Introduction To Generative Adversarial Networks
GANs

Hichem Felouat

hichemfel@gmail.com

https://www.researchgate.net/profile/Hichem_Felouat
https://www.linkedin.com/in/hichemfelouat/

Introduction

- **Generative adversarial networks (GANs)** were proposed in a **2014** paper [1].

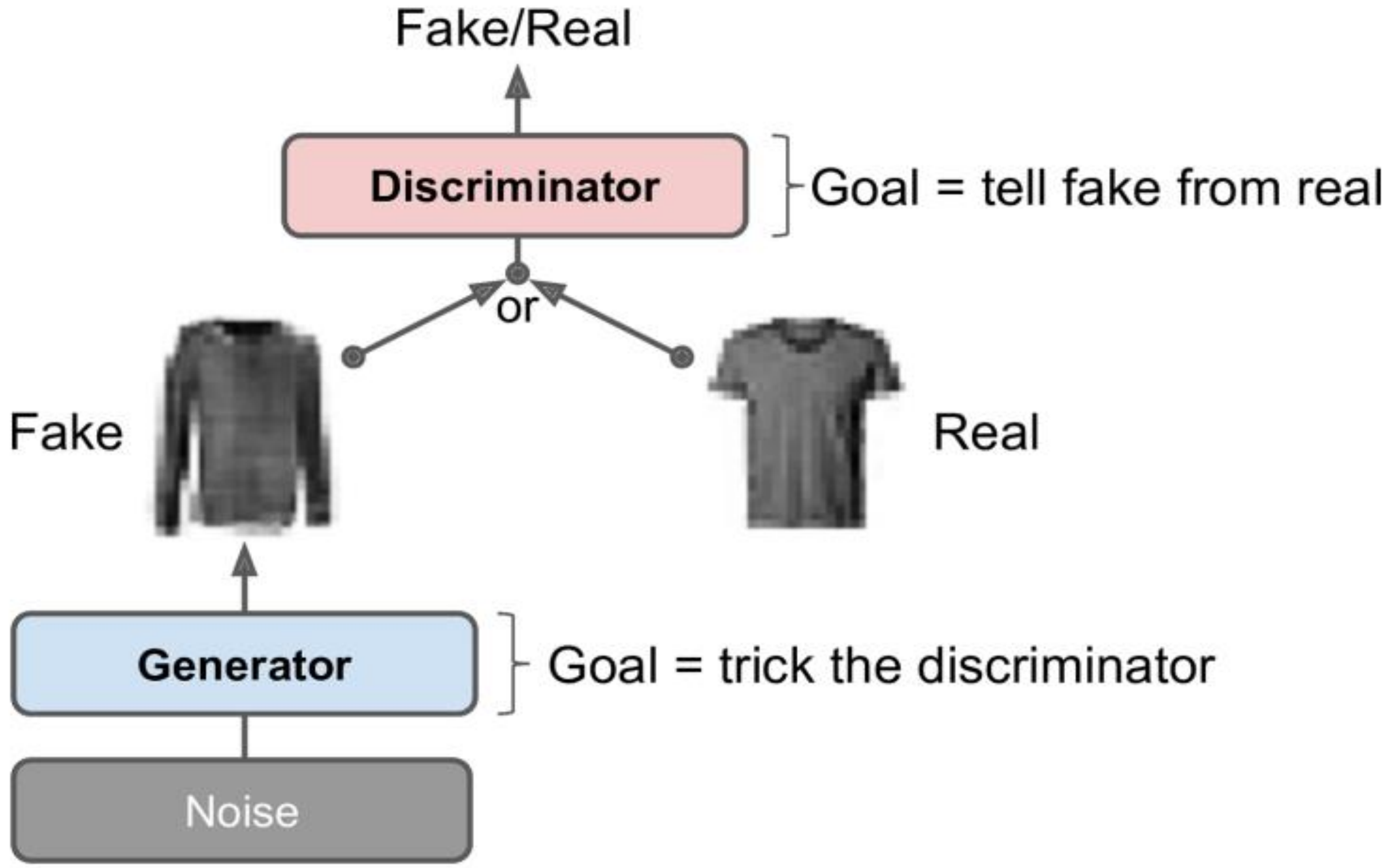
- A GAN is composed of two neural networks (**Generator & Discriminator**).

- **Generator:** Takes a **random distribution** as input (typically Gaussian) and **outputs some data** - typically, an image. You can think of the random inputs as the latent representations (i.e., codings) of the image to be generated.

