**Deep Convolutional Generative Adversarial Network**

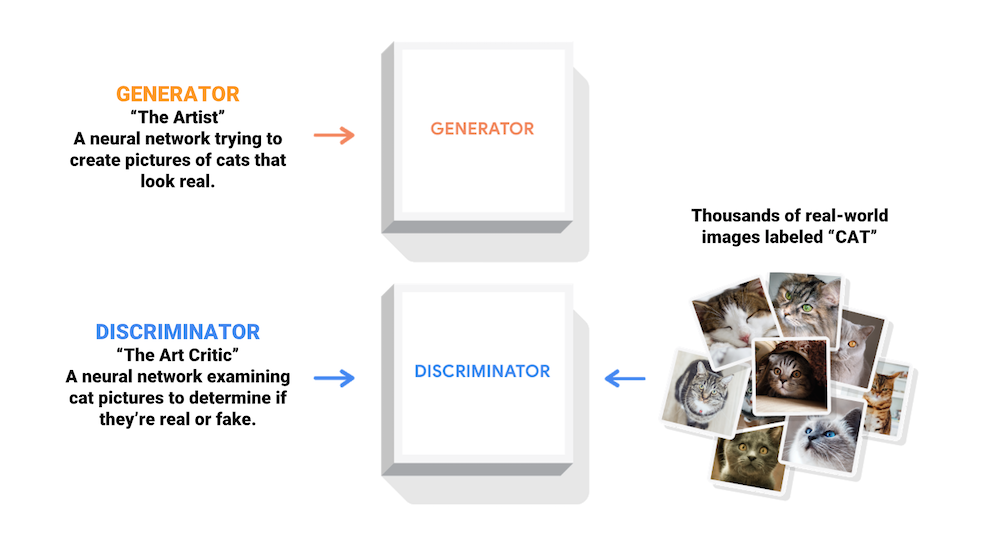
bookmark\_border

|  |  |  |
| --- | --- | --- |
| [Run in Google Colab](https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/tutorials/generative/dcgan.ipynb) | [View source on GitHub](https://github.com/tensorflow/docs/blob/master/site/en/tutorials/generative/dcgan.ipynb) | [Download notebook](https://storage.googleapis.com/tensorflow_docs/docs/site/en/tutorials/generative/dcgan.ipynb) |

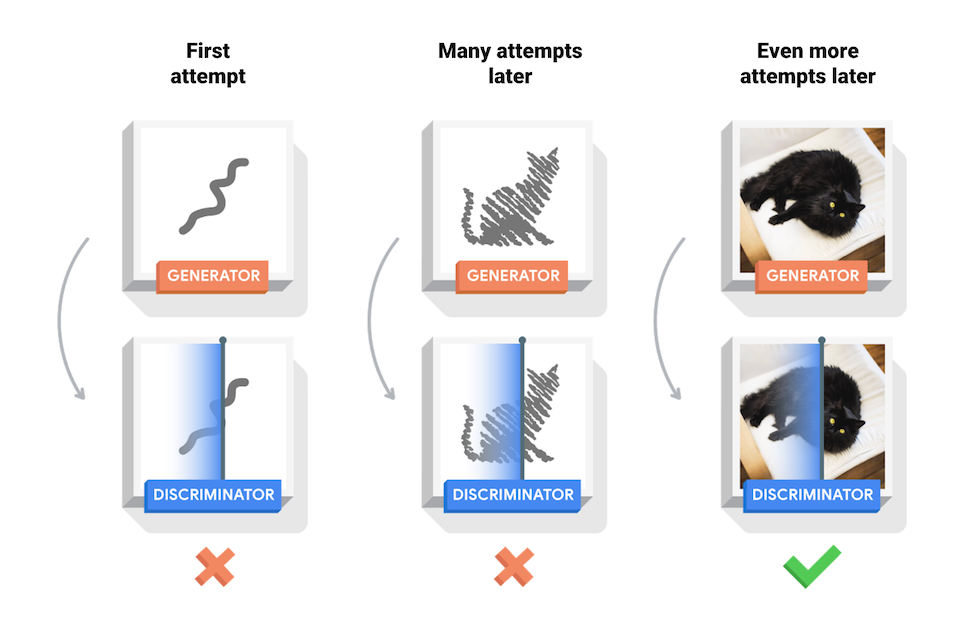
This tutorial demonstrates how to generate images of handwritten digits using a [Deep Convolutional Generative Adversarial Network](https://arxiv.org/pdf/1511.06434.pdf) (DCGAN). The code is written using the [Keras Sequential API](https://www.tensorflow.org/guide/keras) with a [tf.GradientTape](https://www.tensorflow.org/api_docs/python/tf/GradientTape) training loop.

