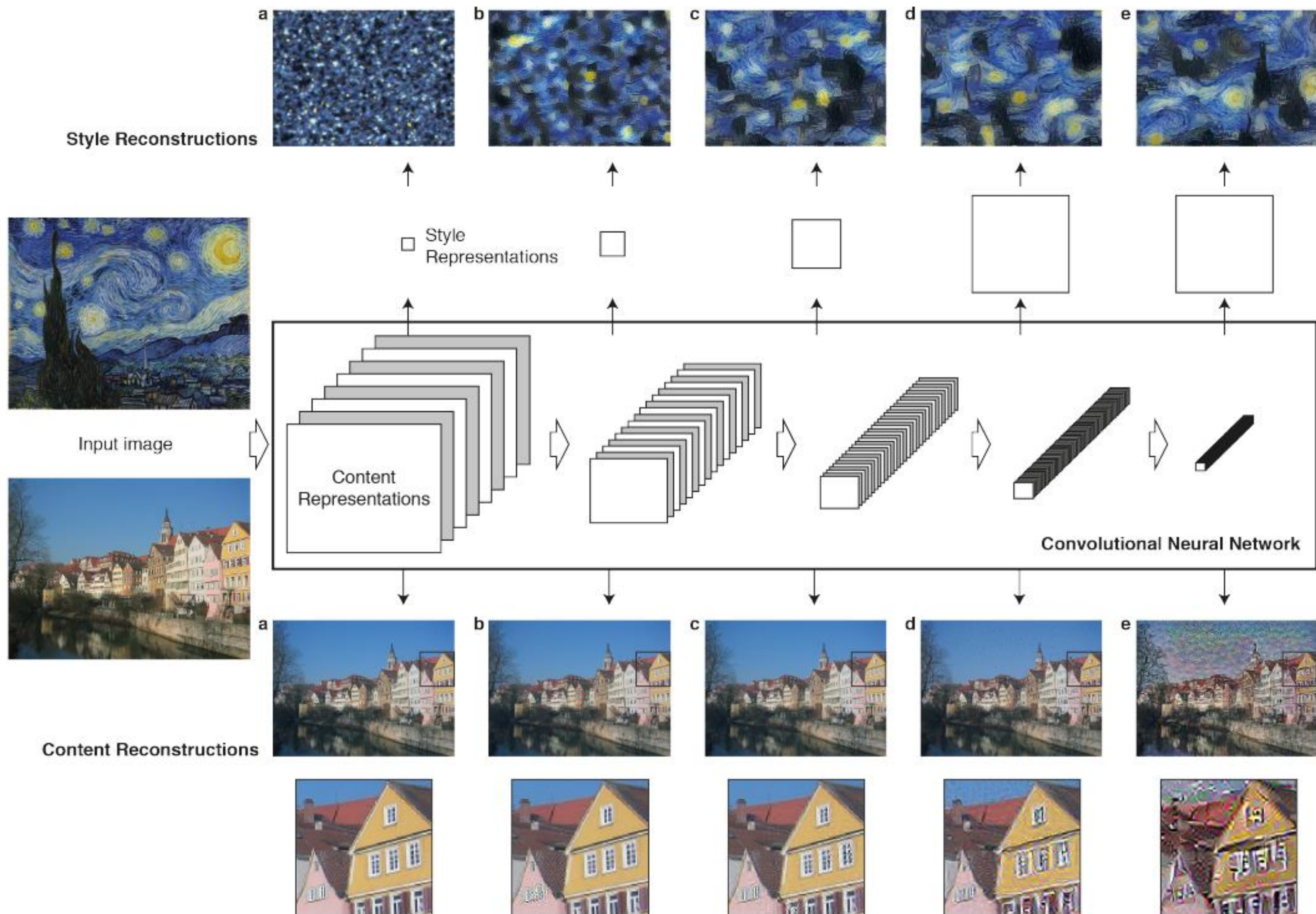
Free · Offers In-App Purchases

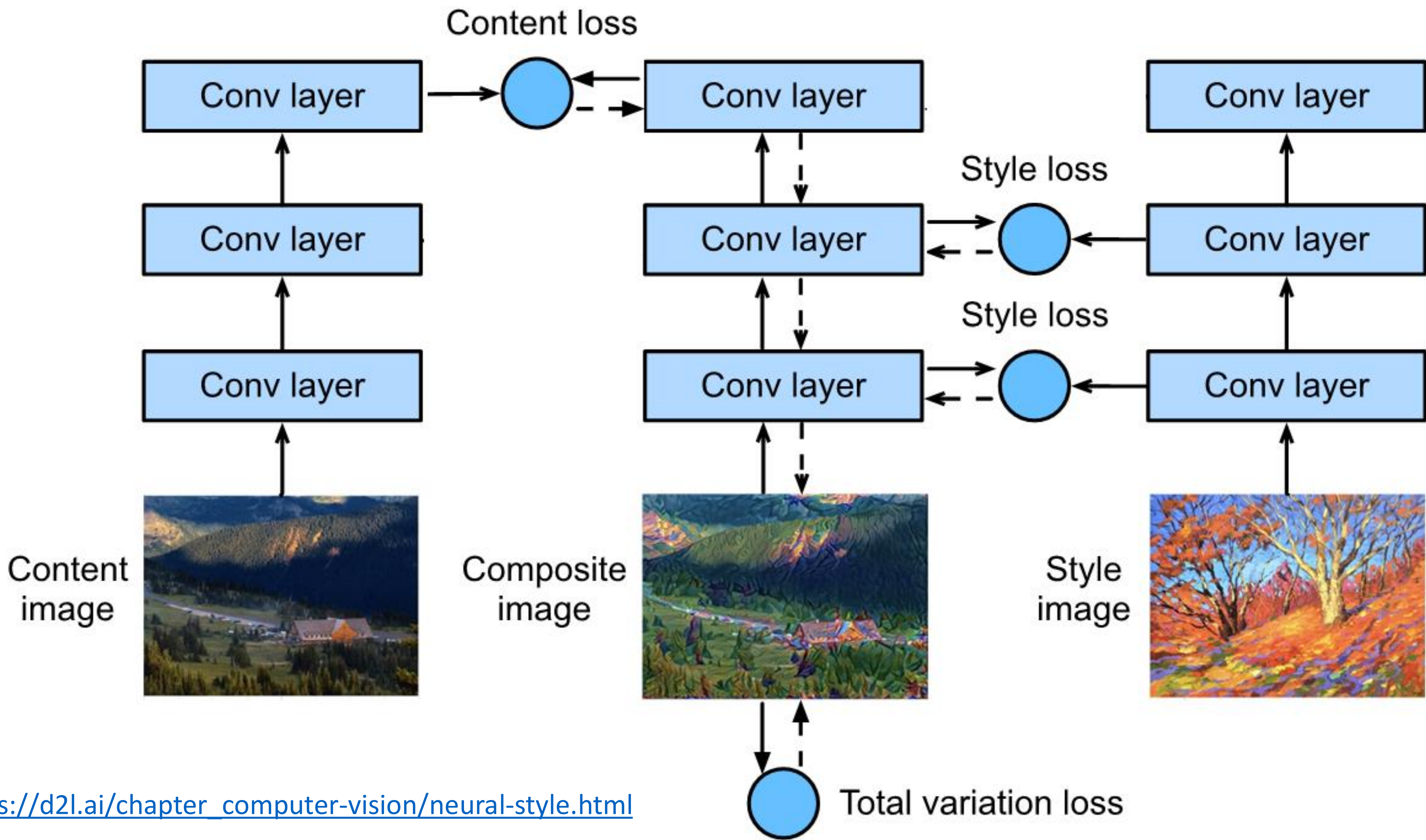
Screenshots

iPhone iPad



- ‘conv1 1’ (a), ‘conv2 1’ (b), ‘conv3 1’ (c), ‘conv4 1’ (d) ‘conv5 1’ (e) of the VGG16-network





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

Content Loss + Style Loss

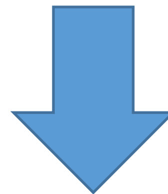
- Using pre-trained model (VGG)
- Content Loss

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

P is output of original image
F is output of random noise

- The style representations are the correlations between different convolution layers
- Correlation is calculated by Gram matrix

