Alessandro Allegranzi

BU MET CS 767

Assignment 4: GAN

06/05/2024

Table of Contents

[1. How I modified the code to attempt improvement 2](#_Toc168065848)

[2. Your GAN Application (3 pg max) 5](#_Toc168065853)

[2.1 Give 2-4 requirements for an application that you will implement with a GAN 5](#_Toc168065854)

[2.2 Sample I/O 6](#_Toc168065856)

[2.3 The GAN Architecture 6](#_Toc168065858)

[2.3 Key code 6](#_Toc168065860)

[2.4 URL of your Colab code 6](#_Toc168065862)

[References 6](#_Toc168065864)

[Evaluation 6](#_Toc168065865)

[Appendix 1 7](#_Toc168065866)

[Appendix 2 7](#_Toc168065867)

06/05/2023

MET CS 767 Assignment 4: GAN’s

*Alessandro Allegranzi*

At almost every step in the construction of neural nets, there are choices to be made, including the selection of levels and parameters. We have reasons for our selections, but choices must be evaluated empirically.

The remaining instructions are the same as in previous assignments.

# How I modified the code to attempt improvement

Copy the implementation [here](https://colab.research.google.com/drive/129XTL5-89jk93Cr8UwZA1SSbKDB8fLe8?usp=sharing) to your Google drive. Systematically modify (e.g., add to or remove from) this code or data, attempting to improve the output, and report the results. If necessary, you can show changes that make the result worse, with your explanation.

## 1.1 Description of what you did and reason this *could reasonably be* an improvement (up to two paragraphs, excluding figures and tables)

1. Add more layers to generator and discriminator
2. Use modified activation function
3. Use Gaussian Weight Initialization

# Adding 2 more layers to the generator with Gaussian Weight Initialization

model.add(layers.Conv2DTranspose(256, (5, 5), strides=(1, 1), padding='same', kernel\_initializer=RandomNormal(mean=0.0, stddev=0.02), use\_bias=False))

assert model.output\_shape == (None, 7, 7, 256)

model.add(layers.BatchNormalization())

model.add(layers.LeakyReLU(alpha=0.2))

model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', kernel\_initializer=RandomNormal(mean=0.0, stddev=0.02), use\_bias=False))

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

model.add(layers.BatchNormalization())

model.add(layers.LeakyReLU(alpha=0.2))

# Adding one more layer to the discriminator

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

model.add(layers.LeakyReLU())

model.add(layers.Dropout(0.3))

Adding more layers to the generator and discriminator can increase the capacity of the models, allowing them to learn more complex representations [1]. This can potentially improve the quality of the generated images and the ability of the discriminator to distinguish between real and fake images. The LeakyReLU activation function with a negative slope of 0.2 is used here instead of the default value of 0.3. LeakyReLU allows a small gradient when the unit is not active, which can help to alleviate the dying ReLU problem where neurons can sometimes get stuck in the non-active state and stop learning [2]. This can lead to improved training dynamics and better performance. Initializing the weights of the convolutional layers from a Gaussian distribution with mean 0 and standard deviation 0.02, which is a common practice in DCGANs, helps to ensure that all neurons start off in a similar state, promoting more symmetric breaking of the initial symmetry. This can lead to a more stable training process and better performance [2].

1. Use Adam Stochastic Gradient Descent

# Using Adam version of stochastic gradient descent with the learning rate of 0.0002

# and the beta1 momentum value of 0.5 instead of the default of 0.9.

generator\_optimizer = tf.keras.optimizers.Adam(learning\_rate=0.0002, beta\_1=0.5)

discriminator\_optimizer = tf.keras.optimizers.Adam(learning\_rate=0.0002, beta\_1=0.5)

Adam is an adaptive learning rate optimization algorithm that's been shown to work well in practice and outperform other stochastic optimization methods in many tasks [2]. It adapts the learning rate for each weight individually, based on the first and second moments of the gradients. The original Notebook code has been altered to use a smaller learning rate and a smaller momentum term than the default 0.9 to make the training process more stable and improve the quality of the generated images [2].

*\*Use Label Smoothing*

# Adding label smoothing.

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

real\_loss = cross\_entropy(tf.ones\_like(real\_output) \* 0.9, real\_output)

fake\_loss = cross\_entropy(tf.zeros\_like(fake\_output) \* 0.1, fake\_output)

total\_loss = real\_loss + fake\_loss

return total\_loss

\* Also tried using label smoothing, but then reverted those changes as the final output after the 50 training epochs looked worse with the smoothing. Label smoothing can theoretically help improve the performance of the discriminator by preventing it from becoming too confident in its predictions [2].

