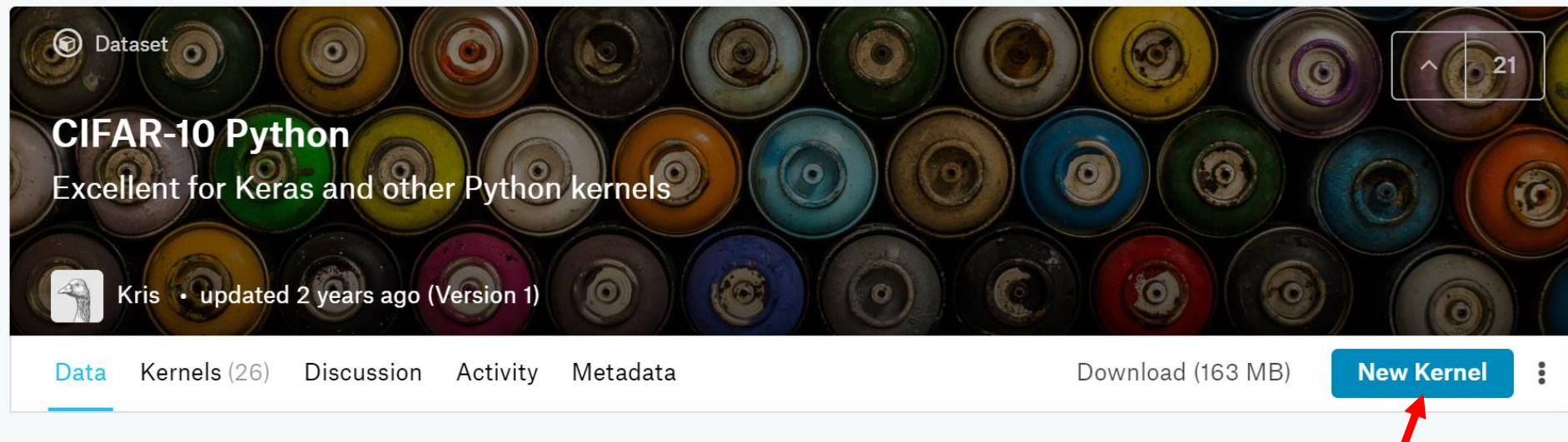


DCGAN 기초 및 실습

Youngtaek Hong, PhD

Kaggle, CIFAR-10 python dataset



Dataset

CIFAR-10 Python

Excellent for Keras and other Python kernels

Kris • updated 2 years ago (Version 1)

[Data](#) [Kernels \(26\)](#) [Discussion](#) [Activity](#) [Metadata](#) [Download \(163 MB\)](#) [New Kernel](#) [⋮](#)

Select Kernel Type



Script

```
import numpy as np # linear algebra
import pandas as pd # data processing,

# Input data files are available in the current working directory (by default in
# "../input/")

from subprocess import check_output
print(check_output(["ls", "../input"]))

# Any results you write to the current directory are saved as output files
```

- Python, R, RMarkdown
- Runs all the code, every time
- Ideal for fitting a model and competition submissions
- Shares code for review and RMarkdown reports

Notebook

Introduction

```
# Loading in the training data
train = pd.read_csv("../train.csv")
```

- Jupyter Notebooks in Python or R
- Runs cells of code and Markdown
- Ideal for interactive data exploration and polished analysis
- Shares insights through code & commentary



DCGAN 구현

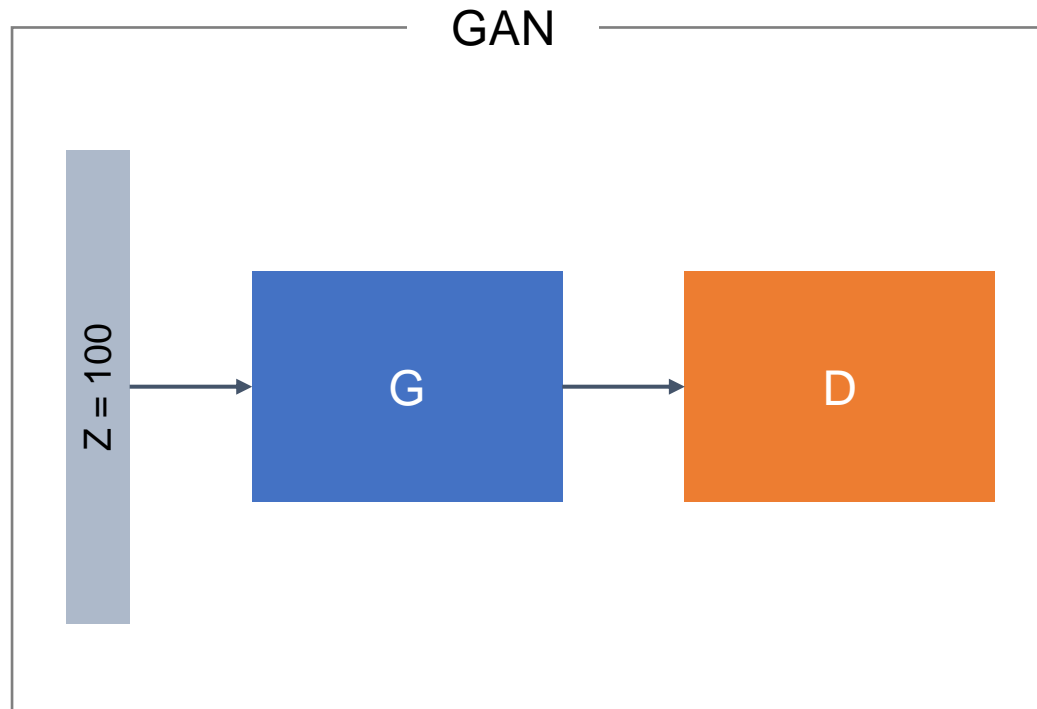
- 케라스를 이용한 가장 기본적인 형태의 DCGAN을 구현
- CIFAR10 데이터셋의 이미지로 DCGAN을 훈련하겠습니다.
- 이 데이터셋은 32×32 크기의 RGB 이미지 50,000개로 이루어져 있고 10개의 클래스를 가집니다(클래스마다 5,000개의 이미지가 있습니다).
- 문제를 간단하게 만들기 위해 "bird" 클래스의 이미지만 사용
- airplane : 0
automobile : 1
bird : 2
cat : 3
deer : 4
dog : 5
....

GAN 구현 세부 사항

- generator 네트워크는 (latent_dim,) 크기의 벡터를 (32, 32, 3) 크기의 이미지로 매핑합니다. (latent_dim는 임의로 설정 가능, 오늘 예제에서는 100으로 설정)
- Discriminator는 (32, 32, 3) 크기의 이미지가 진짜일 확률을 추정하여 이진 값(binary)으로 매핑합니다.
- 생성자와 판별자를 연결하는 gan 네트워크를 만듭니다. $Gan(x) = discriminator(generator(x))$ 입니다.
- 이 gan 네트워크는 잠재 공간의 벡터를 판별자의 평가로 매핑합니다.
- 판별자는 생성자가 잠재 공간의 벡터를 디코딩한 것이 얼마나 현실적인지를 평가합니다.

Network pipeline

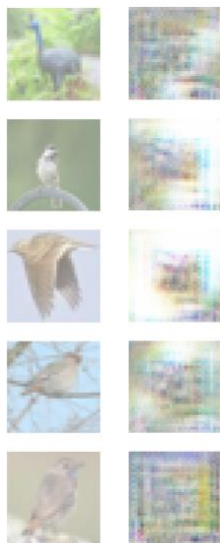
- $Gan(x) = \text{discriminator}(\text{generator}(x))$ 입니다.
- 이 gan 네트워크는 잠재 공간의 벡터를 판별자의 평가로 매핑합니다.
- 판별자는 생성자가 잠재 공간의 벡터를 디코딩한 것이 얼마나 현실적인지를 평가합니다.