- **Discriminator:** Takes either a fake image from the generator or a real image from the training set as input, and must guess whether the input image is fake or real.

[1] Ian Goodfellow et al., "Generative Adversarial Nets," Proceedings of the 27th International Conference on Neural Information Processing Systems 2 (2014): 2672–2680.

Introduction



Applications of GAN

1) Generate Examples for Image Datasets
2) Generate Photographs of Human Faces
3) Generate Realistic Photographs
4) Generate Cartoon Characters
5) Image-to-Image Translation
6) Text-to-Image Translation
7) Semantic-Image-to-Photo Translation
8) Face Frontal View Generation
9) Generate New Human Poses
10) Photos to Emojis
11) Photograph Editing
12) Face Aging
13) Photo Blending
14) Super Resolution
15) Photo Inpainting
16) Clothing Translation
17) Video Prediction
18) 3D Object Generation

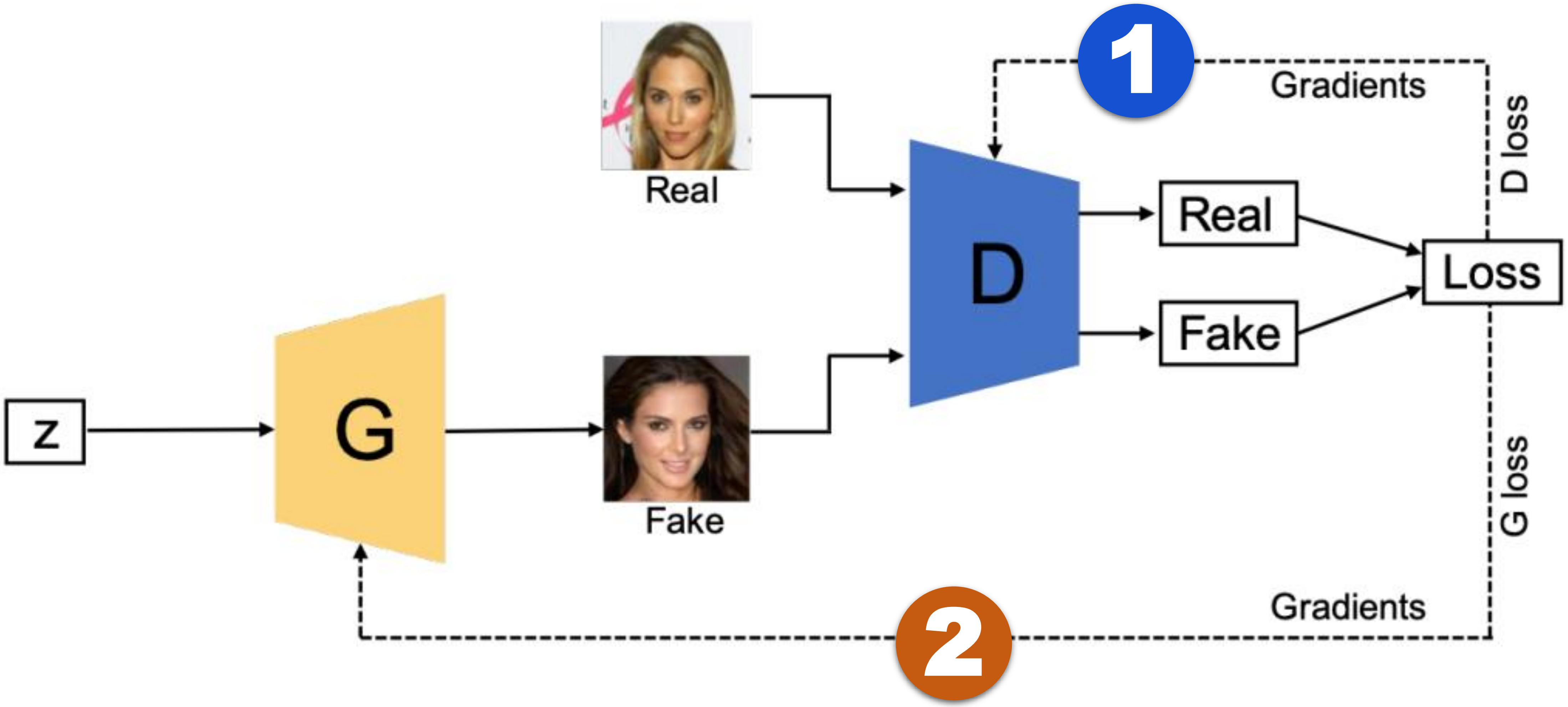
Generative adversarial networks: a survey on applications and challenges
https://link.springer.com/article/10.1007/s13735-020-00196-w
gans-awesome-applications
https://github.com/nashory/gans-awesome-applications
https://machinelearningmastery.com/impressive-applications-of-generative-adversarial-networks/

GAN Training

- The generator and the discriminator **have opposite goals**: the discriminator tries to tell fake images from real images, while the generator tries to produce images that look real enough to trick the discriminator.