**What are GANs?**

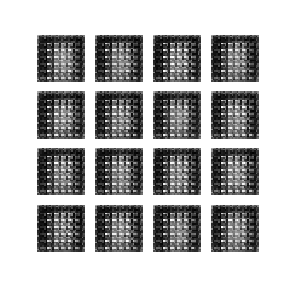
[Generative Adversarial Networks](https://arxiv.org/abs/1406.2661) (GANs) are one of the most interesting ideas in computer science today. Two models are trained simultaneously by an adversarial process. A *generator* ("the artist") learns to create images that look real, while a *discriminator* ("the art critic") learns to tell real images apart from fakes.



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



This notebook demonstrates this process on the MNIST dataset. The following animation shows a series of images produced by the *generator* as it was trained for 50 epochs. The images begin as random noise, and increasingly resemble hand written digits over time.



To learn more about GANs, see MIT's [Intro to Deep Learning](http://introtodeeplearning.com/) course.

**Setup**

import tensorflow as tf

2024-08-16 06:32:45.015549: E external/local\_xla/xla/stream\_executor/cuda/cuda\_fft.cc:485] Unable to register cuFFT factory: Attempting to register factory for plugin cuFFT when one has already been registered

2024-08-16 06:32:45.036284: E external/local\_xla/xla/stream\_executor/cuda/cuda\_dnn.cc:8454] Unable to register cuDNN factory: Attempting to register factory for plugin cuDNN when one has already been registered

2024-08-16 06:32:45.042482: E external/local\_xla/xla/stream\_executor/cuda/cuda\_blas.cc:1452] Unable to register cuBLAS factory: Attempting to register factory for plugin cuBLAS when one has already been registered

tf.\_\_version\_\_

'2.17.0'

# To generate GIFs

pip install imageio

pip install git+https://github.com/tensorflow/docs

import glob

import imageio

import matplotlib.pyplot as plt

import numpy as np

import os

import PIL

from tensorflow.keras import layers

import time

from IPython import display

**Load and prepare the dataset**

You will use the MNIST dataset to train the generator and the discriminator. The generator will generate handwritten digits resembling the MNIST data.

(train\_images, train\_labels), (\_, \_) = tf.keras.datasets.mnist.load\_data()

train\_images = train\_images.reshape(train\_images.shape[0], 28, 28, 1).astype('float32')

train\_images = (train\_images - 127.5) / 127.5 # Normalize the images to [-1, 1]

BUFFER\_SIZE = 60000

BATCH\_SIZE = 256

# Batch and shuffle the data

train\_dataset = tf.data.Dataset.from\_tensor\_slices(train\_images).shuffle(BUFFER\_SIZE).batch(BATCH\_SIZE)

WARNING: All log messages before absl::InitializeLog() is called are written to STDERR

I0000 00:00:1723789973.811300 174689 cuda\_executor.cc:1015] successful NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-pci#L344-L355

I0000 00:00:1723789973.815200 174689 cuda\_executor.cc:1015] successful NUMA node read from SysFS had negative value (-1), but there must be at least one NUMA node, so returning NUMA node zero. See more at https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-pci#L344-L355

**Create the models**

Both the generator and discriminator are defined using the [Keras Sequential API](https://www.tensorflow.org/guide/keras" \l "sequential_model).