GAN 구현 세부 사항

- “진짜”/”가짜” 레이블과 함께 진짜 이미지와 가짜 이미지 샘플을 사용해 판별자를 훈련합니다.
- 일반적인 이미지 분류 모델을 훈련하는 것과 동일합니다.

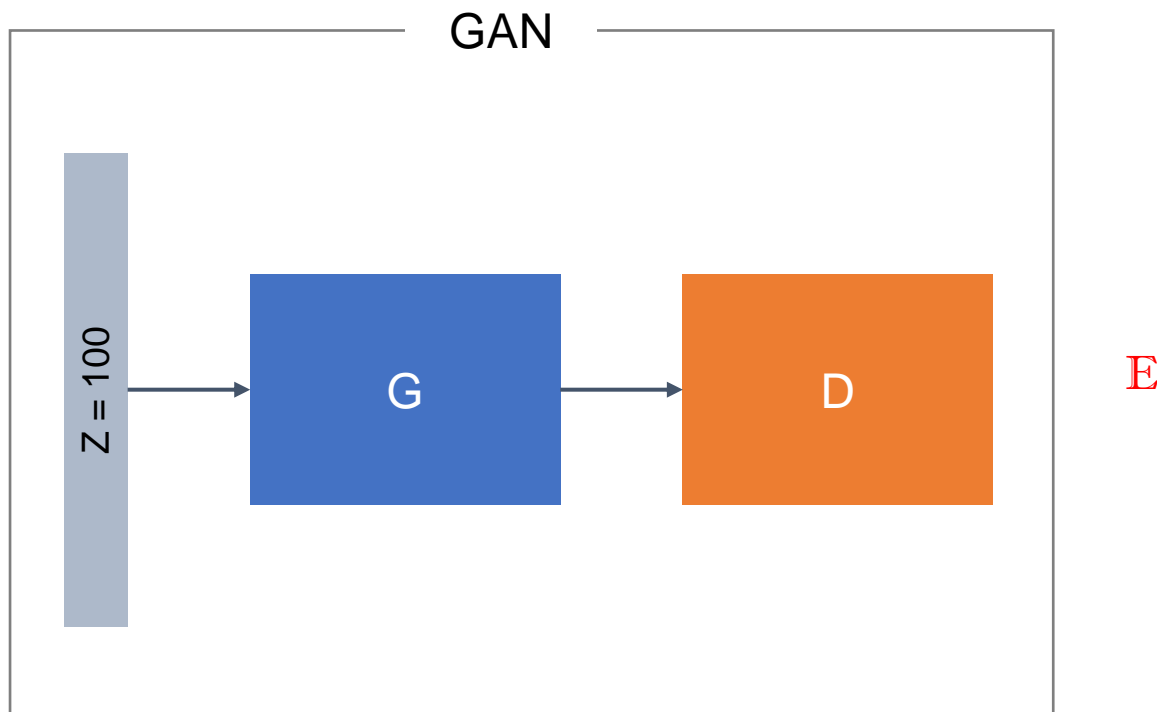
Supervised Learning



“진짜” ”가짜”

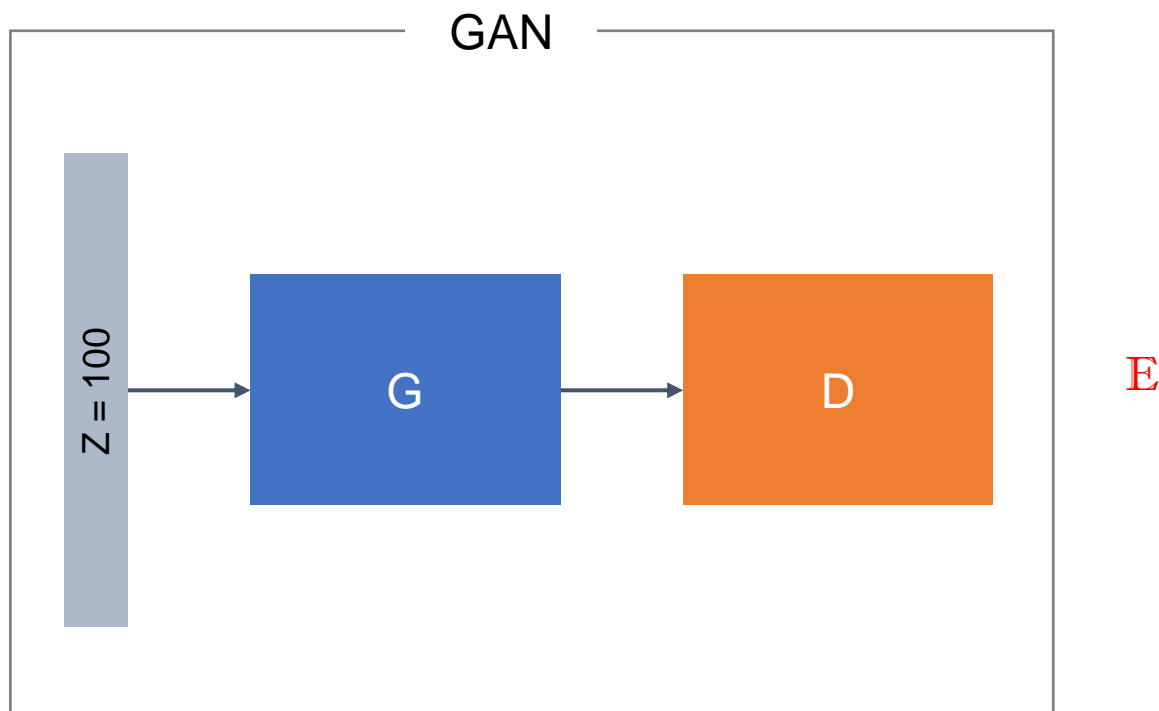
GAN 구현 세부 사항

- 생성자를 훈련하려면 gan 모델의 손실에 대한 생성자 가중치의 그래디언트를 사용합니다.



GAN 구현 세부 사항

- 이 말은 매 단계마다 생성자에 의해 디코딩된 이미지를 판별자가 "진짜"로 분류하도록 만드는 방향으로 생성자의 가중치를 이동한다는 뜻입니다. 다른 말로하면 판별자를 속이도록 생성자를 훈련합니다.