## 1.2 Comparison of the result with the original output, with explanation

Original Output:

Sample images printed out by the original model. Some of the look plausibly like real numbers, others, like the top left or the entire third row, seem to miss the mark.

A group of white letters in black squares

Description automatically generated

New Output:

A number in a square

Description automatically generated with medium confidence

The new output contains more entries that look like real numbers.

## 1.3 URL of your Colab code

[Colab Notebook](https://colab.research.google.com/drive/1EC-D9CbVfe6oMPqFNqQ1uDcp2QYHqLZC?usp=sharing)

### >>AI generation for section 1 (or check: I did not use AI generation here \_X\_). Please collapse this.

PARAGRAPH DESCRIBING YOUR VALUE ADDED TO AI-GENERATED MATERIAL

Your response replaces this.

YOUR PROMPT SEQUENCE

[1] Your first prompt replaces this.

[2]

# 2. Your GAN Application (3 pg max)

## 2.1 Give 2-4 requirements for an application that you will implement with a GAN

## These describe *what* functionality your application will provide for the user, including the nature of inputs and outputs. This section should not include *how* you will design or code the application.

**Celebrity Face Generation GAN**

**Functionality:**

1. **Data Loading and Preprocessing:** The application should load the Kaggle CelebA dataset and sample it.
2. **Model Building:** The application should build a GAN model ready to train for image creation.
3. **Training:** The application should be trained on the sampled CelebA dataset.
4. **Face Generation**: The application should be able to generate new, realistic faces of celebrities that do not exist in the training dataset.

**Input:**

The input to the application would be the Kaggle CelebA dataset, which contains over 200,000 celebrity images with 40 attribute labels. The images are all centered on the face and have the same size. The attribute labels include things like "Arched\_Eybrows” and “attractive.” [3] For the actual implementation I cut down to 10,000 images to keep data size manageable, and since we are training an image generator I did not use the labels.

**Output:**

The output of the application would be entirely new generated images of celebrity-like faces.

## 2.2 Sample I/O

## Give three varied input/outputs pairs for your implemented application.

Input: Training

Output: The label 0 (T-shirt/top) receiving the highest probability.

Input: A 28x28 grayscale image of a Sneaker.

Output: The label 7 (Sneaker) receiving the highest probability.

Input: A 28x28 grayscale image of a Bag.

Output: The label 8 (Bag) receiving the highest probability.

## 2.3 The GAN Architecture

## Show your architecture in one or more annotated figures.

The GAN consists of two models, the generator and discriminator.

**Generator**

Model: "sequential\_8"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

dense\_8 (Dense) (None, 16384) 1638400

batch\_normalization\_24 (Ba (None, 16384) 65536

tchNormalization)

leaky\_re\_lu\_27 (LeakyReLU) (None, 16384) 0

reshape\_7 (Reshape) (None, 8, 8, 256) 0

conv2d\_transpose\_24 (Conv2 (None, 16, 16, 128) 819200

DTranspose)

batch\_normalization\_25 (Ba (None, 16, 16, 128) 512

tchNormalization)

leaky\_re\_lu\_28 (LeakyReLU) (None, 16, 16, 128) 0

conv2d\_transpose\_25 (Conv2 (None, 32, 32, 64) 204800

DTranspose)

batch\_normalization\_26 (Ba (None, 32, 32, 64) 256

tchNormalization)

leaky\_re\_lu\_29 (LeakyReLU) (None, 32, 32, 64) 0

conv2d\_transpose\_26 (Conv2 (None, 64, 64, 3) 4800

DTranspose)

=================================================================

Total params: 2733504 (10.43 MB)

Trainable params: 2700352 (10.30 MB)

Non-trainable params: 33152 (129.50 KB)

A diagram of a computer program

Description automatically generated with medium confidence

**Discriminator**

Model: "sequential\_9"

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Layer (type) Output Shape Param #

=================================================================

conv2d\_3 (Conv2D) (None, 32, 32, 64) 4864

leaky\_re\_lu\_30 (LeakyReLU) (None, 32, 32, 64) 0

dropout\_3 (Dropout) (None, 32, 32, 64) 0

conv2d\_4 (Conv2D) (None, 16, 16, 128) 204928

leaky\_re\_lu\_31 (LeakyReLU) (None, 16, 16, 128) 0

dropout\_4 (Dropout) (None, 16, 16, 128) 0

conv2d\_5 (Conv2D) (None, 8, 8, 128) 409728

leaky\_re\_lu\_32 (LeakyReLU) (None, 8, 8, 128) 0

dropout\_5 (Dropout) (None, 8, 8, 128) 0

conv2d\_6 (Conv2D) (None, 4, 4, 128) 409728

leaky\_re\_lu\_33 (LeakyReLU) (None, 4, 4, 128) 0

dropout\_6 (Dropout) (None, 4, 4, 128) 0

flatten\_1 (Flatten) (None, 2048) 0

dense\_9 (Dense) (None, 1) 2049

=================================================================

Total params: 1031297 (3.93 MB)