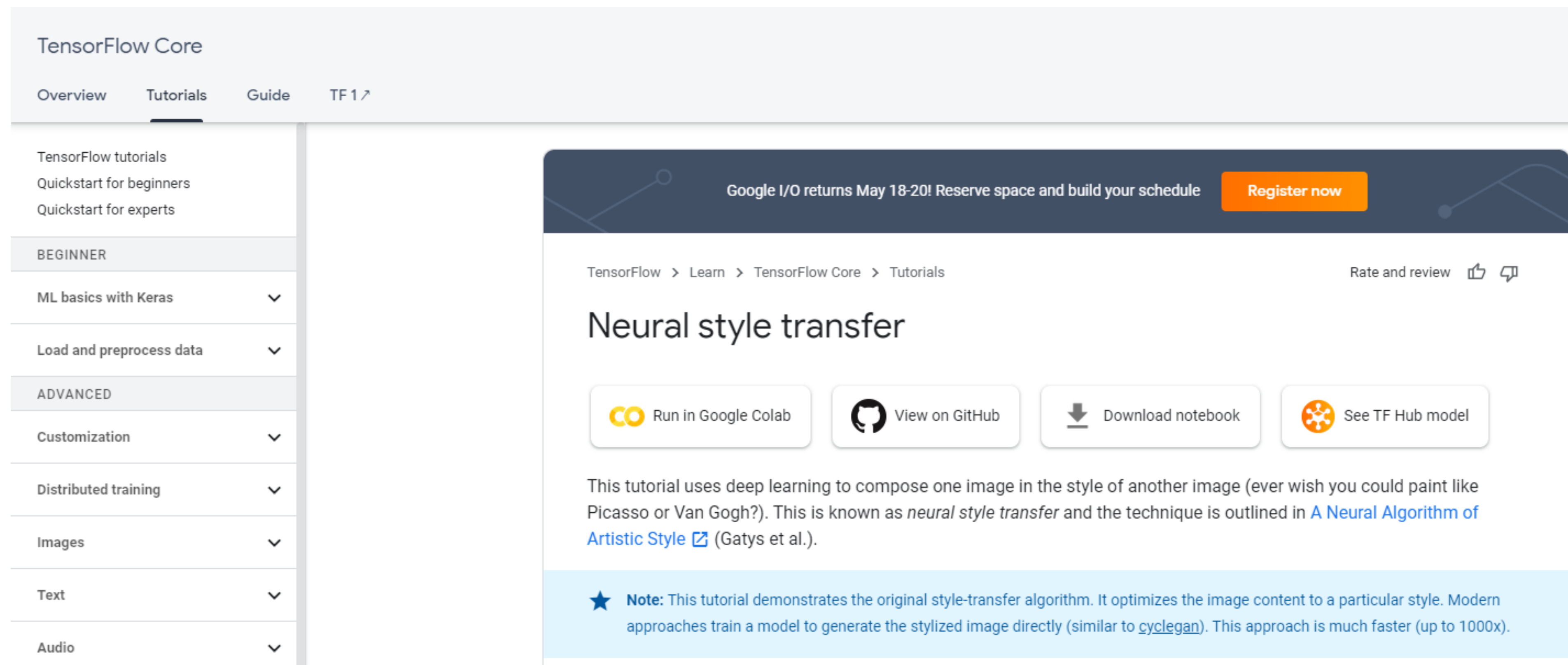
$$E_l = \frac{1}{4N_l^2 M_l^2} \sum_{i,j} (G_{ij}^l - A_{ij}^l)^2 \qquad \mathcal{L}_{style}(\vec{a}, \vec{x}) = \sum_{l=0}^{L} w_l E_l$$



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

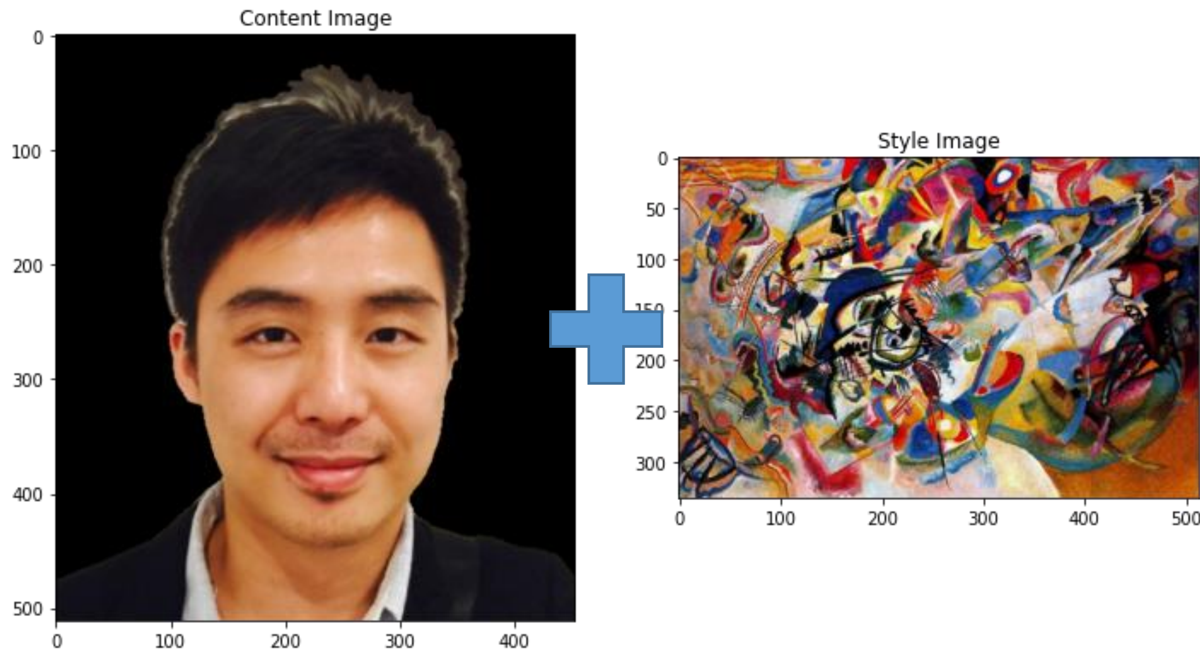
Style Transfer in TensorFlow Tutorial

- https://www.tensorflow.org/tutorials/generative/style_transfer
- Note that modern approaches train a model to generate the stylized image directly (like CycleGAN) and are much faster (up to 1000x)