UNSUPERVISED REPRESENTATION LEARNING WITH DEEP CONVOLUTIONAL GENERATIVE ADVERSARIAL NETWORKS

Alec Radford & Luke Metz

indico Research

Boston, MA

`{alec, luke}@indico.io`

Soumith Chintala

Facebook AI Research

New York, NY

`soumith@fb.com`



DCGAN architecture

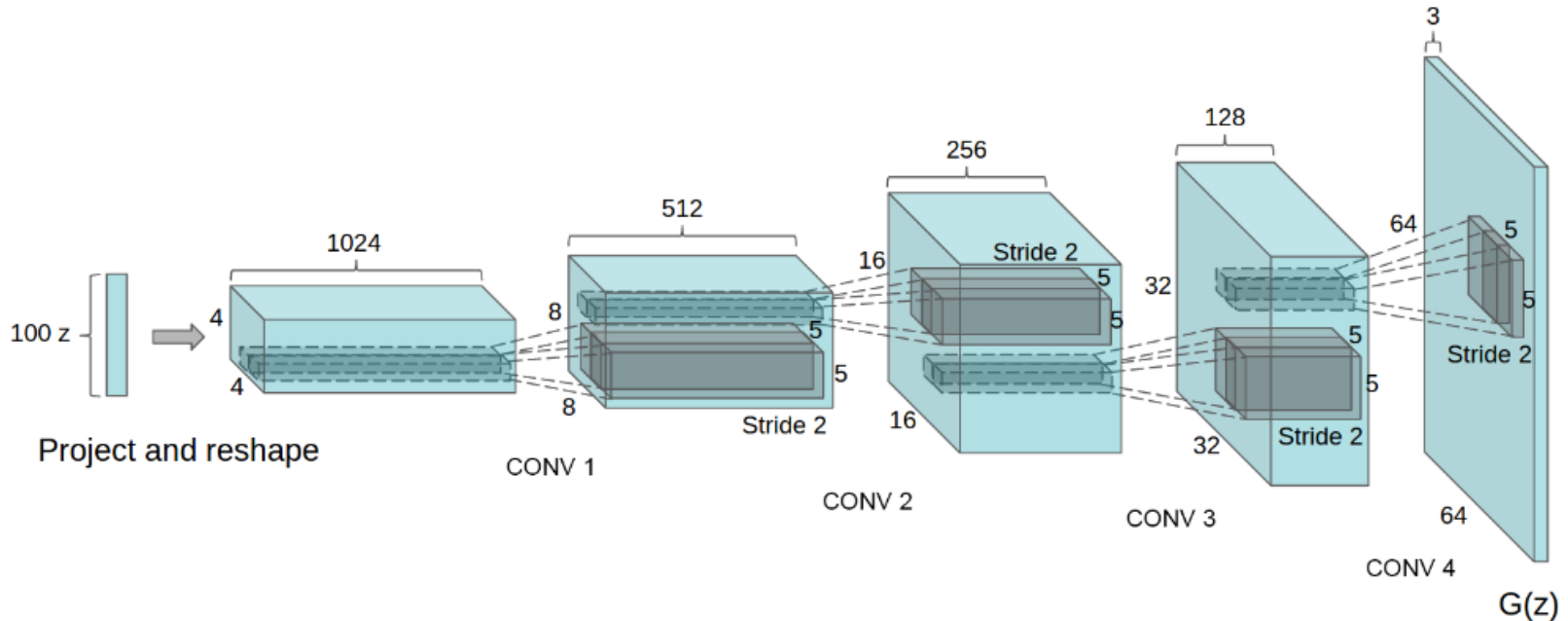


Figure 1: DCGAN generator used for LSUN scene modeling. A 100 dimensional uniform distribution Z is projected to a small spatial extent convolutional representation with many feature maps. A series of four fractionally-strided convolutions (in some recent papers, these are wrongly called deconvolutions) then convert this high level representation into a 64×64 pixel image. Notably, no fully connected or pooling layers are used.

Generator Architecture

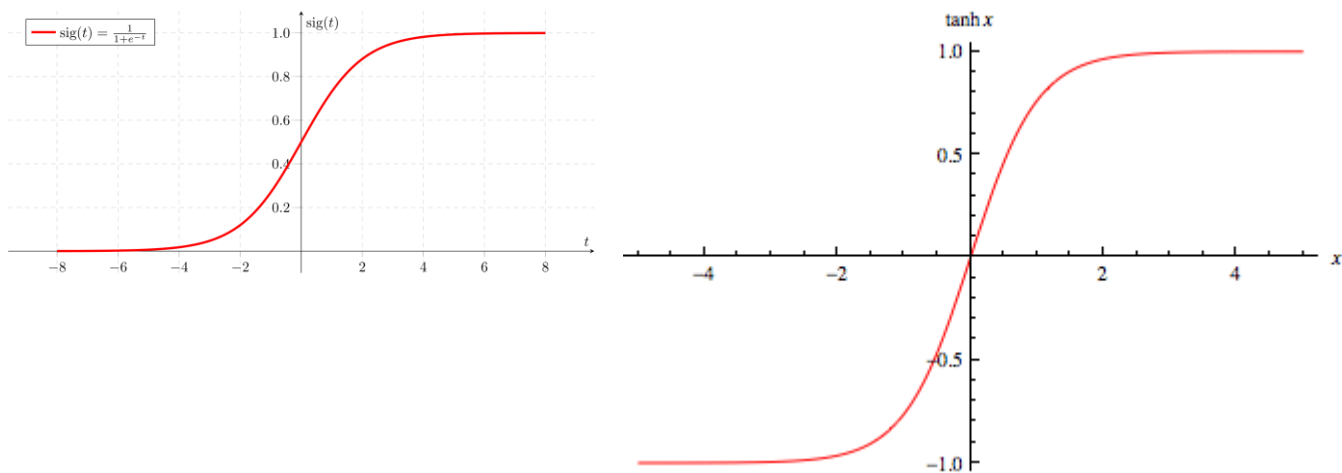
계층 #	계층 이름	구성
1	입력 계층	input_shape=(batch_size, 100), output_shape=(batch_size, 100)
2	조밀 계층	neurons=2048, input_shape=(batch_size, 100), output_shape=(batch_size, 2048), activation='relu'
3.	조밀 계층	neurons=16384, input_shape=(batch_size, 100), output_shape=(batch_size, 2048), batch_normalization=Yes, activation='relu'
4.	모양 변경 계층	input_shape=(batch_size=16384), output_shape=(batch_size, 8, 8, 256)
5.	상향 표본추출 계층	size=(2, 2), input_shape=(batch_size, 8, 8, 256), output_shape=(batch_size, 16, 16, 256)
6.	2D 합성곱 계층	filters=128, kernel_size=(5, 5), strides=(1, 1), padding='same', input_shape=(batch_size, 16, 16, 256), output_shape=(batch_size, 16, 16, 128), activation='relu'
7.	상향 표본추출 계층	size=(2, 2), input_shape=(batch_size, 16, 16, 128), output_shape=(batch_size, 32, 32, 128)
8.	2D 합성곱 계층	filters=64, kernel_size=(5, 5), strides=(1, 1), padding='same', activation=ReLU, input_shape=(batch_size, 32, 32, 128), output_shape=(batch_size, 32, 32, 64), activation='relu'
9.	상향 표본추출 계층	size=(2, 2), input_shape=(batch_size, 32, 32, 64), output_shape=(batch_size, 64, 64, 64)
10.	2D 합성곱 계층	filters=3, kernel_size=(5, 5), strides=(1, 1), padding='same', activation=ReLU, input_shape=(batch_size, 64, 64, 64), output_shape=(batch_size, 64, 64, 3), activation='tanh'

Discriminator Architecture

계층 #	계층 이름	구성
1.	입력 계층	input_shape=(batch_size, 64, 64, 3), output_shape=(batch_size, 64, 64, 3)
2.	2D 합성곱 계층	filters=128, kernel_size=(5, 5), strides=(1, 1), padding='valid', input_shape=(batch_size, 64, 64, 3), output_shape=(batch_size, 64, 64, 128), activation='leakyrelu', leaky_relu_alpha=0.2
3.	최대 풀링 계층	pool_size=(2, 2), input_shape=(batch_size, 64, 64, 128), output_shape=(batch_size, 32, 32, 128)
4.	2D 합성곱 계층	filters=256, kernel_size=(3, 3), strides=(1, 1), padding='valid', input_shape=(batch_size, 32, 32, 128), output_shape=(batch_size, 30, 30, 256), activation='leakyrelu', leaky_relu_alpha=0.2
5.	2D 최대 풀링 계층	pool_size=(2, 2), input_shape=(batch_size, 30, 30, 256), output_shape=(batch_size, 15, 15, 256)
6.	2D 합성곱 계층	filters=512, kernel_size=(3, 3), strides=(1, 1), padding='valid', input_shape=(batch_size, 15, 15, 256), output_shape=(batch_size, 13, 13, 512), activation='leakyrelu', leaky_relu_alpha=0.2
7.	2D 최대 풀링 계층	pool_size=(2, 2), input_shape=(batch_size, 13, 13, 512), output_shape=(batch_size, 6, 6, 512)
8.	평탄화 계층	input_shape=(batch_size, 6, 6, 512), output_shape=(batch_size, 18432)
9.	조밀 계층	neurons=1024, input_shape=(batch_size, 18432), output_shape=(batch_size, 1024), activation='leakyrelu', 'leakyrelu_alpha'=0.2
10.	조밀 계층	neurons=1, input_shape=(batch_size, 1024), output_shape=(batch_size, 1), activation='sigmoid'

