

Working with a Generative

Adversarial Network

Javier A. Diaz Velazquez

Colorado State University Global

CSC580: Applying Machine Learning and
Neural Networks

Dr. Issa

September 10, 2023

Generative Adversarial Network

The goal of the analysis was that after researching information about the Generative Adversarial Network (GAN) or Generative – AI, it must be implemented to analyze and compare its behavior and performance. Per the assignment's instructions, the GAN was implemented using the CIFAR 10 dataset, consisting of over 60,000 32 x 32 color images in ten different classes with 6,000 images per class. The main idea was to gauge and benchmark the performance of the combined neural networks that compose the GAN architecture, which is composed of two convolutional neural networks known as the generator, which trains to generate fake data or images with noise added to it and a discriminator which is trained to identify and classify the original data from the fake data. The training process or technique is performed in an adversarial mode, meaning that each neural network competes against the other; this process happens iteratively until each network improves.

GAN_V1 Analysis

The analysis of the Generative Adversarial network (GAN) was performed using two different hardware infrastructures in order to benchmark the runtime execution of the algorithm; the first couple of tests were performed by utilizing CPU compute power, the second couple of tests were performed using GPU compute power. However, the GPU test was performed using Google Collaboratory due to the computational demand that this particular algorithm requires. When mentioning a "couple of tests," it means that this particular implementation took much time and debugging to get at least to execute. The runtime and training process for the algorithm took roughly 1 hour using the CPU, while the training with the GPU took around 35 minutes; these tests were performed with the original hyperparameters of 15000 epochs, 32 batch sizes, at 2500 display intervals.

Image 1 – 1 to 1 – 4 shows the Generative Adversarial Network (GAN) algorithm implementation, CPU execution runtime, and the generated output results.

```

37 # Input shape.
38 image_shape = (32, 32, 3)
39 latent_dims = 100
40
41 # Dataset progress bar.
42 for progress_2 in tqdm(range(101), desc="Loading Dataset", ascii=False, ncols=75):
43     tm.sleep(0.01)
44
45 msg_1 = f"\nIFAR10 dataset----->>>\n(msg_0)\n\n"
46 print(msg_1)
47 tm.sleep(1)
48
49 # This class houses the generator conv network.
50
51 class build_generator:
52     generator_net = tf.keras.Sequential()
53
54     generator_net.add(tf.keras.layers.Dense(128 * 8 * 8, activation='relu', input_dim=dataset_load.latent_dims))
55     generator_net.add(tf.keras.layers.Reshape((8, 8, 128)))
56     generator_net.add(tf.keras.layers.UpSampling2D())
57
58     generator_net.add(tf.keras.layers.Conv2D(128, kernel_size=3, padding='same'))
59     generator_net.add(tf.keras.layers.BatchNormalization(momentum=0.78))
60     generator_net.add(tf.keras.layers.Activation('relu'))
61     generator_net.add(tf.keras.layers.UpSampling2D())
62
63     generator_net.add(tf.keras.layers.Conv2D(64, kernel_size=3, padding='same'))
64     generator_net.add(tf.keras.layers.BatchNormalization(momentum=0.78))
65     generator_net.add(tf.keras.layers.Activation('relu'))
66
67     generator_net.add(tf.keras.layers.Conv2D(3, kernel_size=3, padding='same'))
68     generator_net.add(tf.keras.layers.Activation('tanh'))
69
70 noise = tf.keras.Input(shape=(dataset_load.latent_dims,))
71 image = generator_net(noise)
72
73 generator_net.summary()
74
75 sent1 = Model(noise, image)
76