- Because the GAN is composed of two networks with different objectives, **it can not be trained like a regular neural network**. Each training iteration is divided into **two phases**:

GAN Training



GAN Training

First phase:

- We train **the discriminator**. A batch of **real images** is sampled from the training set and is completed with an equal number of **fake images** produced by the generator (**The labels** are: **0 = fake images and 1 = real images**).
- The discriminator is trained on this labeled batch **for one step**, using **the binary cross-entropy loss**.
- **Backpropagation** only optimizes **the weights of the discriminator** during this phase.

GAN Training

Second phase:

- We train **the generator**. We first use it to produce another **batch of fake images**, and once again the discriminator is used to tell whether the images are fake or real.

- This time **we do not add real images in the batch** (The generator never actually sees any real images).

- **The weights of the discriminator are frozen** during this step, so **backpropagation** only affects **the weights of the generator**.

Common Problems

- **Vanishing Gradients:** when the discriminator doesn't provide enough information for the generator to make progress (The original GAN paper proposed a modification to minimax loss to deal with vanishing gradients)[2].

- **Mode Collapse:** this is when the generator starts **producing the same output** (or a small set of outputs) over and over again. How can this happen? Suppose that the generator gets better at producing convincing (class1) than any other class. It will fool the discriminator a bit more with (class1), and this will encourage it to produce even more images of (class1). Gradually, it will forget how to produce anything else.

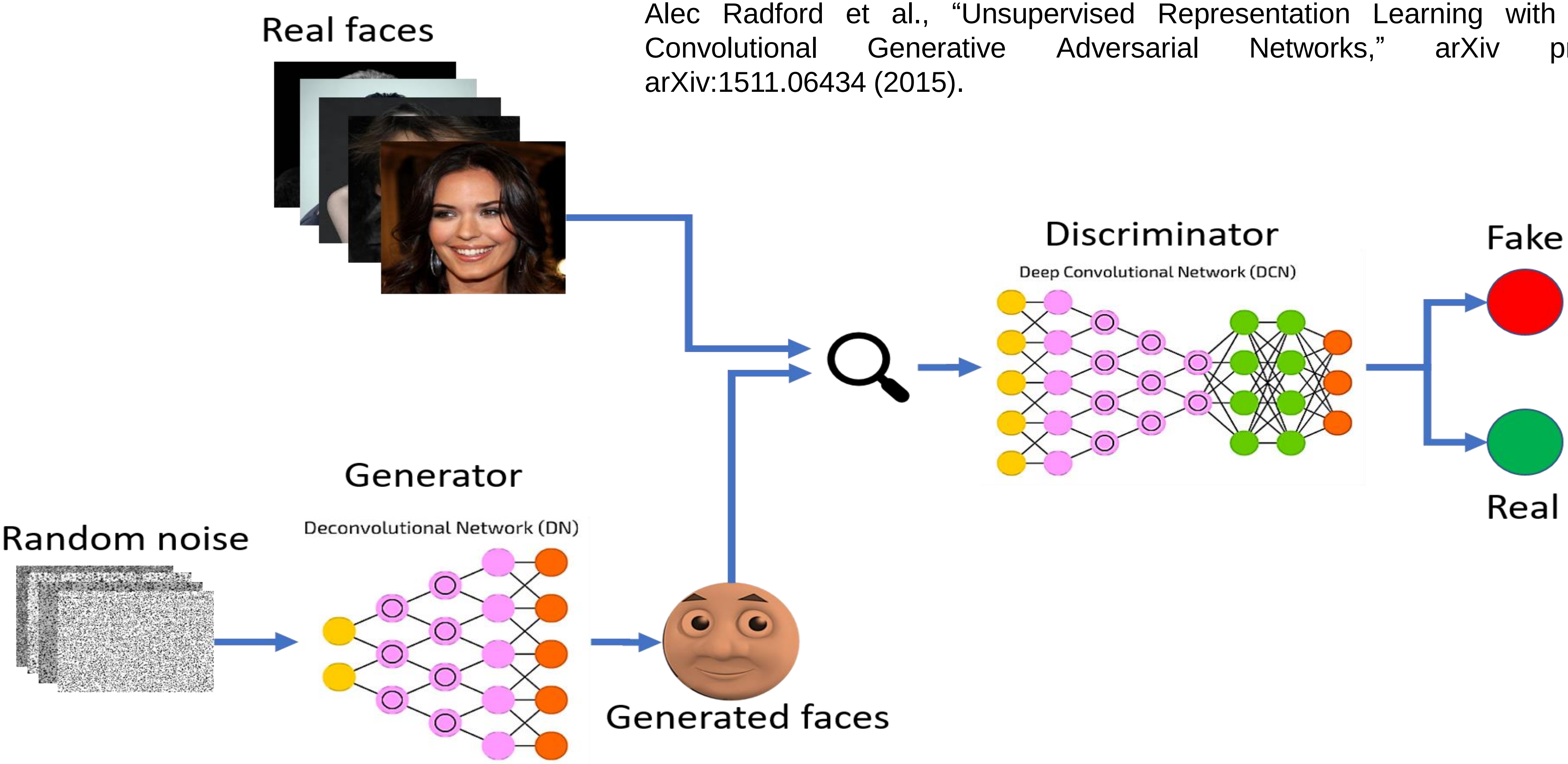
- **GANs are very sensitive to the hyperparameters:** you may have to spend a lot of effort fine-tuning them.