GAN 구현 세부 사항

- 생성자의 마지막 활성화로 sigmoid 대신 tanh 함수를 사용
- 균등 분포가 아니고 정규 분포(가우시안 분포)를 사용하여 잠재 공간에서 포인트를 샘플링



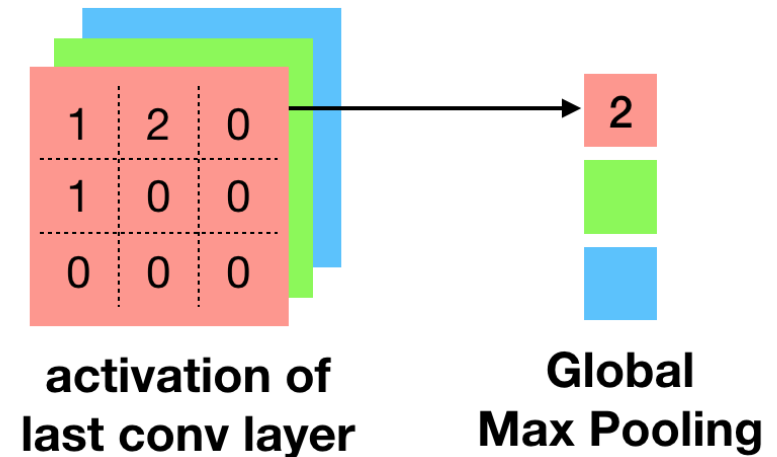
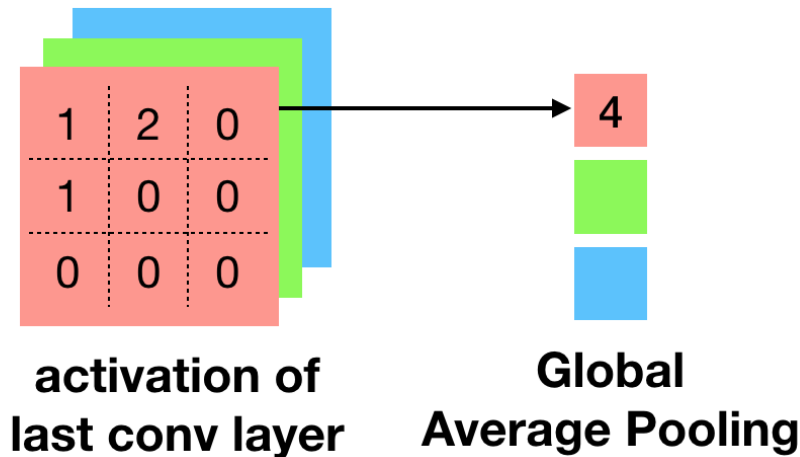
- 판별자에 드롭아웃을 사용하거나 판별자를 위해 레이블에 랜덤 노이즈를 추가합니다.

Keypoints

- The first is the all convolutional net (Springenberg et al., 2014) which replaces deterministic spatial pooling functions (such as maxpooling) with strided convolutions, allowing the network to learn its own spatial downsampling.
- We use this approach in our generator, allowing it to learn its own spatial upsampling, and discriminator.

Keypoints

- Second is the trend towards **eliminating fully connected layers** on top of convolutional features.
- The strongest example of this is **global average pooling** which has been utilized in state-of-the-art image classification models (Mordvintsev et al.).
- We found **global average pooling increased model stability** but hurt convergence speed.



Keypoints

- Third is **Batch Normalization** (Ioffe & Szegedy, 2015) which stabilizes learning by normalizing the input to each unit to have zero mean and unit variance.
- This helps deal with training problems that arise due to poor initialization and helps gradient flow in deeper models.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_1 \dots x_m\}$;

Parameters to be learned: γ, β

Output: $\{y_i = \text{BN}_{\gamma, \beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^m x_i \quad // \text{ mini-batch mean}$$

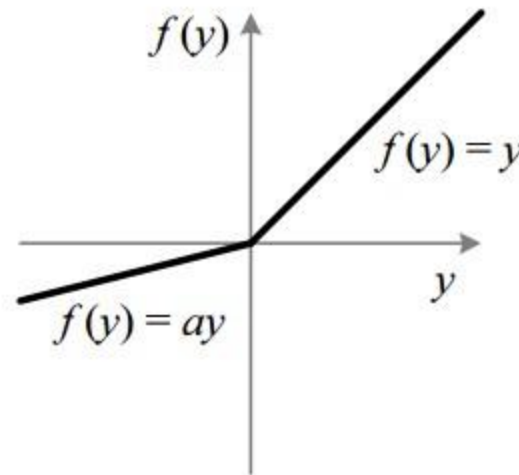
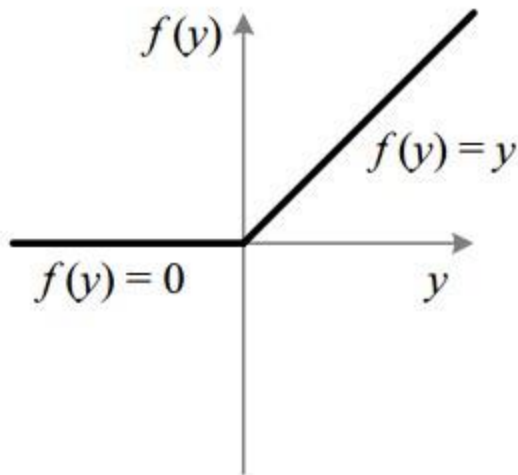
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^m (x_i - \mu_{\mathcal{B}})^2 \quad // \text{ mini-batch variance}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \quad // \text{ normalize}$$

$$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \text{BN}_{\gamma, \beta}(x_i) \quad // \text{ scale and shift}$$

Keypoints

- The **ReLU activation** (Nair & Hinton, 2010) is used in the generator **except for the output layer** which uses the **Tanh function**.
- We observed that using a bounded activation allowed the model to learn more quickly to saturate and cover the color space of the training distribution.
- Within the **discriminator** we found the **leaky rectified activation** (Maas et al., 2013) (Xu et al., 2015) to work well, especially for higher resolution modeling.



Summary

Architecture guidelines for stable Deep Convolutional GANs

- Replace any pooling layers with strided convolutions (discriminator) and fractional-strided convolutions (generator).
- Use batchnorm in both the generator and the discriminator.
- Remove fully connected hidden layers for deeper architectures.
- Use ReLU activation in generator for all layers except for the output, which uses Tanh.
- Use LeakyReLU activation in the discriminator for all layers.

Details of Adversarial training

- No pre-processing was applied to training images besides **scaling** to the range of the tanh activation function **[-1, 1]**.
- All models were trained with mini-batch stochastic gradient descent (SGD) with a mini-batch size of 128.
- All **weights were initialized** from a **zero-centered** Normal distribution with standard deviation 0.02.
- In the **LeakyReLU**, the slope of the leak was set to **0.2** in all models.
- We used the **Adam optimizer** (Kingma & Ba, 2014) with tuned hyperparameters.
 - learning rate of 0.0002
 - momentum term β_1 at the 0.5

GAN 훈련 방법

- 잠재 공간에서 무작위로 포인트를 뽑습니다(랜덤 노이즈).
- 이 랜덤 노이즈를 사용해 `generator`에서 이미지를 생성합니다.
- 생성된 이미지와 진짜 이미지를 섞습니다.
- 진짜와 가짜가 섞인 이미지와 이에 대응하는 타깃을 사용해 `discriminator`를 훈련합니다.
- 타깃은 "진짜"(실제 이미지일 경우) 또는 "가짜"(생성된 이미지일 경우)입니다.