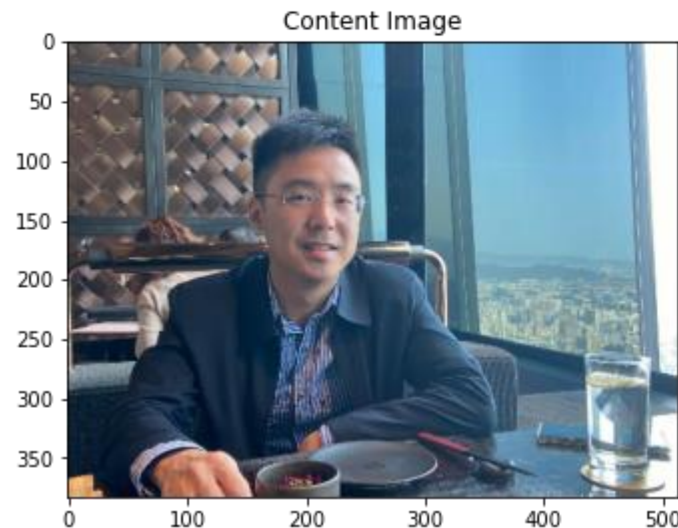
Fast Style Transfer using TF-Hub

```
import tensorflow_hub as hub
hub_model = hub.load('https://tfhub.dev/google/magenta/arbitrary-image-stylization-v1-256/2')
stylized_image = hub_model(tf.constant(content_image), tf.constant(style_image))[0]
tensor_to_image(stylized_image)
```



Get the Style and Content Images

```
style_path = tf.keras.utils.get_file('kandinsky5.jpg','https://storage.googleapis.com/download.tensorflow.org/example_images/Vassily_Kandinsky%2C_1913_-_Composition_7.jpg')

content_image = load_img('./img/kt_photo2.jpg')
style_image = load_img(style_path)

plt.subplot(1, 2, 1)
imshow(content_image, 'Content Image')
plt.subplot(1, 2, 2)
imshow(style_image, 'Style Image')
```



Get Pre-trained VGG19

- load a VGG19 without the classification head, and list the layer names

```
vgg = tf.keras.applications.VGG19(include_top=False,
 weights='imagenet’)

vgg.summary()
```

```
Model: "vgg19"
_________________________________________________________________
Layer (type) Output Shape Param #
=================================================================
input_2 (InputLayer) [(None, None, None, 3)] 0
_________________________________________________________________
block1_conv1 (Conv2D) (None, None, None, 64) 1792
_________________________________________________________________
block1_conv2 (Conv2D) (None, None, None, 64) 36928
_________________________________________________________________
block1_pool (MaxPooling2D) (None, None, None, 64) 0
_________________________________________________________________
block2_conv1 (Conv2D) (None, None, None, 128) 73856
_________________________________________________________________
block2_conv2 (Conv2D) (None, None, None, 128) 147584
_________________________________________________________________
block2_pool (MaxPooling2D) (None, None, None, 128) 0
_________________________________________________________________
block3_conv1 (Conv2D) (None, None, None, 256) 295168
_________________________________________________________________
block3_conv2 (Conv2D) (None, None, None, 256) 590080
_________________________________________________________________
block3_conv3 (Conv2D) (None, None, None, 256) 590080
_________________________________________________________________
block3_conv4 (Conv2D) (None, None, None, 256) 590080
_________________________________________________________________
block3_pool (MaxPooling2D) (None, None, None, 256) 0
_________________________________________________________________
block4_conv1 (Conv2D) (None, None, None, 512) 1180160
_________________________________________________________________
block4_conv2 (Conv2D) (None, None, None, 512) 2359808
_________________________________________________________________
block4_conv3 (Conv2D) (None, None, None, 512) 2359808
_________________________________________________________________
block4_conv4 (Conv2D) (None, None, None, 512) 2359808
_________________________________________________________________
block4_pool (MaxPooling2D) (None, None, None, 512) 0
_________________________________________________________________
block5_conv1 (Conv2D) (None, None, None, 512) 2359808
_________________________________________________________________
block5_conv2 (Conv2D) (None, None, None, 512) 2359808
_________________________________________________________________
block5_conv3 (Conv2D) (None, None, None, 512) 2359808
_________________________________________________________________
block5_conv4 (Conv2D) (None, None, None, 512) 2359808
_________________________________________________________________
block5_pool (MaxPooling2D) (None, None, None, 512) 0
=================================================================
```

Select Intermediate Layers

- Choose intermediate layers from the network to represent the style and content of the image

```
content_layers = ['block5_conv2']

style_layers = ['block1_conv1',
                'block2_conv1',
                'block3_conv1',
                'block4_conv1',
                'block5_conv1']

num_content_layers = len(content_layers)
num_style_layers = len(style_layers)
```

Build a Model with Intermediate Layer Outputs

```
def vgg_layers(layer_names):
  """ Creates a vgg model that returns a list of intermediate output values."""
  # Load our model. Load pretrained VGG, trained on imagenet data
  vgg = tf.keras.applications.VGG19(include_top=False, weights='imagenet')
  vgg.trainable = False

  outputs = [vgg.get_layer(name).output for name in layer_names]

  model = tf.keras.Model([vgg.input], outputs)
  return model
```