**The Generator**

The generator uses [tf.keras.layers.Conv2DTranspose](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2DTranspose) (upsampling) layers to produce an image from a seed (random noise). Start with a Dense layer that takes this seed as input, then upsample several times until you reach the desired image size of 28x28x1. Notice the [tf.keras.layers.LeakyReLU](https://www.tensorflow.org/api_docs/python/tf/keras/layers/LeakyReLU) activation for each layer, except the output layer which uses tanh.

def make\_generator\_model():

model = tf.keras.Sequential()

model.add(layers.Dense(7\*7\*256, use\_bias=False, input\_shape=(100,)))

model.add(layers.BatchNormalization())

model.add(layers.LeakyReLU())

model.add(layers.Reshape((7, 7, 256)))

assert model.output\_shape == (None, 7, 7, 256) # Note: None is the batch size

model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', use\_bias=False))

assert model.output\_shape == (None, 7, 7, 128)

model.add(layers.BatchNormalization())

model.add(layers.LeakyReLU())

model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use\_bias=False))

assert model.output\_shape == (None, 14, 14, 64)

model.add(layers.BatchNormalization())

model.add(layers.LeakyReLU())

model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same', use\_bias=False, activation='tanh'))

assert model.output\_shape == (None, 28, 28, 1)

return model

Use the (as yet untrained) generator to create an image.

generator = make\_generator\_model()

noise = tf.random.normal([1, 100])

generated\_image = generator(noise, training=False)

plt.imshow(generated\_image[0, :, :, 0], cmap='gray')

/tmpfs/src/tf\_docs\_env/lib/python3.9/site-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input\_shape`/`input\_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.

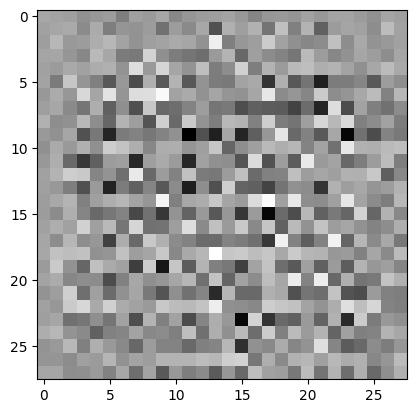
super().\_\_init\_\_(activity\_regularizer=activity\_regularizer, \*\*kwargs)

W0000 00:00:1723789976.899989 174689 gpu\_timer.cc:114] Skipping the delay kernel, measurement accuracy will be reduced

W0000 00:00:1723789976.926511 174689 gpu\_timer.cc:114] Skipping the delay kernel, measurement accuracy will be reduced

W0000 00:00:1723789976.928496 174689 gpu\_timer.cc:114] Skipping the delay kernel, measurement accuracy will be reduced

<matplotlib.image.AxesImage at 0x7f730c2a83a0>



**The Discriminator**

The discriminator is a CNN-based image classifier.

def make\_discriminator\_model():

model = tf.keras.Sequential()

model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',

input\_shape=[28, 28, 1]))

model.add(layers.LeakyReLU())

model.add(layers.Dropout(0.3))

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

model.add(layers.LeakyReLU())

model.add(layers.Dropout(0.3))

model.add(layers.Flatten())

model.add(layers.Dense(1))

return model

Use the (as yet untrained) discriminator to classify the generated images as real or fake. The model will be trained to output positive values for real images, and negative values for fake images.

discriminator = make\_discriminator\_model()

decision = discriminator(generated\_image)

print (decision)

tf.Tensor([[0.00357757]], shape=(1, 1), dtype=float32)

/tmpfs/src/tf\_docs\_env/lib/python3.9/site-packages/keras/src/layers/convolutional/base\_conv.py:107: UserWarning: Do not pass an `input\_shape`/`input\_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.

super().\_\_init\_\_(activity\_regularizer=activity\_regularizer, \*\*kwargs)

W0000 00:00:1723789977.415503 174689 gpu\_timer.cc:114] Skipping the delay kernel, measurement accuracy will be reduced

W0000 00:00:1723789977.422830 174689 gpu\_timer.cc:114] Skipping the delay kernel, measurement accuracy will be reduced

W0000 00:00:1723789977.426039 174689 gpu\_timer.cc:114] Skipping the delay kernel, measurement accuracy will be reduced

**Define the loss and optimizers**

Define loss functions and optimizers for both models.

