

GAN

Generadora de animales



Nicolás Boolls
IA 2023

Objetivos

El objetivo de esta GAN será generar imágenes realistas de diversos tipos de animales, utilizando en primer lugar el dataset Animals-10 de Kaggle, con grandes cantidades de datos y luego un dataset recopilado por mi con águilas con una menor cantidad de datos para comparar el funcionamiento de la GAN con menos de mil elementos para generar imágenes. Las imágenes generadas serán de 32x32 píxeles debido al costo computacional y tiempo requerido para entrenar una GAN que genere imágenes de mayor calidad, pero siempre y cuando estos factores lo permitan, se puede incrementar la calidad de estas imágenes con la misma arquitectura.

Introducción

Como todas las GAN, esta cuenta con dos componentes principales: un generador y un discriminador. El generador se ocupará de generar datos sintéticos, falsos, a partir de un espacio de entrada aleatorio (vector de ruido), mientras que el discriminador se ocupará de distinguir entre datos reales, y datos generados por el generador.

Estas dos redes serán entrenadas para que el generador intente engañar al discriminador creando imágenes cada vez más realistas, mientras que el discriminador intenta mejorar su capacidad para distinguir entre imágenes reales y falsas.

Arquitectura

Generador

El generador tiene una capa densa inicial con función de activación LeakyReLU, seguida de un reshape para facilitar la convolución traspuesta. Luego, utilizo tres capas de convolución traspuesta con tamaño de filtro 3x3, un stride de 2 y con padding=same para que se mantenga el tamaño de la salida igual al de la entrada. Por último, utiliza una última capa Conv2D con función de activación tanh para que nuestros resultados queden entre -1 y 1 y se facilite la representación gráfica de la imagen generada.

```
def generador_de_imagenes():
    generador = Sequential()

    generador.add(Dense(256*4*4, input_shape = (100,)))
    #generador.add(BatchNormalization())
    generador.add(LeakyReLU())
    generador.add(Reshape((4,4,256)))

    generador.add(Conv2DTranspose(128,kernel_size=3, strides=2, padding = "same"))
    #generador.add(BatchNormalization())
    generador.add(LeakyReLU(alpha=0.2))

    generador.add(Conv2DTranspose(128,kernel_size=3, strides=2, padding = "same"))
    #generador.add(BatchNormalization())
    generador.add(LeakyReLU(alpha=0.2))

    generador.add(Conv2DTranspose(128,kernel_size=3, strides=2, padding = "same"))
    #generador.add(BatchNormalization())
    generador.add(LeakyReLU(alpha=0.2))

    generador.add(Conv2D(3,kernel_size=3, padding = "same", activation='tanh'))

    return(generador)

modelo_generador = generador_de_imagenes()

modelo_generador.summary()
```

Discriminador

El discriminador tiene inicialmente una capa de convolución 2d que utiliza 64 filtros de tamaño 3x3, con padding para que se mantenga del mismo tamaño la imagen a la entrada y a la salida. Luego, se agregan 3 capas adicionales de convolución con aumentos progresivos en el número de filtros. Cada una de estas capas tiene función de activación LeakyReLU con pendiente 0.2.

Por último, se agrega una capa Flatten para aplanar la salida en un vector unidimensional seguido de una capa Dropout con una tasa del 40% para evitar overfitting.

Y finalmente, una capa densa con una única neurona y activación sigmoide para representar si la imagen es real o falsa.