Process the Style Image

```
style_extractor = vgg_layers(style_layers)
style_outputs = style_extractor(style_image*255)

#Look at the statistics of each layer's output
for name, output in zip(style_layers, style_outputs):
  print(name)
  print("  shape: ", output.numpy().shape)
  print("  min: ", output.numpy().min())
  print("  max: ", output.numpy().max())
  print("  mean: ", output.numpy().mean())
  print()
```

```
block1_conv1 shape: (1, 336, 512, 64) min: 0.0 max: 835.5256 mean: 33.97525
block2_conv1 shape: (1, 168, 256, 128) min: 0.0 max: 4625.8857 mean: 199.82687
block3_conv1 shape: (1, 84, 128, 256) min: 0.0 max: 8789.239 mean: 230.78099
block4_conv1 shape: (1, 42, 64, 512) min: 0.0 max: 21566.135 mean: 791.24005
block5_conv1 shape: (1, 21, 32, 512) min: 0.0 max: 3189.2542 mean: 59.179478
```

Calculate the Style

- Style can be described by the means and correlations across the different feature maps.
- Calculate a Gram matrix by taking the outer product of the feature vector with itself at each location and averaging that outer product over all locations.

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

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

StyleContentModel

- Build a model that returns the style and content tensors.

```
class StyleContentModel(tf.keras.models.Model):
  def __init__(self, style_layers, content_layers):
    super(StyleContentModel, self).__init__()
    self.vgg = vgg_layers(style_layers + content_layers)
    self.style_layers = style_layers
    self.content_layers = content_layers
    self.num_style_layers = len(style_layers)
    self.vgg.trainable = False

  def call(self, inputs):
    …
    …
```

```python
class StyleContentModel(tf.keras.models.Model):
  def __init__(self, style_layers, content_layers):
    …
  def call(self, inputs):
    "Expects float input in [0,1]"
    inputs = inputs*255.0
    preprocessed_input = tf.keras.applications.vgg19.preprocess_input(inputs)
    outputs = self.vgg(preprocessed_input)
    style_outputs, content_outputs = (outputs[:self.num_style_layers],
                                      outputs[self.num_style_layers:])

    style_outputs = [gram_matrix(style_output)
                     for style_output in style_outputs]

    content_dict = {content_name: value
                    for content_name, value
                    in zip(self.content_layers, content_outputs)}

    style_dict = {style_name: value
                  for style_name, value
                  in zip(self.style_layers, style_outputs)}

    return {'content': content_dict, 'style': style_dict}
```