# This method returns a helper function to compute cross entropy loss

cross\_entropy = tf.keras.losses.BinaryCrossentropy(from\_logits=True)

**Discriminator loss**

This method quantifies how well the discriminator is able to distinguish real images from fakes. It compares the discriminator's predictions on real images to an array of 1s, and the discriminator's predictions on fake (generated) images to an array of 0s.

def discriminator\_loss(real\_output, fake\_output):

real\_loss = cross\_entropy(tf.ones\_like(real\_output), real\_output)

fake\_loss = cross\_entropy(tf.zeros\_like(fake\_output), fake\_output)

total\_loss = real\_loss + fake\_loss

return total\_loss

**Generator loss**

The generator's loss quantifies how well it was able to trick the discriminator. Intuitively, if the generator is performing well, the discriminator will classify the fake images as real (or 1). Here, compare the discriminators decisions on the generated images to an array of 1s.

def generator\_loss(fake\_output):

return cross\_entropy(tf.ones\_like(fake\_output), fake\_output)

The discriminator and the generator optimizers are different since you will train two networks separately.

generator\_optimizer = tf.keras.optimizers.Adam(1e-4)

discriminator\_optimizer = tf.keras.optimizers.Adam(1e-4)

**Save checkpoints**

This notebook also demonstrates how to save and restore models, which can be helpful in case a long running training task is interrupted.

checkpoint\_dir = './training\_checkpoints'

checkpoint\_prefix = os.path.join(checkpoint\_dir, "ckpt")

checkpoint = tf.train.Checkpoint(generator\_optimizer=generator\_optimizer,

discriminator\_optimizer=discriminator\_optimizer,

generator=generator,

discriminator=discriminator)

**Define the training loop**

EPOCHS = 50

noise\_dim = 100

num\_examples\_to\_generate = 16

# You will reuse this seed overtime (so it's easier)

# to visualize progress in the animated GIF)

seed = tf.random.normal([num\_examples\_to\_generate, noise\_dim])

The training loop begins with generator receiving a random seed as input. That seed is used to produce an image. The discriminator is then used to classify real images (drawn from the training set) and fakes images (produced by the generator). The loss is calculated for each of these models, and the gradients are used to update the generator and discriminator.

# Notice the use of `tf.function`

# This annotation causes the function to be "compiled".

@tf.function

def train\_step(images):

noise = tf.random.normal([BATCH\_SIZE, noise\_dim])

with tf.GradientTape() as gen\_tape, tf.GradientTape() as disc\_tape:

generated\_images = generator(noise, training=True)

real\_output = discriminator(images, training=True)

fake\_output = discriminator(generated\_images, training=True)

gen\_loss = generator\_loss(fake\_output)

disc\_loss = discriminator\_loss(real\_output, fake\_output)

gradients\_of\_generator = gen\_tape.gradient(gen\_loss, generator.trainable\_variables)

gradients\_of\_discriminator = disc\_tape.gradient(disc\_loss, discriminator.trainable\_variables)

generator\_optimizer.apply\_gradients(zip(gradients\_of\_generator, generator.trainable\_variables))

discriminator\_optimizer.apply\_gradients(zip(gradients\_of\_discriminator, discriminator.trainable\_variables))

def train(dataset, epochs):

for epoch in range(epochs):

start = time.time()

for image\_batch in dataset:

train\_step(image\_batch)

# Produce images for the GIF as you go

display.clear\_output(wait=True)

generate\_and\_save\_images(generator,

epoch + 1,

seed)

# Save the model every 15 epochs

if (epoch + 1) % 15 == 0:

checkpoint.save(file\_prefix = checkpoint\_prefix)

print ('Time for epoch {} is {} sec'.format(epoch + 1, time.time()-start))

# Generate after the final epoch

display.clear\_output(wait=True)

generate\_and\_save\_images(generator,

epochs,

seed)

**Generate and save images**

def generate\_and\_save\_images(model, epoch, test\_input):

# Notice `training` is set to False.

# This is so all layers run in inference mode (batchnorm).

predictions = model(test\_input, training=False)

fig = plt.figure(figsize=(4, 4))

for i in range(predictions.shape[0]):

plt.subplot(4, 4, i+1)

plt.imshow(predictions[i, :, :, 0] \* 127.5 + 127.5, cmap='gray')

plt.axis('off')

plt.savefig('image\_at\_epoch\_{:04d}.png'.format(epoch))