Experimental Results

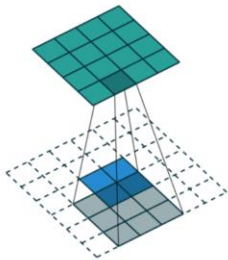
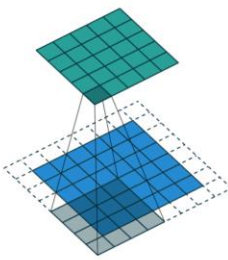
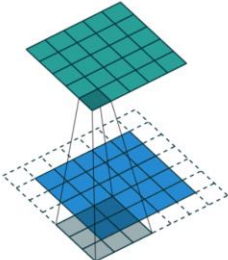
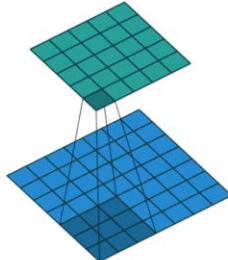
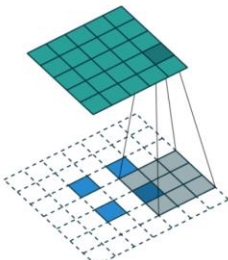
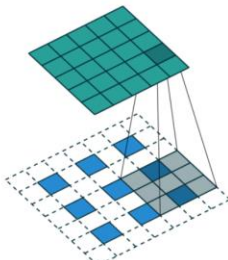
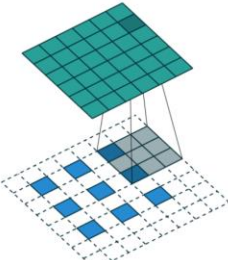


Figure 10: More face generations from our Face DCGAN.

Fractional stride convolution

Transposed convolution animations

N.B.: Blue maps are inputs, and cyan maps are outputs.

			
No padding, no strides, transposed	Arbitrary padding, no strides, transposed	Half padding, no strides, transposed	Full padding, no strides, transposed
			
No padding, strides, transposed	Padding, strides, transposed	Padding, strides, transposed (odd)	

https://github.com/vdumoulin/conv_arithmetic



Generative Adversarial Networks

Introduction to GANs

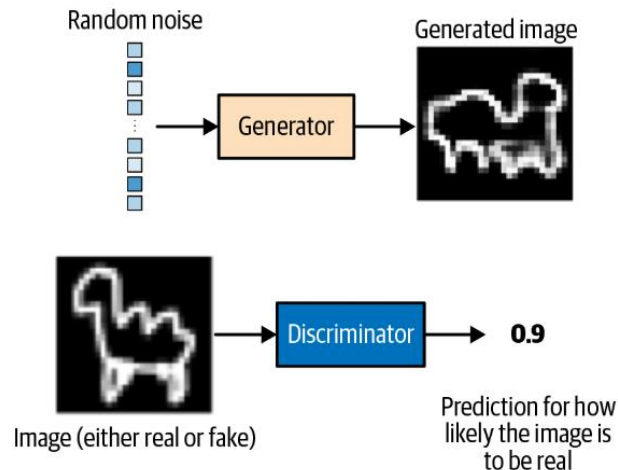
GAN

A **GAN** is a battle between two adversaries, the generator and the discriminator.

The **Generator** tries to convert random noise into observations that look as if they have been sampled from the original dataset and the **Discriminator** tries to predict whether an observation comes from the original dataset or is one of the generator's forgeries.

The key to GANs

How we alternate the training of the two networks, so that as the generator becomes more adept at fooling the discriminator, the discriminator must adapt in order to maintain its ability to correctly identify which observations are fake.



My First GAN

```
gan = GAN(input_dim = (28,28,1)
    , discriminator_conv_filters = [64,64,128,128]
    , discriminator_conv_kernel_size = [5,5,5,5]
    , discriminator_conv_strides = [2,2,2,1]
    , discriminator_batch_norm_momentum = None
    , discriminator_activation = 'relu'
    , discriminator_dropout_rate = 0.4
    , discriminator_learning_rate = 0.0008
    , generator_initial_dense_layer_size = (7, 7, 64)
    , generator_upsample = [2,2, 1, 1]
    , generator_conv_filters = [128,64, 64,1]
    , generator_conv_kernel_size = [5,5,5,5]
    , generator_conv_strides = [1,1, 1, 1]
    , generator_batch_norm_momentum = 0.9
    , generator_activation = 'relu'
    , generator_dropout_rate = None
    , generator_learning_rate = 0.0004
    , optimiser = 'rmsprop'
    , z_dim = 100
)
```

Defining the GAN

My First GAN

The Discriminator

- The goal of the discriminator is to predict if an image is real or fake.
- We will use 'DCGAN(Deep Convolutional generative adversarial network)' instead of original GAN in the paper.

Layer (type)	Output Shape	Param #
discriminator_input (InputLayer)	(None, 28, 28, 1)	0
discriminator_conv_0 (Conv2D)	(None, 14, 14, 64)	1664
activation_1 (Activation)	(None, 14, 14, 64)	0
dropout_1 (Dropout)	(None, 14, 14, 64)	0
discriminator_conv_1 (Conv2D)	(None, 7, 7, 64)	102464
activation_2 (Activation)	(None, 7, 7, 64)	0
dropout_2 (Dropout)	(None, 7, 7, 64)	0
discriminator_conv_2 (Conv2D)	(None, 4, 4, 128)	204928
activation_3 (Activation)	(None, 4, 4, 128)	0
dropout_3 (Dropout)	(None, 4, 4, 128)	0
discriminator_conv_3 (Conv2D)	(None, 4, 4, 128)	409728
activation_4 (Activation)	(None, 4, 4, 128)	0
dropout_4 (Dropout)	(None, 4, 4, 128)	0
flatten_1 (Flatten)	(None, 2048)	0
dense_1 (Dense)	(None, 1)	2049
Total params: 720,833		
Trainable params: 720,833		
Non-trainable params: 0		

Discriminator of GAN

My First GAN

The Discriminator

```
discriminator_input = Input(shape=self.input_dim, name='discriminator_input') ❶
x = discriminator_input

for i in range(self.n_layers_discriminator): ❷

    x = Conv2D(
        filters = self.discriminator_conv_filters[i]
        , kernel_size = self.discriminator_conv_kernel_size[i]
        , strides = self.discriminator_conv_strides[i]
        , padding = 'same'
        , name = 'discriminator_conv_' + str(i)
    )(x)

    if self.discriminator_batch_norm_momentum and i > 0:
        x = BatchNormalization(momentum = self.discriminator_batch_norm_momentum)(x)

    x = Activation(self.discriminator_activation)(x)

    if self.discriminator_dropout_rate:
        x = Dropout(rate = self.discriminator_dropout_rate)(x)

x = Flatten()(x) ❸
discriminator_output = Dense(1, activation='sigmoid'
    , kernel_initializer = self.weight_init)(x) ❹

discriminator = Model(discriminator_input, discriminator_output) ❺
```

- ❶ Define the input to the discriminator
- ❷ Stack convolutional layers on top of each other
- ❸ Flatten the last convolutional layer to a vector
- ❹ Dense layer of one unit, with a sigmoid activation function that transforms the output from the dense layer to the range[0,1]
- ❺ The Keras model that defines the discriminator – a model that takes an input image and outputs a single number between 0 and 1

My First GAN

The Generator

- The input to the generator is a vector, usually drawn from a multivariate standard normal distribution.
- The output is an image of the same size as an image in the original training data.
- The generator of a GAN fulfills exactly the same purpose as the decoder of a VAE: converting a vector in the latent space to an image.
- The concept of mapping from a latent space back to the domain is very common in generative modeling as it gives us the ability to manipulate vectors in the latent space to change high-level features of images in the original domain.