Define Style + Content Loss Function

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

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

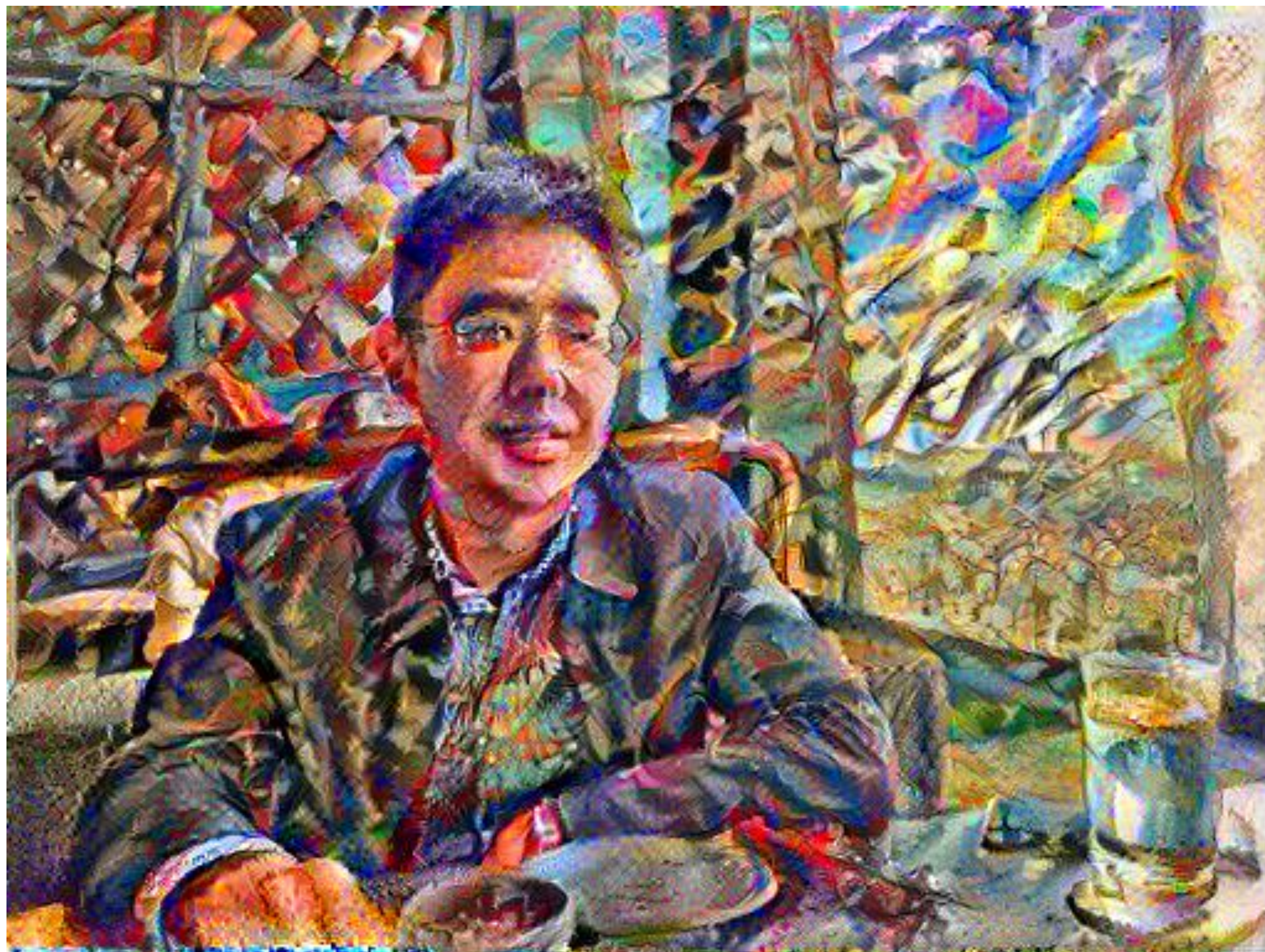
Run gradient descent

```
extractor = StyleContentModel(style_layers, content_layers)
style_targets = extractor(style_image)['style']
content_targets = extractor(content_image)['content']

image = tf.Variable(content_image)
def clip_0_1(image):