[2] https://developers.google.com/machine-learning/gan/loss

Deep Convolutional GANs

Deep Convolutional GANs (DCGANs) - 2015

Alec Radford et al., “Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks,” arXiv preprint arXiv:1511.06434 (2015).



Deep Convolutional GANs

Here are the main guidelines they proposed for building stable convolutional GANs:

1) Replace any **pooling layers** with **strided convolutions** (in **the discriminator**) and **transposed convolutions** (in **the generator**).

2) Use **Batch Normalization** in both **the generator** and **the discriminator**, except in the generator's output layer and the discriminator's input layer.

3) **Remove fully connected hidden layers** for deeper architectures.

4) Use **ReLU** activation in **the generator** for all layers except the output layer, which should use **tanh**.

5) Use **leaky ReLU** activation in **the discriminator** for all layers.

Example: Preparing The Dataset cifar10

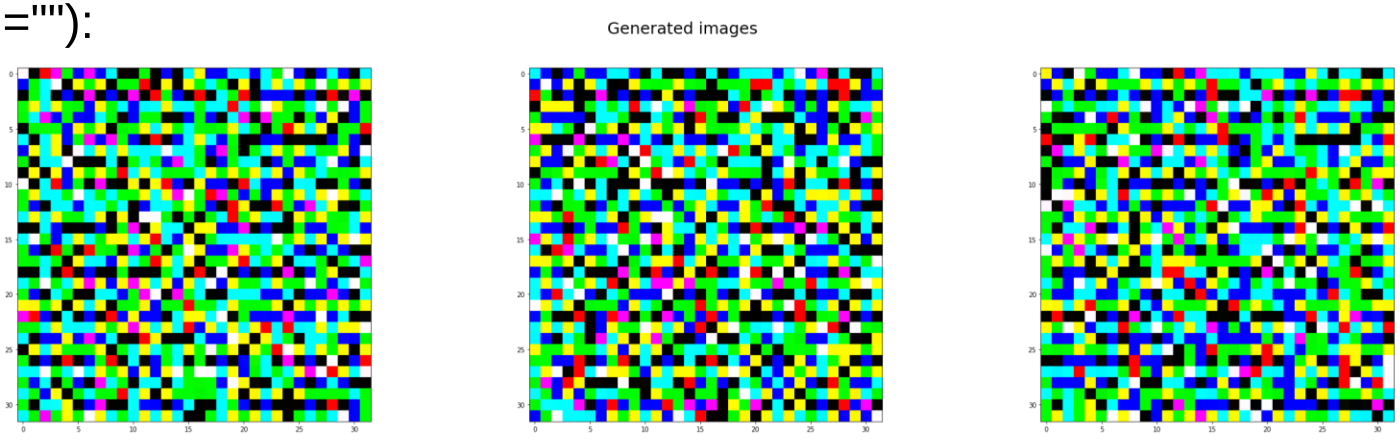
```
import tensorflow as tf
from tensorflow import keras
import matplotlib.pyplot as plt
import numpy as np

# Using Keras to load the dataset
(X_train, y_train), (X_test, y_test) = keras.datasets.cifar10.load_data()
print("X_train shape = ",X_train.shape," X_test shape = ",X_test.shape)

fig = plt.figure()
for i in range(9):
  plt.subplot(3,3,i+1)
  plt.tight_layout()
  plt.imshow(X_train[i], cmap='gray', interpolation='none')
  plt.xticks([])
  plt.yticks([])

# Scale the pixel intensities down to the [0,1] range by dividing them by 255.0
X_train = X_train.astype("float32") / 255.0

# Creating a Dataset to iterate through the images
batch_size = 128
dataset = tf.data.Dataset.from_tensor_slices(X_train).shuffle(1000)
dataset = dataset.batch(batch_size, drop_remainder=True).prefetch(1)
```