```

Figure 1 – 1 shows the generator neural network implementation per the assignment instruction.

```

79
80 class build_discriminator:
81     discriminator_net = tf.keras.Sequential()
82
83     discriminator_net.add(tf.keras.layers.Conv2D(32, kernel_size=3, strides=2, input_shape=dataset_load.image_shape,
84         padding='same'))
85     discriminator_net.add(tf.keras.layers.LeakyReLU(alpha=0.2))
86     discriminator_net.add(tf.keras.layers.Dropout(0.25))
87
88     discriminator_net.add(tf.keras.layers.Conv2D(64, kernel_size=3, strides=2, padding='same'))
89     discriminator_net.add(tf.keras.layers.ZeroPadding2D(padding=((0, 1), (0, 1))))
90     discriminator_net.add(tf.keras.layers.BatchNormalization(momentum=0.82))
91     discriminator_net.add(tf.keras.layers.LeakyReLU(alpha=0.2))
92     discriminator_net.add(tf.keras.layers.Dropout(0.25))
93
94     discriminator_net.add(tf.keras.layers.Conv2D(128, kernel_size=3, strides=2, padding='same'))
95     discriminator_net.add(tf.keras.layers.BatchNormalization(momentum=0.82))
96     discriminator_net.add(tf.keras.layers.LeakyReLU(alpha=0.2))
97     discriminator_net.add(tf.keras.layers.Dropout(0.25))
98
99     discriminator_net.add(tf.keras.layers.Conv2D(256, kernel_size=3, strides=1, padding='same'))
100    discriminator_net.add(tf.keras.layers.BatchNormalization(momentum=0.8))
101    discriminator_net.add(tf.keras.layers.LeakyReLU(alpha=0.25))
102    discriminator_net.add(tf.keras.layers.Dropout(0.25))
103
104    discriminator_net.add(tf.keras.layers.Flatten())
105    discriminator_net.add(tf.keras.layers.Dense(1, activation='sigmoid'))
106
107    image = tf.keras.Input(shape=dataset_load.image_shape)
108    validity = discriminator_net(image)
109
110    discriminator_net.summary()
111
112    sent2 = Model(image, validity)
113

```

Figure 1 – 2 shows the discriminator neural network implementation per the assignment instruction.

```

Fake Loss: [0.3167489171028137, 0.0]
1/1 [=====] - 0s 85ms/step
Epoch: 550 |
GAN Loss: -1.202944278717041 |
Real Loss: [-5.416862964630127, 0.0] |
Fake Loss: [0.2968427538871765, 0.0]
1/1 [=====] - 0s 81ms/step
Epoch: 551 |
GAN Loss: -1.355901837348938 |
Real Loss: [-5.494165420532227, 0.0] |
Fake Loss: [0.21254515647888184, 0.0]
1/1 [=====] - 0s 69ms/step
Epoch: 552 |
GAN Loss: -1.310649037361145 |
Real Loss: [-5.414976119995117, 0.0] |
Fake Loss: [0.18323540687561035, 0.0]
1/1 [=====] - 0s 77ms/step
Epoch: 553 |
GAN Loss: -1.437786340713501 |
Real Loss: [-5.547097682952881, 0.0] |
Fake Loss: [0.19069451093673706, 0.0]
1/1 [=====] - 0s 65ms/step

```

Figure 1 – 3 shows the algorithm runtime with the computer's CPU, which took over an hour to complete.

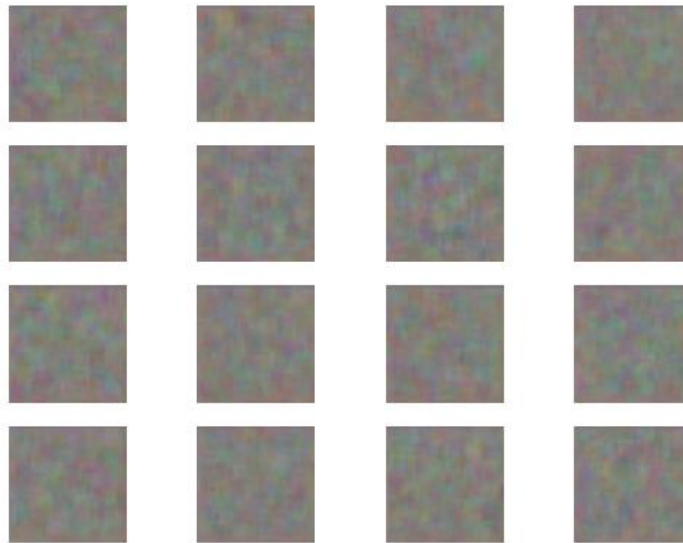


Figure 1 – 4 shows the image from the first epoch. The 16 images are the first output generated when the GAN begins training. Although that part of the implantation was provided, the output results were the same in both cases with the CPU and GPU runtime.

