

Bootcamp Inteligencia Artificial

Nivel Innovador

TALENTO
TECH

Semana 14:

Aplicaciones de Deep Learning

Agenda

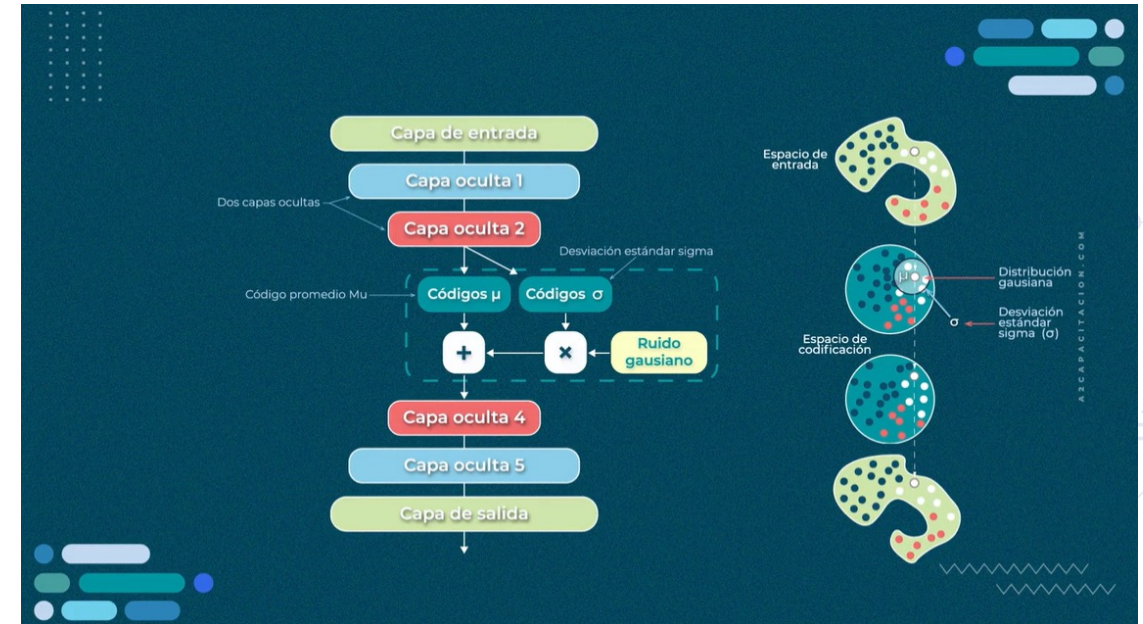
1. Autoencoders y GANs 2

1.1 Autoencoder Variacional

Otra categoría importante de autoencoders es el autoencoder variacional – y se ha vuelto uno de los autoencoders más populares.

Son muy diferentes de todos los autoencoders que hemos visto hasta ahora en estas 2 maneras:

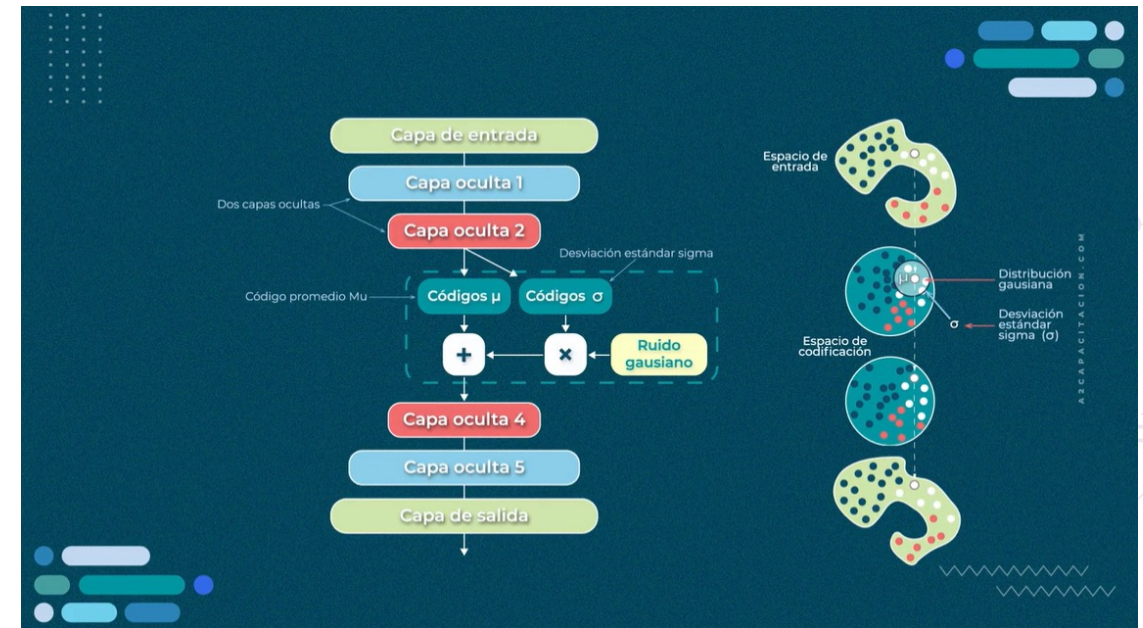
- Son probabilísticos, lo que significa que sus salidas son parcialmente determinados por la probabilidad, incluso después de entrenar.
- Son autoencoders generativos, lo que significa que pueden generar nuevas instancias que parecen que salieron del set de entrenamiento.