Example: The Generator

```
# codings_size : the dimension of the input vector for the generator
codings_size = 100

def build_generator(codings_size=100):
    generator = tf.keras.Sequential()

    # latent variable as input
    generator.add(keras.layers.Dense(1024, activation="relu", input_shape=(codings_size,)))
    generator.add(keras.layers.BatchNormalization())
    generator.add(keras.layers.Dense(1024, activation="relu"))
    generator.add(keras.layers.BatchNormalization())
    generator.add(keras.layers.Dense(128*8*8, activation="relu"))
    generator.add(keras.layers.Reshape((8, 8, 128)))
    assert generator.output_shape == (None, 8, 8, 128)  # Note: None is the batch size

    generator.add(keras.layers.Conv2DTranspose(filters=128, kernel_size=2, strides=2, activation="relu", padding="same"))
    assert generator.output_shape == (None, 16, 16, 128)
    generator.add(keras.layers.BatchNormalization())

    generator.add(keras.layers.Conv2DTranspose(filters=3, kernel_size=2, strides=2, activation="tanh", padding="same"))
    assert generator.output_shape == (None, 32, 32, 3)

    return generator
```

Example: The Generator plot generated images

```
generator = build_generator()
nbr_imgs = 3

def plot_generated_images(nbr_imgs, titleadd=""):
    noise = tf.random.normal([nbr_imgs, 100])
    imgs = generator.predict(noise)

    fig = plt.figure(figsize=(40,10))
    for i, img in enumerate(imgs):
        ax = fig.add_subplot(1,nbr_imgs,i+1)
        ax.imshow((img * 255).astype(np.uint8))
    fig.suptitle("Generated images"+titleadd,fontsize=25)
    plt.show()

plot_generated_images(nbr_imgs)
```



In the beginning, the generator generates random pictures.

Example: The Discriminator

```
# discriminator
def build_discriminator():
    discriminator = tf.keras.Sequential()

    discriminator.add(keras.layers.Conv2D(filters=64, kernel_size=3, strides=2,
            activation=keras.layers.LeakyReLU(0.2), padding="same", input_shape=(32, 32, 3)))
    discriminator.add(keras.layers.Conv2D(filters=128, kernel_size=3, strides=2,
            activation=keras.layers.LeakyReLU(0.2), padding="same"))
    discriminator.add(keras.layers.Conv2D(filters=128, kernel_size=3, strides=2,
            activation=keras.layers.LeakyReLU(0.2), padding="same"))
    discriminator.add(keras.layers.Conv2D(filters=256, kernel_size=3, strides=2,
            activation=keras.layers.LeakyReLU(0.2), padding="same"))
    # classifier
    discriminator.add(keras.layers.Flatten())
    discriminator.add(keras.layers.Dropout(0.4))
    # discriminator.add(keras.layers.Dense(1024, activation=keras.layers.LeakyReLU(0.2)))
    discriminator.add(keras.layers.Dense(1, activation="sigmoid"))
    return discriminator


discriminator = build_discriminator()
# compile model
opt = keras.optimizers.Adam(lr=0.0002, beta_1=0.5)
discriminator.compile(loss="binary_crossentropy", optimizer=opt, metrics=["accuracy"])
discriminator.trainable = False
```