  return tf.clip_by_value(image, clip_value_min=0.0, clip_value_max=1.0)

opt = tf.optimizers.Adam(learning_rate=0.02, beta_1=0.99, epsilon=1e-1)

style_weight=1e-2
content_weight=1e4

# Use tf.GradientTape to update the image.
@tf.function()
def train_step(image):
  with tf.GradientTape() as tape:
    outputs = extractor(image)
    loss = style_content_loss(outputs)

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

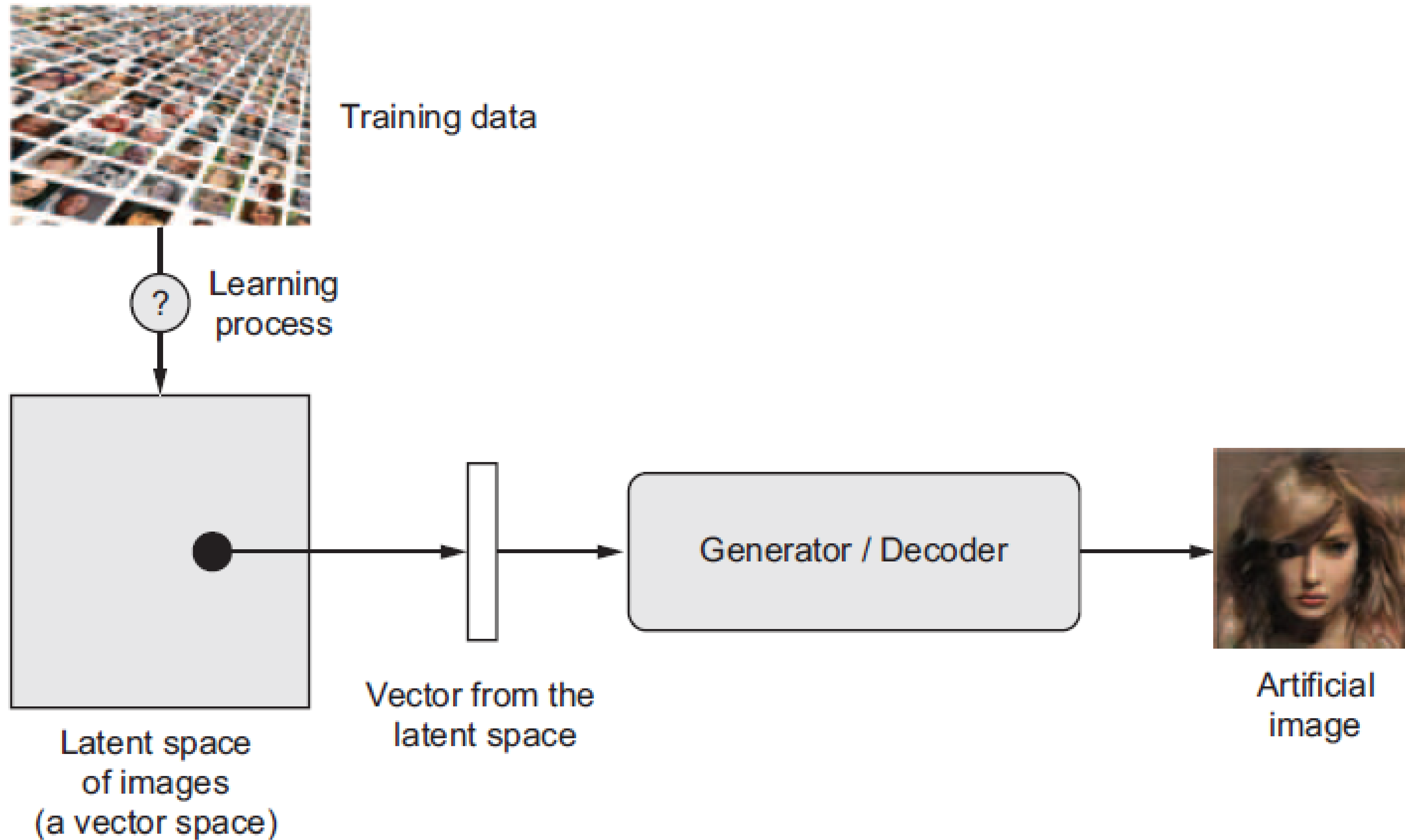
Test Results