The final generated images during epoch 14999 performed with the CPU were odd because the expectation was that the generator neural network would generate fake images out of the real for the generator to distinguish real from fake. Such results could be impacted by the computational cost of the GAN, which the CPU could not provide, causing Windows to crash three times during runtime. Images 1 – 5 show the generated images from the last epoch.

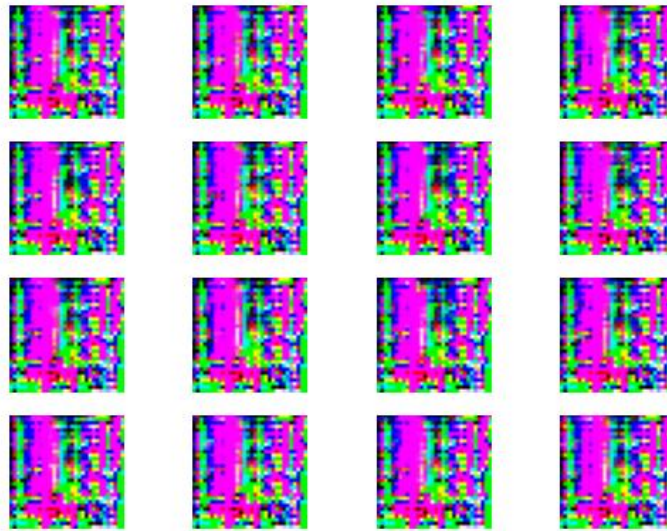
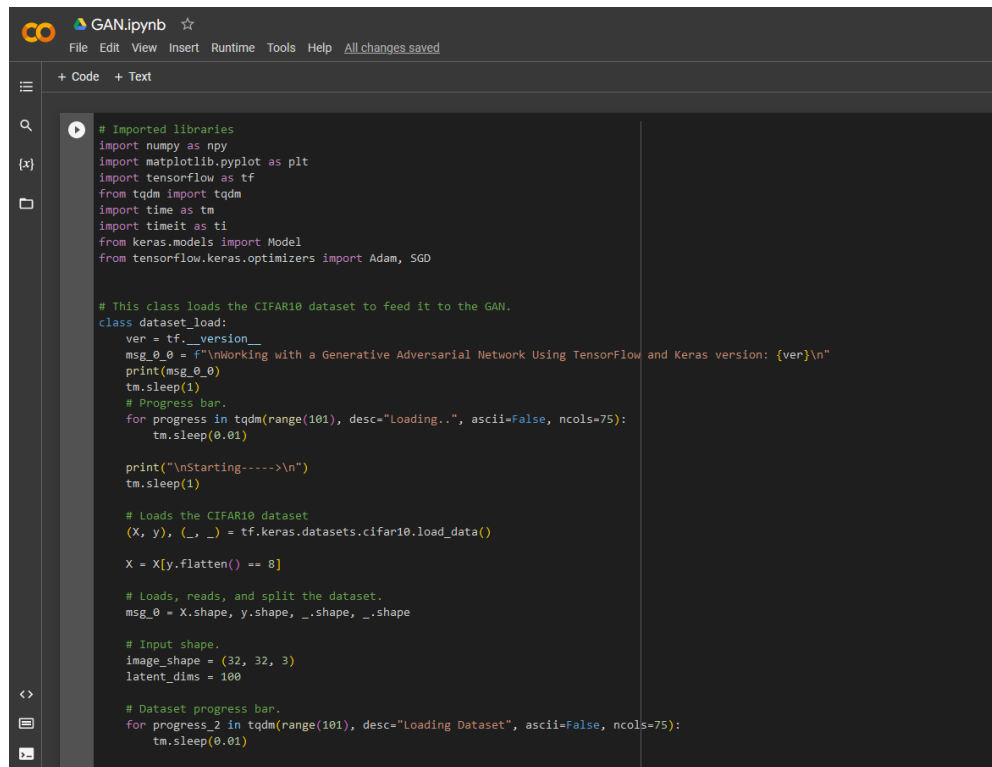