Example: Train the GAN

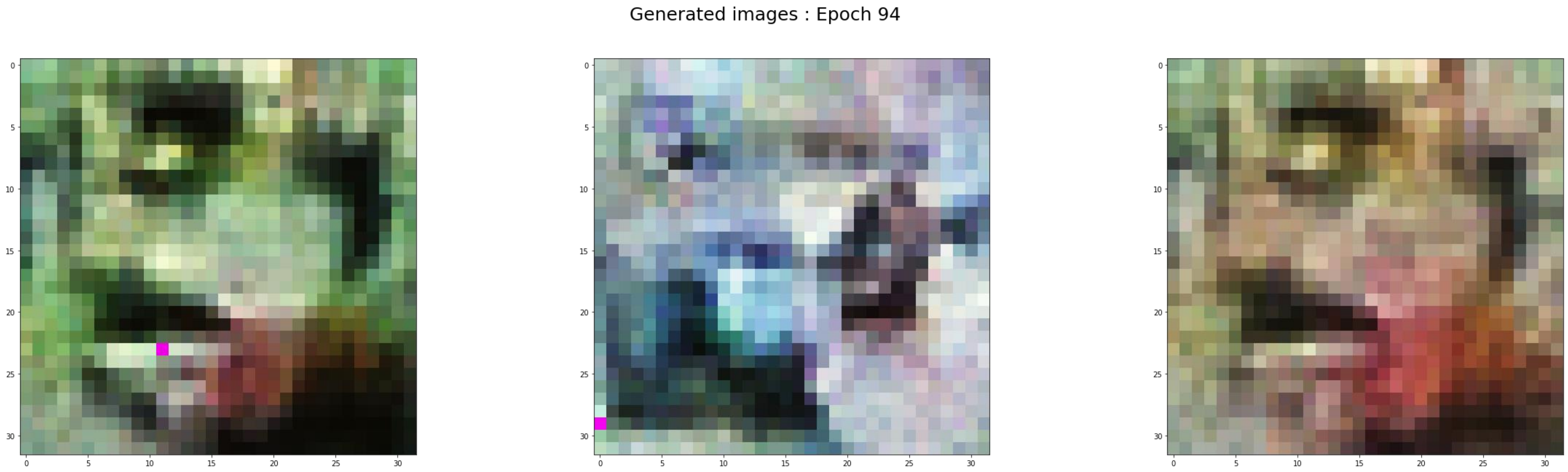
```
gan = keras.models.Sequential([generator, discriminator])
# compile gan
opt = keras.optimizers.Adam(lr=0.0002, beta_1=0.5)
gan.compile(loss="binary_crossentropy", optimizer=opt)

# -----------------------------------------------------------------------------
# For creating an animated gif
from PIL import Image
import cv2
images = []

def animated_gif():
    noise_1 = tf.random.normal(shape=[4, codings_size])
    imgs = generator.predict(noise_1)

    img0 = (imgs[0] * 255).astype(np.uint8)
    img1 = (imgs[1] * 255).astype(np.uint8)
    img2 = (imgs[2] * 255).astype(np.uint8)
    img3 = (imgs[3] * 255).astype(np.uint8)

    img = cv2.hconcat([img0, img1, img2, img3])
    img = Image.fromarray(np.uint8(img)).convert("RGB")
    return img
```

Example: Train the GAN - The Training Loop

```
print("------------------------------------------------------------")
def train_gan(gan, dataset, batch_size, codings_size, n_epochs):
    generator, discriminator = gan.layers
    for epoch in range(n_epochs):
        for X_batch in dataset:
            # phase 1 - training the discriminator
            noise = tf.random.normal(shape=[batch_size, codings_size])
            generated_images = generator.predict(noise)
            X_fake_and_real = tf.concat([generated_images, X_batch], axis=0)
            y1 = tf.constant([[0.]] * batch_size + [[1.]] * batch_size)
            discriminator.trainable = True
           d_loss_accuracy = discriminator.train_on_batch(X_fake_and_real, y1)

            # phase 2 - training the generator
            noise = tf.random.normal(shape=[batch_size, codings_size])
            y2 = tf.constant([[1.]] * batch_size)
            discriminator.trainable = False
            g_loss = gan.train_on_batch(noise, y2)

        print("epoch : ",epoch, " d_loss_accuracy  = ",d_loss, " g_loss = ",g_loss)
        plot_generated_images(3,titleadd=" : Epoch {}".format(epoch))
        # For creating an animated gif
        img = animated_gif()
        images.append(img)
        print("------------------------------------------------------------")
```

Example: Train the GAN

```
n_epochs = 100
train_gan(gan, dataset, batch_size, codings_size, n_epochs)

# Create a gif of the generated images at every epoch
images[0].save("/content/gif_image.gif",
        save_all=True, append_images=images[1:], optimize=False, duration=500, loop=0)
```