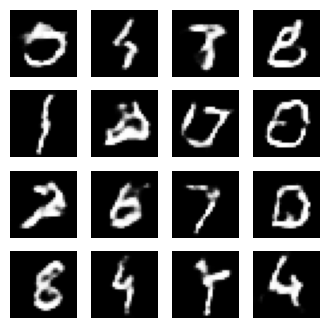
plt.show()

**Train the model**

Call the train() method defined above to train the generator and discriminator simultaneously. Note, training GANs can be tricky. It's important that the generator and discriminator do not overpower each other (e.g., that they train at a similar rate).

At the beginning of the training, the generated images look like random noise. As training progresses, the generated digits will look increasingly real. After about 50 epochs, they resemble MNIST digits. This may take about one minute / epoch with the default settings on Colab.

train(train\_dataset, EPOCHS)



Restore the latest checkpoint.

checkpoint.restore(tf.train.latest\_checkpoint(checkpoint\_dir))

<tensorflow.python.checkpoint.checkpoint.CheckpointLoadStatus at 0x7f730c28dd30>

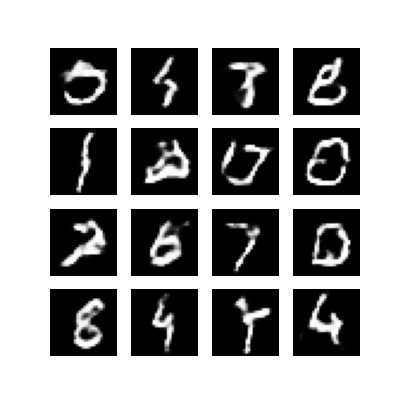
**Create a GIF**

# Display a single image using the epoch number

def display\_image(epoch\_no):

return PIL.Image.open('image\_at\_epoch\_{:04d}.png'.format(epoch\_no))

display\_image(EPOCHS)



Use imageio to create an animated gif using the images saved during training.

anim\_file = 'dcgan.gif'

with imageio.get\_writer(anim\_file, mode='I') as writer:

filenames = glob.glob('image\*.png')

filenames = sorted(filenames)

for filename in filenames:

image = imageio.imread(filename)

writer.append\_data(image)

image = imageio.imread(filename)

writer.append\_data(image)

/tmpfs/tmp/ipykernel\_174689/1982054950.py:7: DeprecationWarning: Starting with ImageIO v3 the behavior of this function will switch to that of iio.v3.imread. To keep the current behavior (and make this warning disappear) use `import imageio.v2 as imageio` or call `imageio.v2.imread` directly.

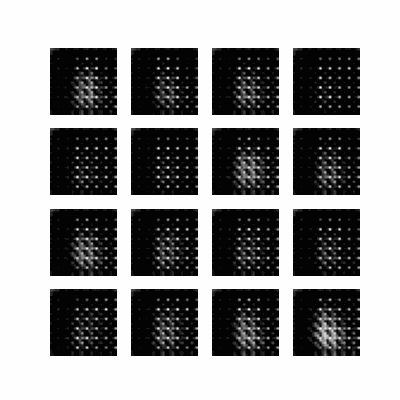
image = imageio.imread(filename)

/tmpfs/tmp/ipykernel\_174689/1982054950.py:9: DeprecationWarning: Starting with ImageIO v3 the behavior of this function will switch to that of iio.v3.imread. To keep the current behavior (and make this warning disappear) use `import imageio.v2 as imageio` or call `imageio.v2.imread` directly.

image = imageio.imread(filename)

import tensorflow\_docs.vis.embed as embed

embed.embed\_file(anim\_file)



**Next steps**

This tutorial has shown the complete code necessary to write and train a GAN. As a next step, you might like to experiment with a different dataset, for example the Large-scale Celeb Faces Attributes (CelebA) dataset [available on Kaggle](https://www.kaggle.com/jessicali9530/celeba-dataset). To learn more about GANs see the [NIPS 2016 Tutorial: Generative Adversarial Networks](https://arxiv.org/abs/1701.00160).