Figure 1 – 5 shows the last images from the last epoch 14999. A blur of color that might be due to the high computational demand.

GAN GPU Runtime Analysis

The following portion of the analysis covers the implementation and execution of the Generative Adversarial Network (GAN) utilizing Google Colab to achieve better performance with GPU compute power. However, despite attempting multiple hyperparameter settings and configurations, the images generated in the last epoch do not resemble an image; they are just a blur of multiple colors. Figure 2 – 1 to 2 – 3 shows the algorithm implementation with GPU and runtime results.



```

GAN.ipynb
File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

# Imported libraries
import numpy as npy
import matplotlib.pyplot as plt
import tensorflow as tf
from tqdm import tqdm
import time as tm
import time as ti
from keras.models import Model
from tensorflow.keras.optimizers import Adam, SGD

# This class loads the CIFAR10 dataset to feed it to the GAN.
class dataset_load:
    ver = tf.__version__
    msg_0_0 = f"\nWorking with a Generative Adversarial Network Using TensorFlow and Keras version: {ver}\n"
    print(msg_0_0)
    tm.sleep(1)
    # Progress bar.
    for progress in tqdm(range(101), desc="Loading..", ascii=False, ncols=75):
        tm.sleep(0.01)

    print("\nStarting---->\n")
    tm.sleep(1)

    # Loads the CIFAR10 dataset
    (X, y), (_, _) = tf.keras.datasets.cifar10.load_data()

    X = X[y.flatten() == 8]

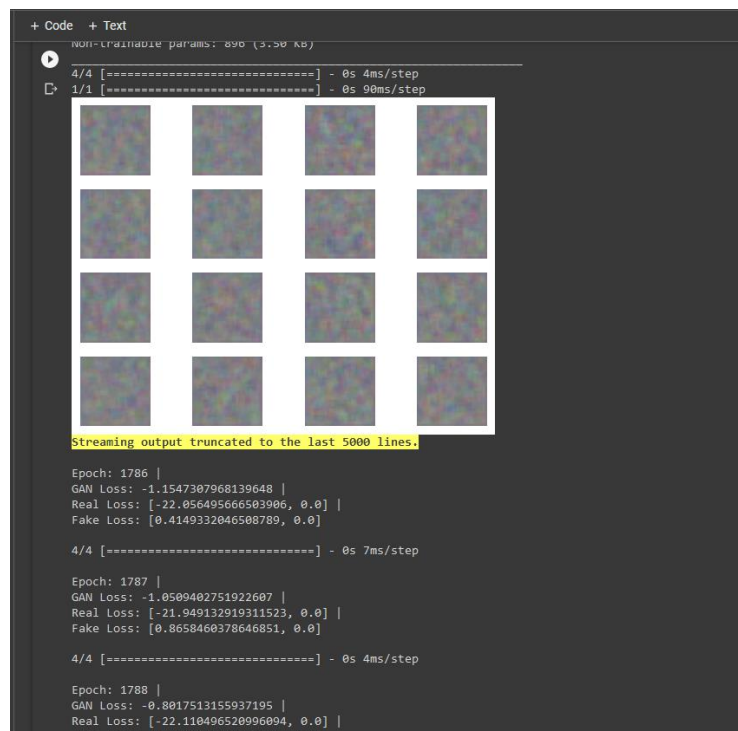
    # Loads, reads, and split the dataset.
    msg_0 = X.shape, y.shape, _,_.shape, _,_.shape

    # Input shape.
    image_shape = (32, 32, 3)
    latent_dims = 100

    # Dataset progress bar.
    for progress_2 in tqdm(range(101), desc="Loading Dataset", ascii=False, ncols=75):
        tm.sleep(0.01)