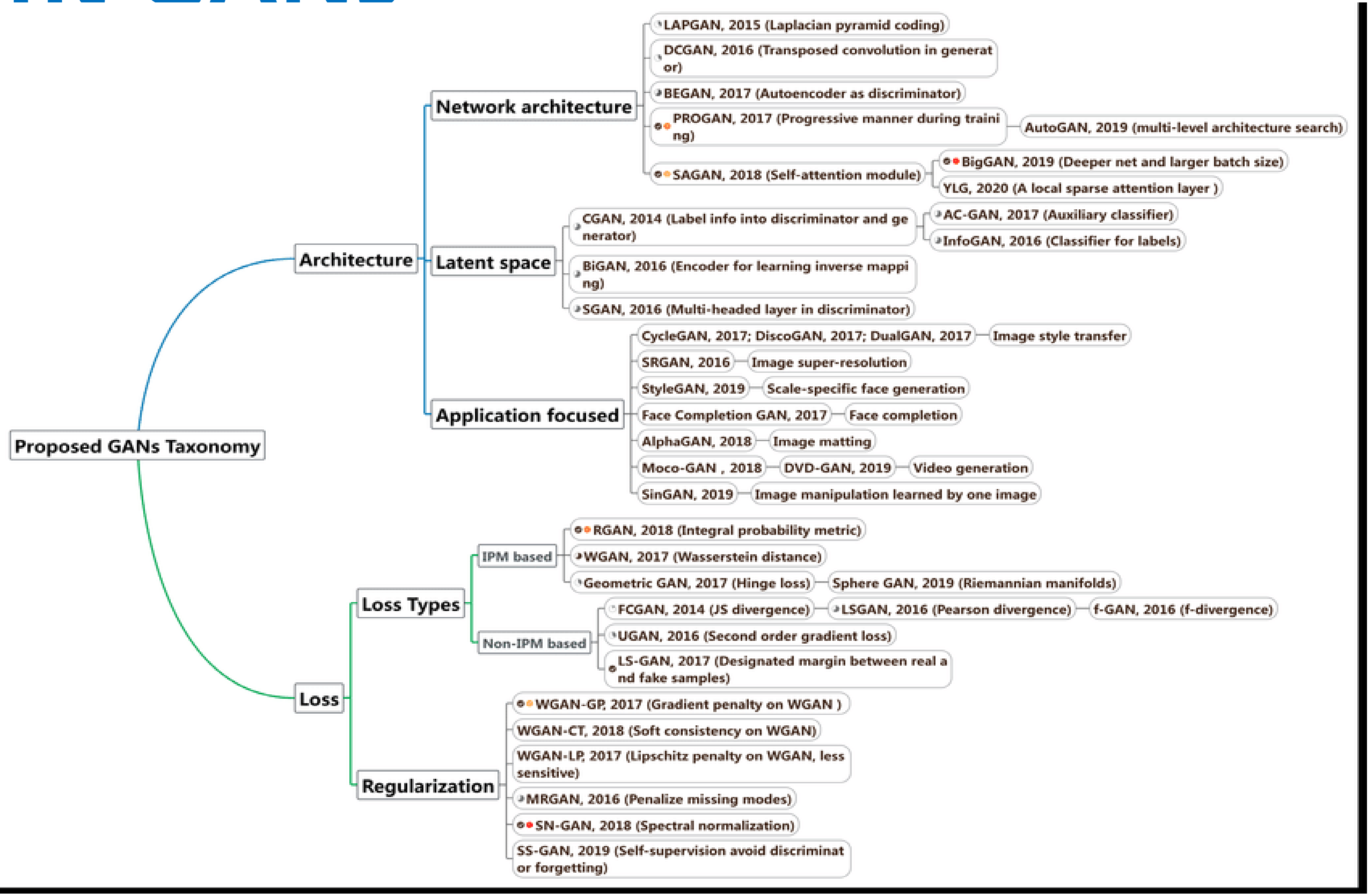
Deep Convolutional GANs:
https://github.com/hichemfelouat/my-codes-of-machine-learning/blob/master/GAN.ipynb

Example: AttGAN

AttGAN - Arbitrary Facial Attribute Editing: Only Change What You Want
https://github.com/elvisyjlin/AttGAN-PyTorch

[Bald, Bangs, Black_Hair, Blond_Hair, Brown_Hair, Bushy_Eyebrows, Eyeglasses, Male, Mouth_Slightly_Open, Mustache, No_Beard, Pale_Skin, Young]