My First GAN

The Generator

Layer (type)	Output Shape	Param #
generator_input (InputLayer)	(None, 100)	0
dense_9 (Dense)	(None, 3136)	316736
batch_normalization_10 (Batch Normalization)	(None, 3136)	12544
activation_36 (Activation)	(None, 3136)	0
reshape_4 (Reshape)	(None, 7, 7, 64)	0
up_sampling2d_10 (UpSampling2D)	(None, 14, 14, 64)	0
generator_conv_0 (Conv2D)	(None, 14, 14, 128)	204928
batch_normalization_11 (Batch Normalization)	(None, 14, 14, 128)	512
activation_37 (Activation)	(None, 14, 14, 128)	0
up_sampling2d_11 (UpSampling2D)	(None, 28, 28, 128)	0
generator_conv_1 (Conv2D)	(None, 28, 28, 64)	204864
batch_normalization_12 (Batch Normalization)	(None, 28, 28, 64)	256
activation_38 (Activation)	(None, 28, 28, 64)	0
generator_conv_2 (Conv2D)	(None, 28, 28, 64)	102464
batch_normalization_13 (Batch Normalization)	(None, 28, 28, 64)	256
activation_39 (Activation)	(None, 28, 28, 64)	0
generator_conv_3 (Conv2D)	(None, 28, 28, 1)	1601
activation_40 (Activation)	(None, 28, 28, 1)	0
Total params: 844,161		
Trainable params: 837,377		
Non-trainable params: 6,784		

Generator of GAN

My First GAN

The Generator

* UpSampling layer

In Keras, we use *Upsampling2D* layer to double the width and height of the input tensor.

This simply repeats each row and column of its input in order to double the size.

We then follow this with a normal convolutional layer with stride 1 to perform the convolution operation.

It is similar idea to convolutional transpose, but instead of filling the gaps between pixels with zeros, upsampling just repeats the existing pixel values.

My First GAN

The Generator

```
generator_input = Input(shape=(self.z_dim,), name='generator_input') ❶
x = generator_input

x = Dense(np.prod(self.generator_initial_dense_layer_size))(x) ❷

if self.generator_batch_norm_momentum:
    x = BatchNormalization(momentum = self.generator_batch_norm_momentum)(x)
x = Activation(self.generator_activation)(x)

x = Reshape(self.generator_initial_dense_layer_size)(x) ❸

if self.generator_dropout_rate:
    x = Dropout(rate = self.generator_dropout_rate)(x)

for i in range(self.n_layers_generator): ❹

    x = UpSampling2D()(x)
    x = Conv2D(
        filters = self.generator_conv_filters[i]
        , kernel_size = self.generator_conv_kernel_size[i]
        , padding = 'same'
        , name = 'generator_conv_' + str(i)
    )(x)

    if i < n_layers_generator - 1: ❺
        if self.generator_batch_norm_momentum:
            x = BatchNormalization(momentum = self.generator_batch_norm_momentum)(x)
        x = Activation('relu')(x)
    else:
        x = Activation('tanh')(x)

generator_output = x
generator = Model(generator_input, generator_output) ❻
```

- ❶ Define the input to the generator – a vector of length 100.
- ❷ We follow this with a *Dense* layer consisting of 3,136 units.
- ❸ which, after applying batch normalization and a *ReLU* activation function, is reshaped to a 7x7x64 tensor.
- ❹ We pass this through four *Conv2D* layers, the first two preceded by *Upsampling2D* layers, to reshape the tensor to 14x14, then 28x28(the original image size). In all but the last layer, we use batch normalization and *ReLU* activation.
- ❺ After the final *Conv2D* layer, we use a *tanh* activation to transform the output to the range [-1, 1]

GAN

Training the GAN

1. Train the Discriminator

- We can train the discriminator by creating a training set where some of the images are randomly selected real observations from the training set and some are outputs from the generator.
- We must freeze the weights of the discriminator while we are training the combined model.

2. Train the Generator

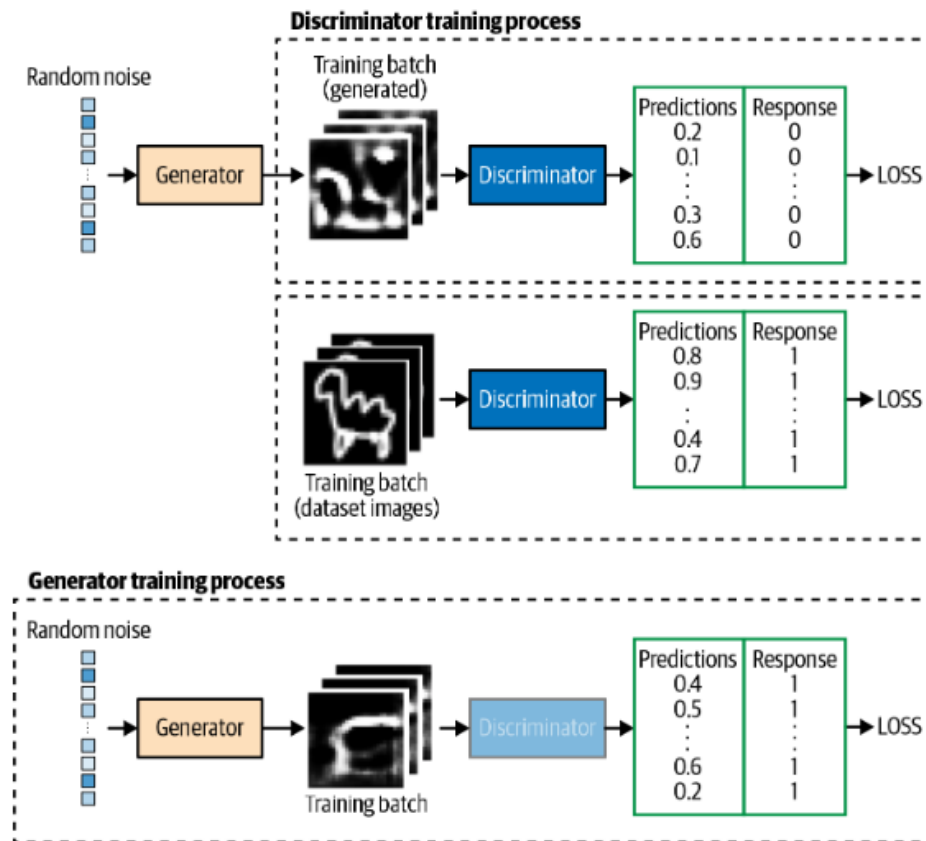
- We only want the image that is generated to fool the discriminator – that is, when the image is fed as input to the discriminator, we want the output to be close to 1.
- We must first connect it to the discriminator to create a keras model that we can train.
- We feed the output from the generator into the discriminator so that the output from this combined model is the probability that the generated image is real, according to the discriminator.

3. Loss function

- Binary cross-entropy loss between the output from the discriminator and the response vector of 1.

My First GAN

Training the GAN



Training the GAN

My First GAN

Training the GAN

```
### COMPILER MODEL THAT TRAINS THE DISCRIMINATOR

self.discriminator.compile(
    optimizer= RMSprop(lr=0.0008)
    , loss = 'binary_crossentropy'
    , metrics = ['accuracy']
) ❶

### COMPILER MODEL THAT TRAINS THE GENERATOR

self.discriminator.trainable = False ❷
model_input = Input(shape=(self.z_dim,), name='model_input')
model_output = discriminator(self.generator(model_input))
self.model = Model(model_input, model_output) ❸

self.model.compile(
    optimizer=RMSprop(lr=0.0004)
    , loss='binary_crossentropy'
    , metrics=['accuracy']
) ❹
```

- ① The discriminator is compiled with binary cross-entropy loss, as the response is binary and we have one output unit with sigmoid activation.
- ② Next, we freeze the discriminator weights—this doesn't affect the existing discriminator model that we have already compiled.
- ③ We define a new model whose input is a 100-dimensional latent vector; this is passed through the generator and frozen discriminator to produce the output probability.
- ④ Again, we use a binary cross-entropy loss for the combined model—the learning rate is slower than the discriminator as generally we would like the discriminator to be stronger than the generator. The learning rate is a parameter that should be tuned carefully for each GAN problem setting.