```

Figure 2 – 1 shows the GAN implementation in Google Colab to utilize GPU computing and debug the learning pattern of the algorithm.



```

+ Code + Text

non-trainable params: 890 (3.30 Kb)

4/4 [=====] - 0s 4ms/step
1/1 [=====] - 0s 90ms/step

Streaming output truncated to the last 5000 lines.

Epoch: 1786 |
GAN Loss: -1.1547307968139648 |
Real Loss: [-22.056495666503906, 0.0] |
Fake Loss: [0.4149332046508789, 0.0]

4/4 [=====] - 0s 7ms/step

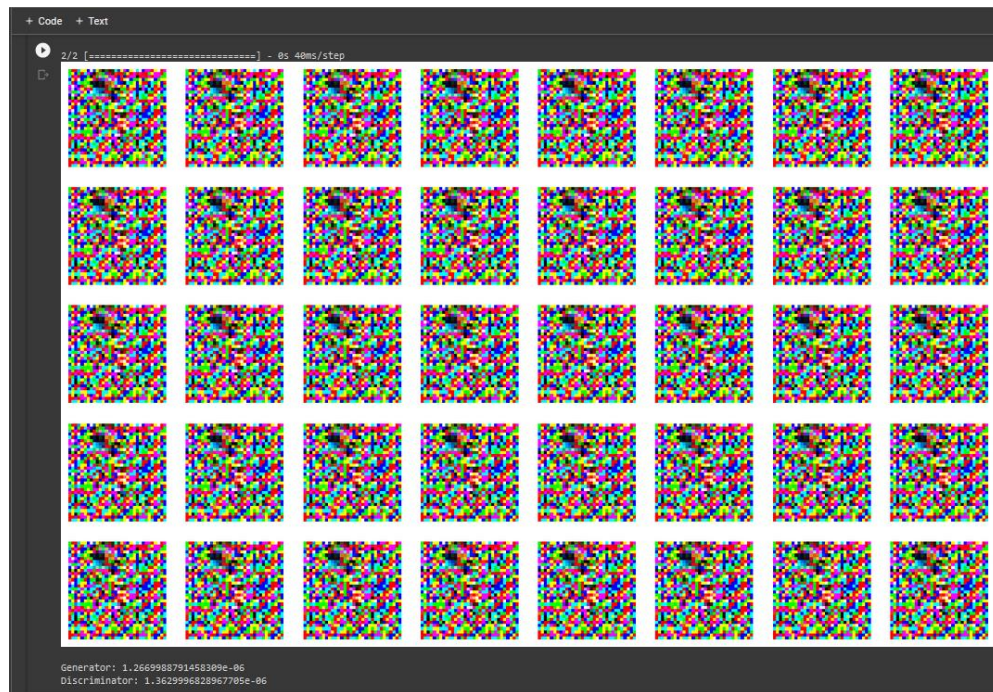
Epoch: 1787 |
GAN Loss: -1.0509402751922607 |
Real Loss: [-21.949132910311523, 0.0] |
Fake Loss: [0.8658460378646851, 0.0]

4/4 [=====] - 0s 4ms/step

Epoch: 1788 |
GAN Loss: -0.8017513155937195 |
Real Loss: [-22.110496520996094, 0.0] |
Fake Loss: [0.5731037500000001, 0.0]

```

Figure 2 – 2 shows the same results when the GAN generated the image of the first epoch, which means that something is wrong with the pattern in which these two networks are being trained.



Figures 2 – 3 showed the same results when the GAN generated the image of the first epoch, which meant that something was wrong with the pattern in which these two networks were being trained. It also entails that something must be worn with the pipeline in which the data passes through both neural networks; a different implementation might be needed.