Se utiliza la función de pérdida de entropía cruzada binaria para calcular la pérdida durante el entrenamiento del modelo.

```

def discriminador_de_imagenes():

    discriminador = Sequential()
    discriminador.add(Conv2D(64, kernel_size=3, padding = "same", input_shape = (32,32,3)))
    discriminador.add(LeakyReLU(alpha=0.2))
    #discriminador.add(Dropout(0.2))

    discriminador.add(Conv2D(128, kernel_size=3, strides=(2,2), padding = "same"))
    discriminador.add(LeakyReLU(alpha=0.2))
    #discriminador.add(Dropout(0.2))

    discriminador.add(Conv2D(128, kernel_size=3, strides=(2,2), padding = "same"))
    discriminador.add(LeakyReLU(alpha=0.2))
    #discriminador.add(Dropout(0.2))

    discriminador.add(Conv2D(256, kernel_size=3, strides=(2,2), padding = "same"))
    discriminador.add(LeakyReLU(alpha=0.2))
    #discriminador.add(Dropout(0.2))

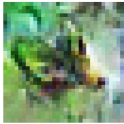


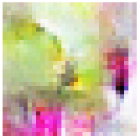





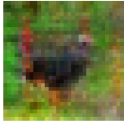












    discriminador.add(Flatten())
    discriminador.add(Dropout(0.4))
    discriminador.add(Dense(1, activation='sigmoid'))

    opt = Adam(lr=0.0002 ,beta_1=0.5)
    discriminador.compile(loss='binary_crossentropy', optimizer= opt , metrics = ['accuracy'])

    return(discriminador)

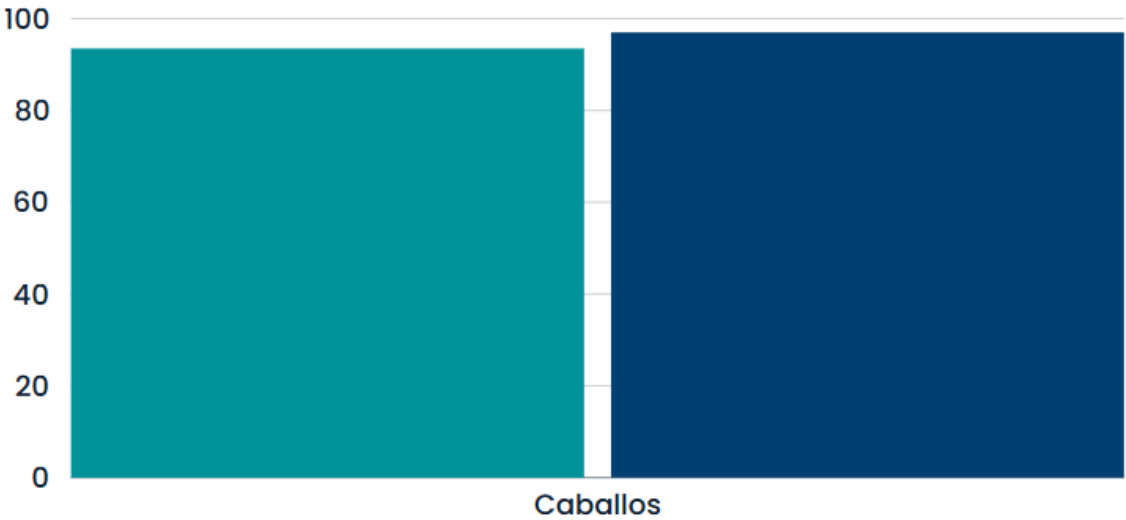
```

Desarrollo

Animal	Epocas				
Mariposas	40				
Arañas	30				
Elefantes	20				
Gallinas	50				
Caballos	50				
Vacas	40				
Águilas (dataset propio)	30				