My First GAN

Training the GAN

```
def train_discriminator(x_train, batch_size):

    valid = np.ones((batch_size,1))
    fake = np.zeros((batch_size,1))

    # TRAIN ON REAL IMAGES
    idx = np.random.randint(0, x_train.shape[0], batch_size)
    true_imgs = x_train[idx]
    self.discriminator.train_on_batch(true_imgs, valid) ❶

    # TRAIN ON GENERATED IMAGES
    noise = np.random.normal(0, 1, (batch_size, z_dim))
    gen_imgs = generator.predict(noise)
    self.discriminator.train_on_batch(gen_imgs, fake) ❷

def train_generator(batch_size):

    valid = np.ones((batch_size,1))

    noise = np.random.normal(0, 1, (batch_size, z_dim))
    self.model.train_on_batch(noise, valid) ❸

epochs = 2000
batch_size = 64

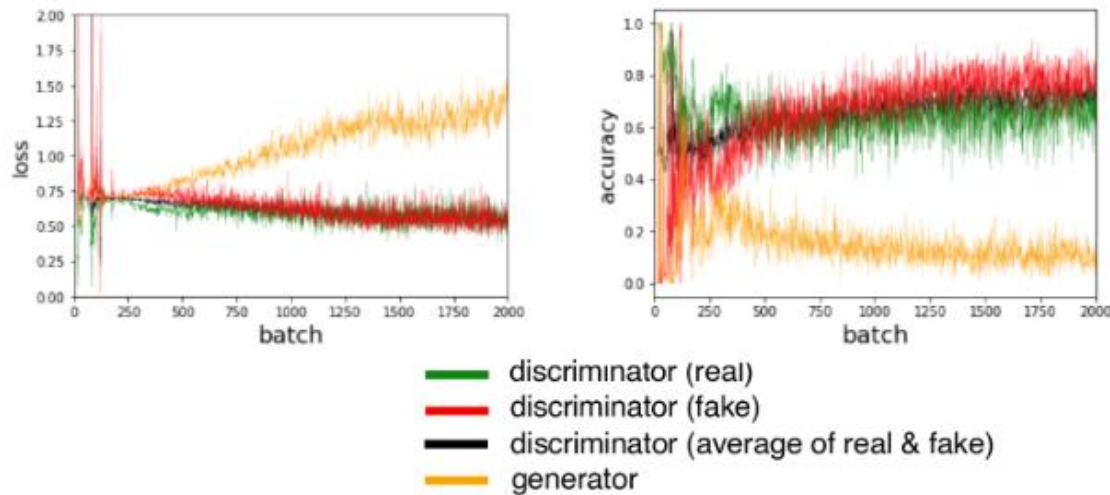
for epoch in range(epochs):

    train_discriminator(x_train, batch_size)
    train_generator(batch_size)
```

- ❶ One batch update of the discriminator involves first training on a batch of true images with the response 1
- ❷ then on a batch of generated images with the response 0.
- ❸ One batch update of the generator involves training on a batch of generated images with the response 1. As the discriminator is frozen, its weights will not be affected; instead, the generator weights will move in the direction that allows it to better generate images that are more likely to fool the discriminator (i.e., make the discriminator predict values close to 1).

My First GAN

Training the GAN

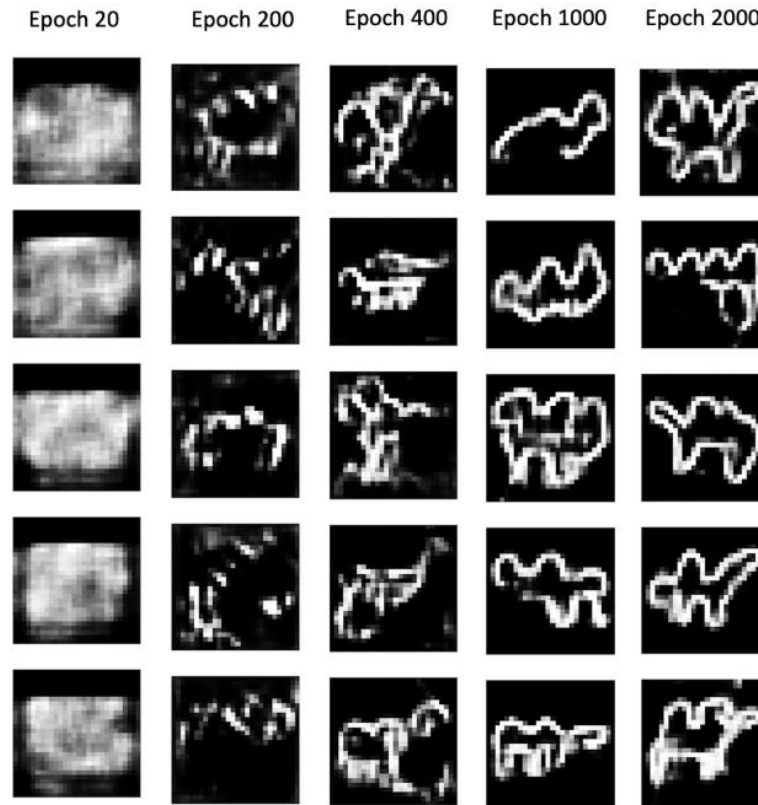


Loss and accuracy of the discriminator and generator during training

After a suitable number of epochs, the discriminator and generator will have found an equilibrium that allows **the generator to learn meaningful information from the discriminator and the quality of the images will start to improve.**

My First GAN

Training the GAN



Output from the generator at specific epochs during training

My First GAN

Training the GAN

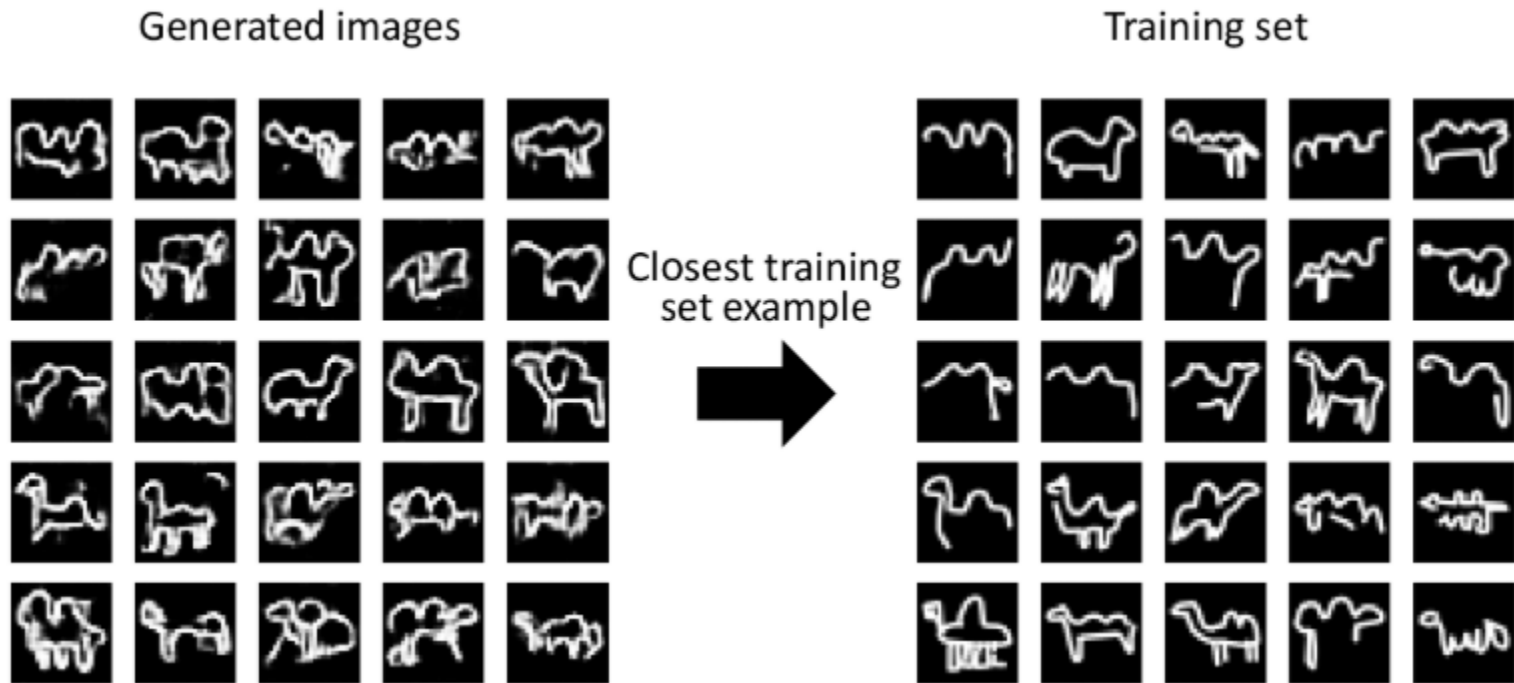
- Another requirement of a successful generative model is that it doesn't only reproduce images from the training set. To test this, we can find the image from the training set that is closest to a particular generated example.

```
def l1_compare_images(img1, img2):  
    return np.mean(np.abs(img1 - img2))
```