GAN_V2 Analysis

Since the current implementation was not working correctly and all resources and ideas of what was causing the abnormality were exhausted, a different pipeline configuration for the GAN needed to be implemented. The second implementation consisted of dividing the batches into mini-batches at a rate of 128 batches at 200 epochs; this method pauses the discriminator's weights throughout the learning process while training the GAN to update the generator, and then the generator learns to produce images that are increasingly difficult for the discriminator to distinguish from authentic images. Figure 3 – 1 to 3 – 3 shows the runtime results of the second Generative Adversarial Network (GAN) implementation.

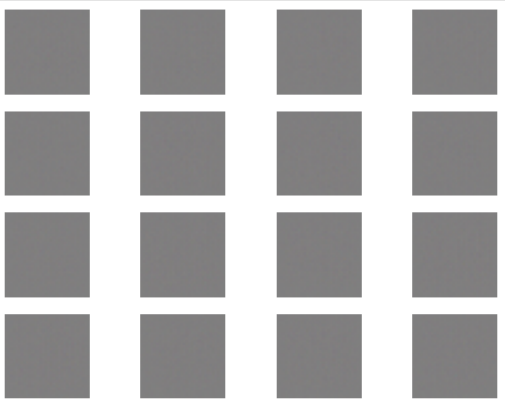

```

Working with a Generative Adversarial Network Using TensorFlow and Keras version: 2.13.0
Loading..: 100%|████████████████████████████████████████| 101/101 [00:01<00:00, 97.78it/s]

Starting---->

Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 [=====] - 14s 0us/step
Loading Dataset: 100%|████████████████████████████████████████| 101/101 [00:01<00:00, 97.57it/s]

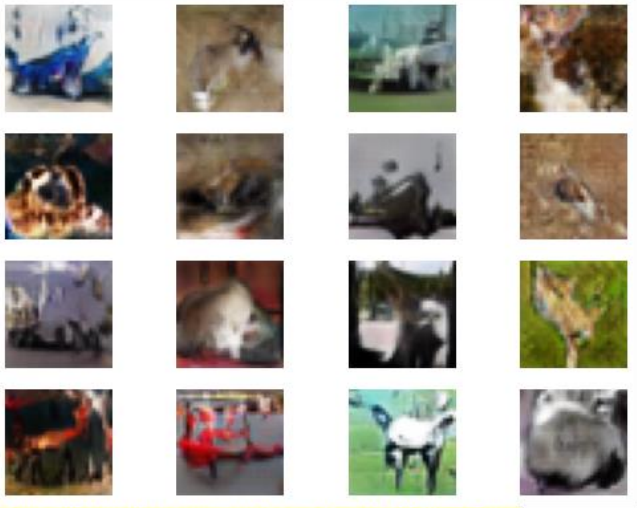
CIFAR10 dataset----->>
((50000, 32, 32, 3), (50000, 1), (10000, 1), (10000, 1))

1/1 [=====] - 8s 8s/step

Streaming output truncated to the last 5000 lines.
2/2 [=====] - 0s 3ms/step

```

Figure 3 – 1 shows the image from the first epoch, but this time, the image quality looks smoother and better than the image generated with the first epoch of the GAN_V1.

```

5/5 [=====] - 0s 2ms/step
Accuracy real: 66%, fake: 87%
WARNING:tensorflow:Compiled the loaded model, but the compiled metrics have already been calculated. This will lead to incorrect metrics in the future.
1/1 [=====] - 0s 24ms/step

Streaming output truncated to the last 5000 lines.
2/2 [=====] - 0s 3ms/step

```

Figure 3 – 2 shows the image generated from epoch 139, which represents a success in the data pipeline of the GAN_V2. Furthermore, the GAN demonstrated an excellent performance distinguishing the images from real to fake.