Recent GANs



Proposed GANs Taxonomy
Architecture
Network architecture
LAPGAN, 2015 (Laplacian pyramid coding)
DCGAN, 2016 (Transposed convolution in generator)
BEGAN, 2017 (Autoencoder as discriminator)
PROGAN, 2017 (Progressive manner during training)
AutoGAN, 2019 (multi-level architecture search)
SAGAN, 2018 (Self-attention module)
BigGAN, 2019 (Deeper net and larger batch size)
YLG, 2020 (A local sparse attention layer)
Latent space
CGAN, 2014 (Label info into discriminator and generator)
AC-GAN, 2017 (Auxiliary classifier)
InfoGAN, 2016 (Classifier for labels)
BiGAN, 2016 (Encoder for learning inverse mapping)
SGAN, 2016 (Multi-headed layer in discriminator)
Application focused
CycleGAN, 2017; DiscoGAN, 2017; DualGAN, 2017
Image style transfer
SRGAN, 2016
Image super-resolution
StyleGAN, 2019
Scale-specific face generation
Face Completion GAN, 2017
Face completion
AlphaGAN, 2018
Image matting
Moco-GAN , 2018
DVD-GAN, 2019
Video generation
SinGAN, 2019
Image manipulation learned by one image
Loss
Loss Types
IPM based
RGAN, 2018 (Integral probability metric)
WGAN, 2017 (Wasserstein distance)
Geometric GAN, 2017 (Hinge loss)
Sphere GAN, 2019 (Riemannian manifolds)
Non-IPM based
FCGAN, 2014 (JS divergence)
LSGAN, 2016 (Pearson divergence)
f-GAN, 2016 (f-divergence)
UGAN, 2016 (Second order gradient loss)
LS-GAN, 2017 (Designated margin between real and fake samples)
Regularization
WGAN-GP, 2017 (Gradient penalty on WGAN)
WGAN-CT, 2018 (Soft consistency on WGAN)
WGAN-LP, 2017 (Lipschitz penalty on WGAN, less sensitive)
MRGAN, 2016 (Penalize missing modes)
SN-GAN, 2018 (Spectral normalization)
SS-GAN, 2019 (Self-supervision avoid discriminator forgetting)

Recent GANs

1

Generative Adversarial Networks in Computer Vision: A Survey and Taxonomy

Zhengwei Wang, Qi She, Tomás E. Ward

Abstract

Generative adversarial networks (GANs) have been extensively studied in the past few years. Arguably their most significant impact has been in the area of computer vision where great advances have been made in challenges such as plausible image generation, image-to-image translation, facial attribute manipulation and similar domains. Despite the significant successes achieved to date, applying GANs to real-world problems still poses significant challenges, three of which we focus on here. These are: (1) the generation of high quality images, (2) diversity of image generation, and (3) stable training. Focusing on the degree to which popular GAN technologies have made progress against these challenges, we provide a detailed review of the state of the art in GAN-related research in the published scientific literature. We further structure this review through a convenient taxonomy we have adopted based on variations in GAN architectures and loss functions. While several reviews for GANs have been presented to date, none have considered the status of this field based on their progress towards addressing practical challenges relevant to computer vision. Accordingly, we review and critically discuss the most popular architecture-variant, and loss-variant GANs, for tackling these challenges. Our objective is to provide an overview as well as a critical analysis of the status of GAN research in terms of relevant progress towards important computer vision application requirements. As we do this we also discuss the most compelling applications in computer vision in which GANs have demonstrated considerable success along with some suggestions for future research directions. Code

https://arxiv.org/abs/1906.01529

GANs in NLP

In this paper, the author **explores the uses of GAN** in this **NLP** task and **proposed a GAN** architecture that does the same.

https://arxiv.org/abs/1905.01976

TextKD-GAN: Text Generation using Knowledge Distillation and Generative Adversarial Networks

Md. Akmal Haidar and Mehdi Rezagholizadeh
{md.akmal.haidar, mehdi.rezagholizadeh}@huawei.com

Huawei Noah's Ark Lab, Montreal Research Center, Montreal, Canada

Abstract. Text generation is of particular interest in many NLP applications such as machine translation, language modeling, and text summarization. Generative adversarial networks (GANs) achieved a remarkable success in high quality image generation in computer vision, and recently, GANs have gained lots of interest from the NLP community as well. However, achieving similar success in NLP would be more challenging due to the discrete nature of text. In this work, we introduce a method using knowledge distillation to effectively exploit GAN setup for text generation. We demonstrate how autoencoders (AEs) can be used for providing a continuous representation of sentences, which is a smooth representation that assign non-zero probabilities to more than one word. We distill this representation to train the generator to synthesize similar smooth representations. We perform a number of ex-

Thanks For Your Attention

Hichem Felouat …