L1 distance

My First GAN

Training the GAN

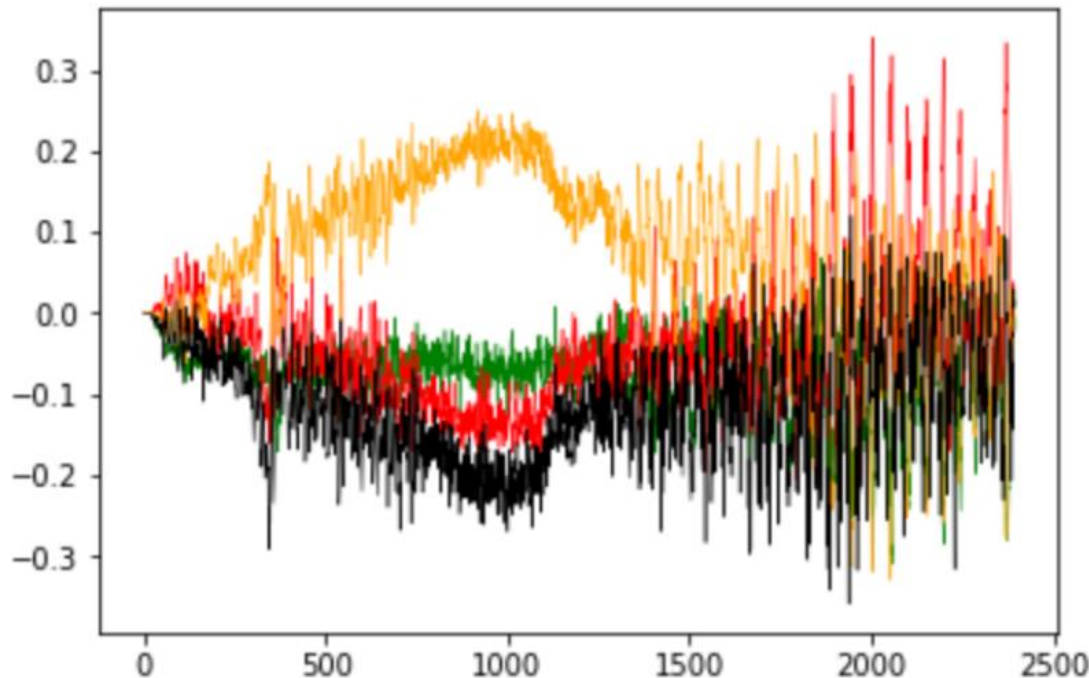


Closest matches of generated images from the training set.

GAN Challenges

Oscillating loss

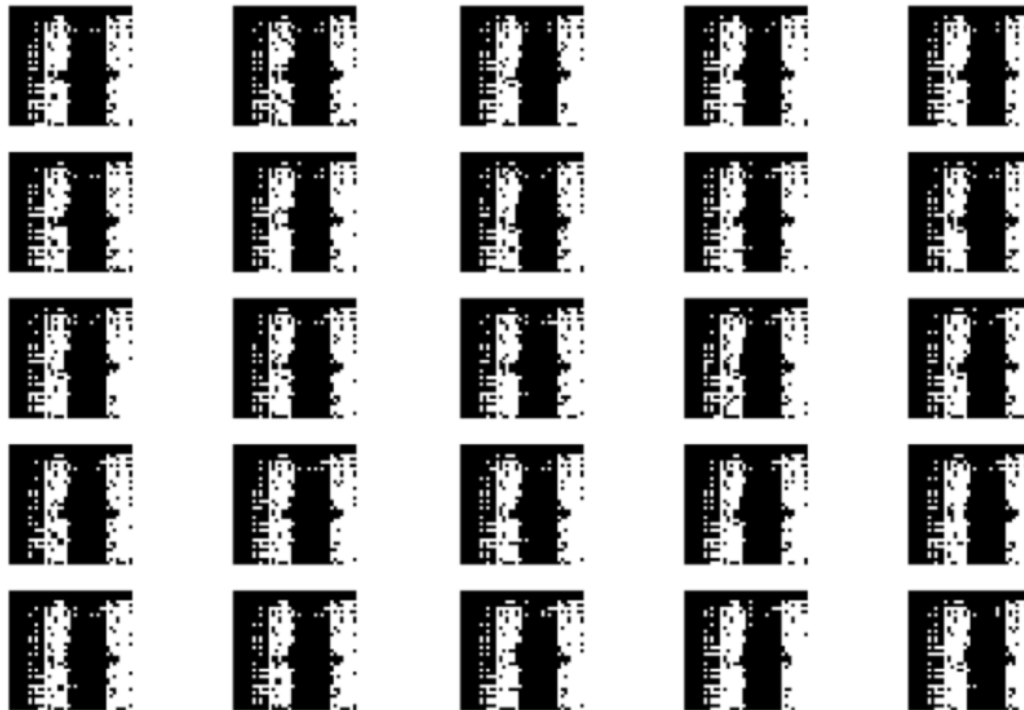
The loss of the discriminator and generator can start to oscillate wildly, rather than exhibiting long-term stability. Typically, there is some small oscillation of the loss between batches, but in the long term you should be looking for loss that stabilizes or gradually increases or decreases, rather than erratically fluctuating, to ensure GAN converges and improves over time.



GAN Challenges

Mode Collapse

Mode collapse occurs when the generator finds a small number of samples that fool the discriminator and therefore isn't able to produce any examples other than this limited set.



GAN Challenges

Uninformative Loss

Since the generator is only graded against the current discriminator and the discriminator is constantly improving, we cannot compare the loss function evaluated at different points in the training process. This lack of correlation between the generator loss and image quality sometimes makes GAN training difficult to monitor.

Hyperparameters

There are a large number of hyperparameters to tune. GANs are highly sensitive to very slight changes in all of parameters, and finding a set of parameters that works is often a case of educated trial and error, rather than following an established set of guidelines.

This is why it is important to understand the inner workings of the GAN and know how to interpret the loss function—so that you can identify sensible adjustments to the hyperparameters that might improve the stability of the model.

Tackling the GAN Challenges

In recent years, several key advancements have drastically improved the overall stability of GAN models and diminished the likelihood of some of the problems listed earlier, such as mode collapse.