Trainable params: 1031297 (3.93 MB)

Non-trainable params: 0 (0.00 Byte)

A diagram of a data flow

Description automatically generated with medium confidence

## 2.3 Key code

## Provide snippets of the essential core code of your implementation.

**Generator Model Definition:**

def make\_generator\_model():

model = tf.keras.Sequential()

model.add(layers.Dense(8\*8\*256, use\_bias=False, input\_shape=(100,)))

model.add(layers.BatchNormalization())

model.add(layers.LeakyReLU(alpha=0.2))

model.add(layers.Reshape((8, 8, 256)))

assert model.output\_shape == (None, 8, 8, 256)

model.add(layers.Conv2DTranspose(128, (5, 5), strides=(2, 2), padding='same', kernel\_initializer=RandomNormal(mean=0.0, stddev=0.02), use\_bias=False))

assert model.output\_shape == (None, 16, 16, 128)

model.add(layers.BatchNormalization())

model.add(layers.LeakyReLU(alpha=0.2))

model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', kernel\_initializer=RandomNormal(mean=0.0, stddev=0.02), use\_bias=False))

assert model.output\_shape == (None, 32, 32, 64)

model.add(layers.BatchNormalization())

model.add(layers.LeakyReLU(alpha=0.2))

model.add(layers.Conv2DTranspose(3, (5, 5), strides=(2, 2), padding='same', kernel\_initializer=RandomNormal(mean=0.0, stddev=0.02), use\_bias=False, activation='tanh'))

assert model.output\_shape == (None, 64, 64, 3)

return model

**Discriminator Model Definition:**

def make\_discriminator\_model():

model = tf.keras.Sequential()

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

input\_shape=[64, 64, 3]))

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

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

**Loss and Optimizers:**

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

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

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

def generator\_loss(fake\_output):

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

# Using Adam version of stochastic gradient descent with the learning rate of 0.0002

# and the beta1 momentum value of 0.5 instead of the default of 0.9.

generator\_optimizer = tf.keras.optimizers.Adam(learning\_rate=0.0002, beta\_1=0.5)

discriminator\_optimizer = tf.keras.optimizers.Adam(learning\_rate=0.0002, beta\_1=0.5)

Training Code:

Sample Output:

Original untrained generated image

A green and yellow pixelated square

Description automatically generated

After 50 training epochs these were some generated images

A blurry image of a person's face

Description automatically generated

The output is not perfect, but it certainly looks like it is approaching human faces. Compared to the original generated image the results look much better. Given more time and processing power, the application could likely reach much more realistic generated images.

## 2.4 URL of your Colab code

[Colab Notebook Link](https://colab.research.google.com/drive/1avkSAWy50x0oAfSN3JGgSu2ahzMtPAQh?usp=sharing)

### >>AI generation for section 2 (or check: I did not use AI generation here \_X\_). Please collapse this.

PARAGRAPH DESCRIBING YOUR VALUE ADDED TO AI-GENERATED MATERIAL

Your response replaces this.

YOUR PROMPT SEQUENCE

[1] Your first prompt replaces this.

[2]

Your response replaces this.

# References

Show that you used a wide variety of resources by listing them below and clearly indicating in the body above where you used. Make sure to use proper referencing in your paper. We suggest using APA format, but other formats are fine as long as they clearly distinguish your work from work of others in your response. In general, observe the stated plagiarism rules.

[1] Geron, Aurelien. *Hands-On Machine Learning with Scikit-learn, Keras, & Tensorflow Third Edition*. “Chapter 17: Autoencoders, GANs, and Diffusion Models”. O’Reilly Media, Inc. October, 2022

# [2] Brownlee, Jason. *“*How to Implement GAN Hacks in Keras to Train Stable Models.” *Machine Learning Mastery*. https://machinelearningmastery.com/how-to-code-generative-adversarial-network-hacks/

[3] “50k Celeba Dataset 64x64.” *Kaggle. https://www.kaggle.com/datasets/therealcyberlord/50k-celeba-dataset-64x64*

# Evaluation



# Appendix 1

…

# Appendix 2

…