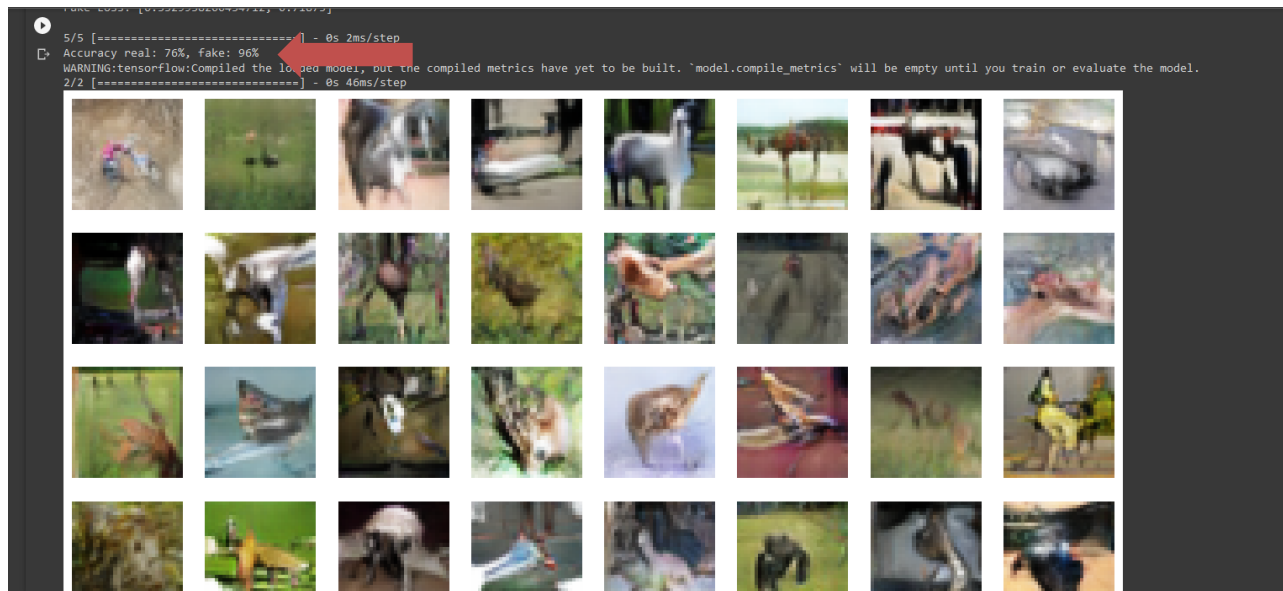


Figure 3 – 3 shows the image generated from the last epoch 199, which demonstrates how the performance of the models gets better during training, allowing the discriminator network to be more accurate based on the images injected by the generator network.

Conclusion

The GAN implementation has demonstrated the complexity and potential of these types of neural networks, which are not only bound to classifying images, but their versatility in generating data and their ability to capture complex patterns make them a valuable tool in various fields. The Generative Adversarial Network has also demonstrated that its performance is as good as the hardware it runs on due to the high computational demand that this network requires for and during training. For instance, during runtime, the GAN requires a minimum of over 32 GB of system RAM to complete the execution until the end; less than 32 GB of system RAM and the system will crash. Ask me how I know!... From the performance standpoint, the GAN has demonstrated that its performance improves while the training advances. Comparing the results from epoch 139 with the final epoch 199 shows a 10% increase when distinguishing the actual images from the fake ones. Finally, this project has been educational, informative, challenging, and full of bugs, making us more efficient in this emerging career of AI/ML.

References

Brownlee J. (2019). How to Develop a GAN to Generate CIFAR10 Small Color Photographs.

<https://machinelearningmastery.com/how-to-develop-a-generative-adversarial-network-for-a-cifar-10-small-object-photographs-from-scratch/>

Gupta A. (2023). Building a Generative Adversarial Network using Keras.

<https://www.geeksforgeeks.org/building-a-generative-adversarial-network-using-keras/>

Sayak P. (2021). Conditional GAN. https://keras.io/examples/generative/conditional_gan/