```
train_step(image)
train_step(image)
train_step(image)
tensor_to_image(image)
```

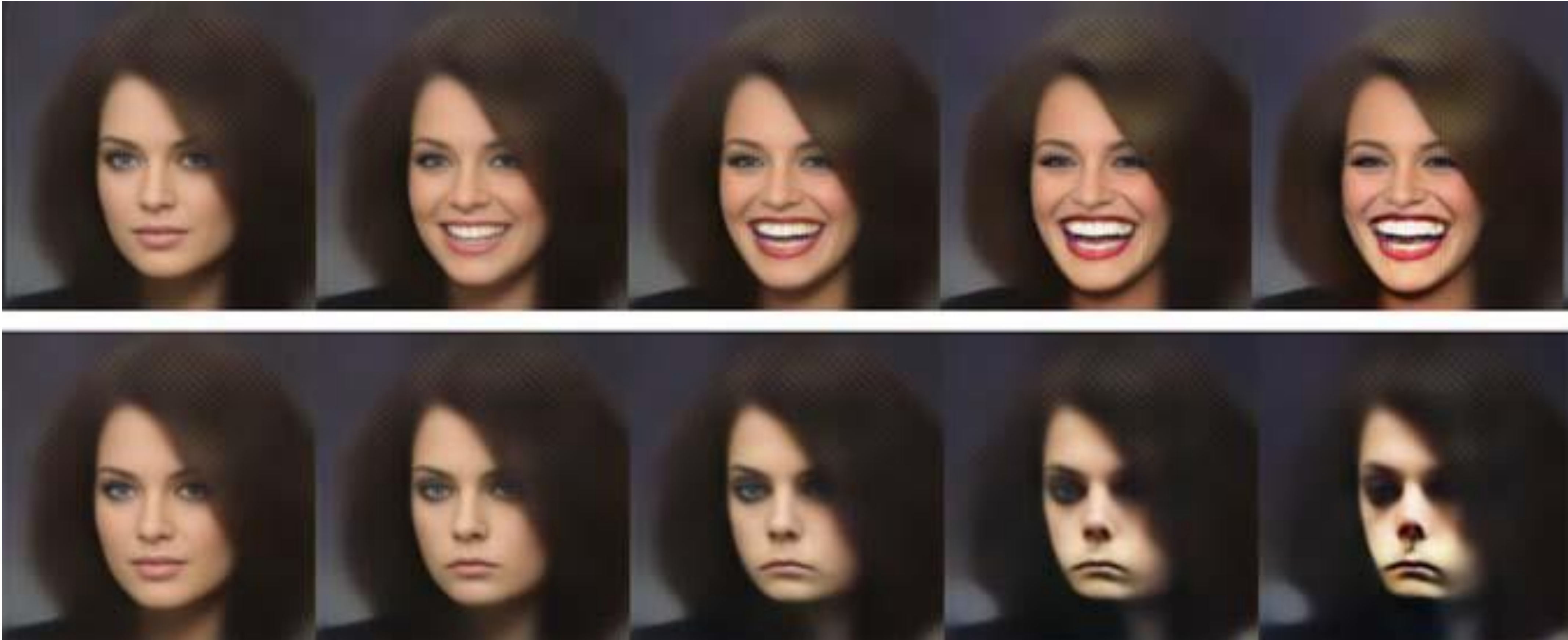
Run 1000 Times



Generating Images with Variational Auto-encoder

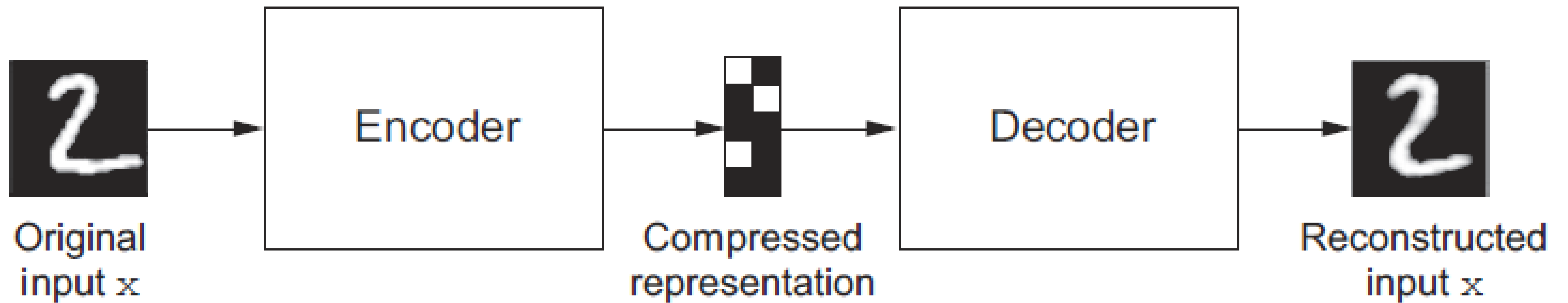


The Smile Vector



Auto-encoder

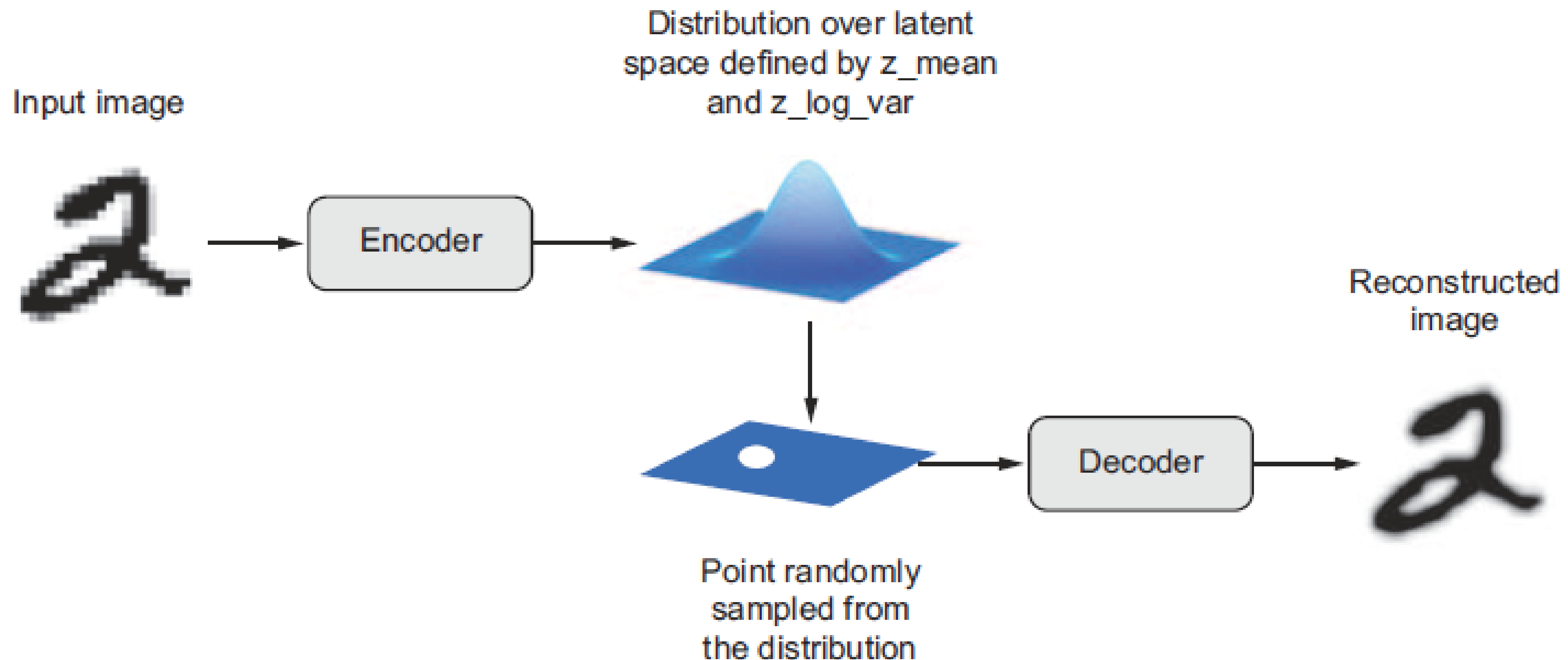
- Learn compressed representation of input x



Variational Auto-encoder

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

[deep-learning-with-python-notebooks/chapter12_part04_variational-autoencoders.ipynb at master · fchollet/deep-learning-with-python-notebooks · GitHub]



Pseudo Code of Encode and Decoder

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

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

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

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

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

Encoder

```
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()
import keras
from keras import layers
from keras import backend as K
from keras.models import Model
import numpy as np

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

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

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

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

Sampling

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

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

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

Decoder

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

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

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

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

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

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

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

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

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

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

Training VAE

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

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

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

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

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

Use Decoder to Turn Latent Vectors into Images

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

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

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

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

0
50
100
150
200
250
300
350
400

0 50 100 150 200 250 300 350 400

References

- Francois Chollet, “Deep Learning with Python,” Chapter 8
- https://www.tensorflow.org/tutorials/generative/
- https://developers.google.com/machine-learning/gan