Resultados

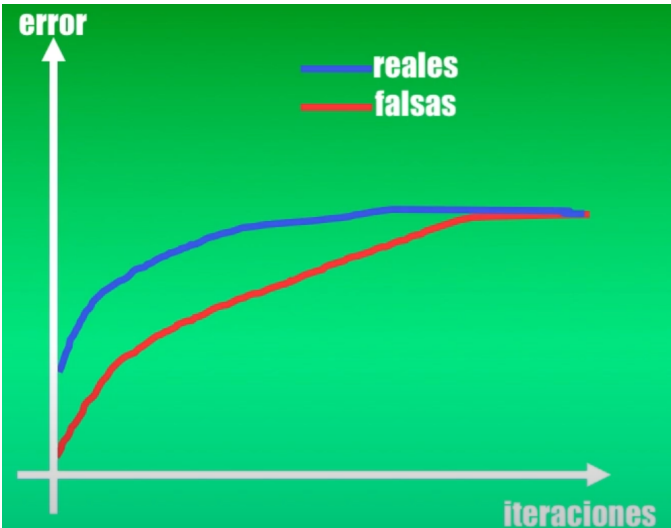
ACCURACY REALES VS FALSOS



Para el ejemplo de la generación de caballos, muestro los resultados (aunque todos tuvieron resultados MUY similares), en el cual se ve que la precisión de los datos reales (Real Accuracy) fue de un 93,5% comparado con un 97% de los datos falsos. Esto quiere decir que el discriminador es bastante preciso para determinar los datos reales, tiene un accuracy bastante alto, y tiene un 97% de precisión para determinar cuáles datos son falsos. Esto no es necesariamente malo considerando el número de épocas que se realizó para este informe, ya que al ser bastante bajo, el generador no se llegó a entrenar lo suficiente para que comience a engañar un poco mas al discriminador. Esto lo hice así porque la capacidad computacional requerida para realizar más épocas en tantas GAN se me iba de las manos y además se ve

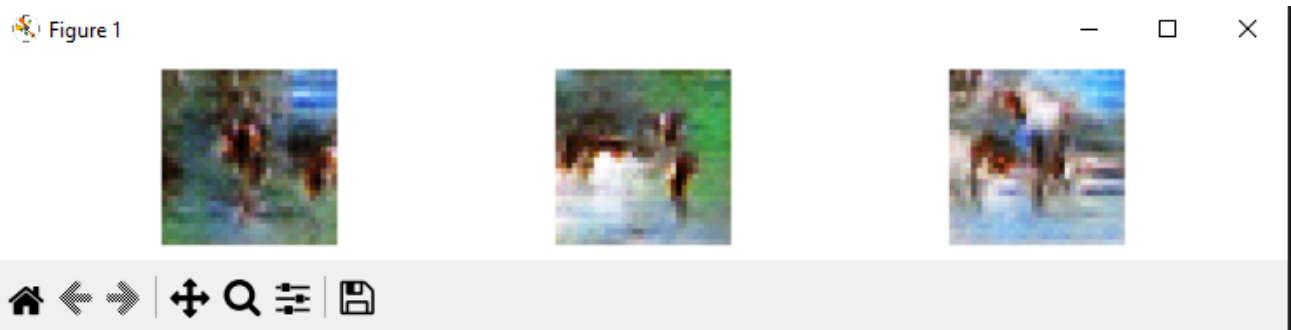
bastante claro en el progreso de las épocas que hay una mejora en la generación de imágenes (tanto como lo permite una imagen de 32x32 pixels).

Teniendo en cuenta que el funcionamiento de una GAN bien diseñada a lo largo de las iteraciones es el de la siguiente imagen, podemos argumentar que dado nuestros resultados estamos en el inicio de la red y a medida que incrementa las épocas se irá estabilizando hasta que el generador engañe completamente al discriminador.



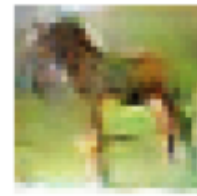
A continuación un ejemplo de la mejora de generación de imágenes con los caballos, en las primeras 10 épocas y en las épocas 40-50.

Épocas 0-10



Épocas 40-50

Figure 1



Conclusiones

A primera vista puede parecer que la GAN no está haciendo un trabajo prolijo pero teniendo en cuenta que las imagenes generadas son de 32x32 (es decir, se ven ampliadas en los resultados y por ende pierden calidad), los resultados no están nada mal y muestra el potencial que estas redes tienen para la generación de imagenes realmente convincentes. Además, es importante también tener en consideración la baja cantidad de épocas que entrenó cada GAN, siendo todas menor a 50 y los buenos resultados que dio. Dado las precisiones tan altas de los datos falsos y la evidente mejoría en las imagenes generadas a través de las épocas, se observa que de haberse dejado durante un número mayor de épocas, el generador habría comenzado a engañar aún más al discriminador, y las imagenes hubieran sido cada vez más y más realistas, especialmente con los casos donde lo entrené durante menos épocas que los resultados no fueron muy convincentes.