1.2 Autoencoder Variacional

Tienen la estructura básica de todos los autoencoders, con un encoder seguido de un decoder (en este ejemplo 2 capas ocultas), pero hay un twist, en vez de producir directamente un código para una entrada dada, el encoder produce un código promedio μ y una desviación estándar sigma.

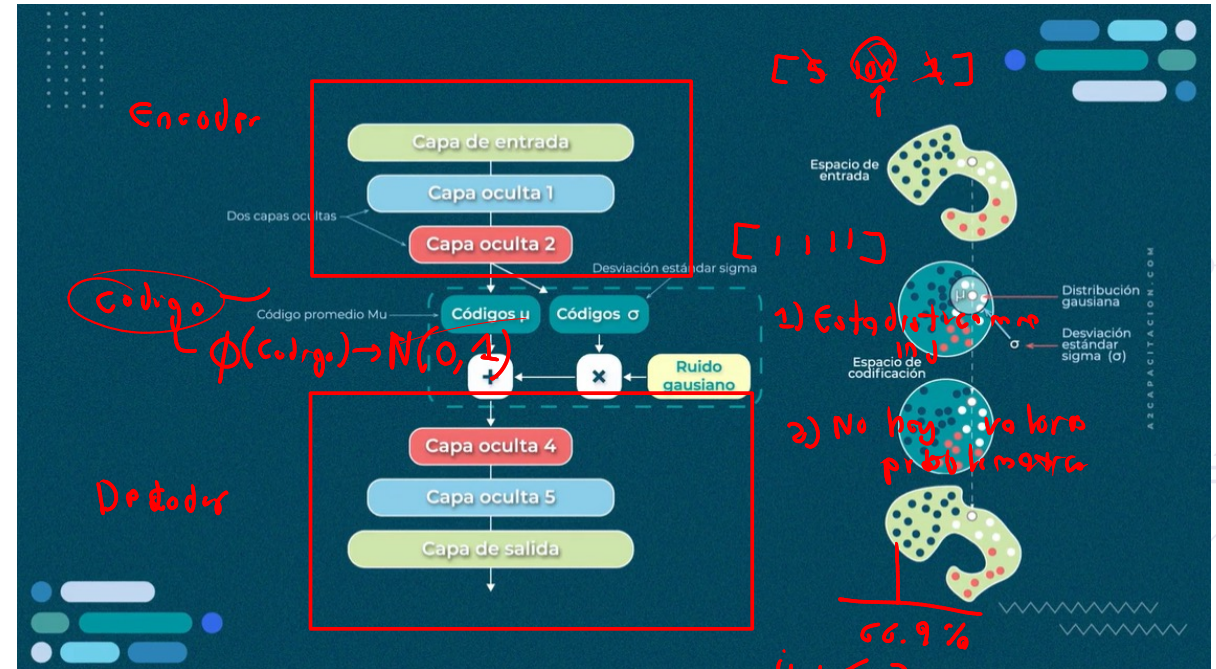
La codificación actual es muestreada al azar desde una distribución gaussiana con media μ y desviación estándar sigma. Después de eso el decodificador descodifica el código muestreado de manera normal.



1.3 Autoencoder Variacional

Como puedes ver en el diagrama, a pesar de que las entradas pueden tener una distribución muy jodida, un autoencoder variacional suele producir codificaciones que se ven como si fueran sacadas de una distribución gaussiana sencilla: durante el entrenamiento la función de costo empuja las codificaciones a migrar gradualmente dentro del espacio de codificación (espacio latente) para acabarse viendo como una capa de puntos gaussianos.

Una gran concuencia es que después de entrenar un autoencoder variacional, puedes muy fácilmente entrenar una nueva instancia: solo muestrea una codificación aleatoria desde la distribución gaussiana y ¡listo!



1.4 Autoencoder Variacional

Ahora, veamos a la función de costo. Se compone de 2 partes, la primera es la común pérdida de reconstrucción que empuja el autocodificador a reproducir sus entradas. La segunda es la pérdida latente que empuja el autocodificador a verse como si estuvieran muestreadas desde una simple distribución gaussiana, es la divergencia KL entre la distribución objetivo y la distribución verdadera de las codificaciones.

La matemática es un poco más compleja que con el autoencoder disperso, debido al ruido gaussiano, lo que limita la cantidad de información que puede ser transmitida a la capa de codificación. Dicho eso la ecuación de pérdida latente para el autoencoder variacional es esta:

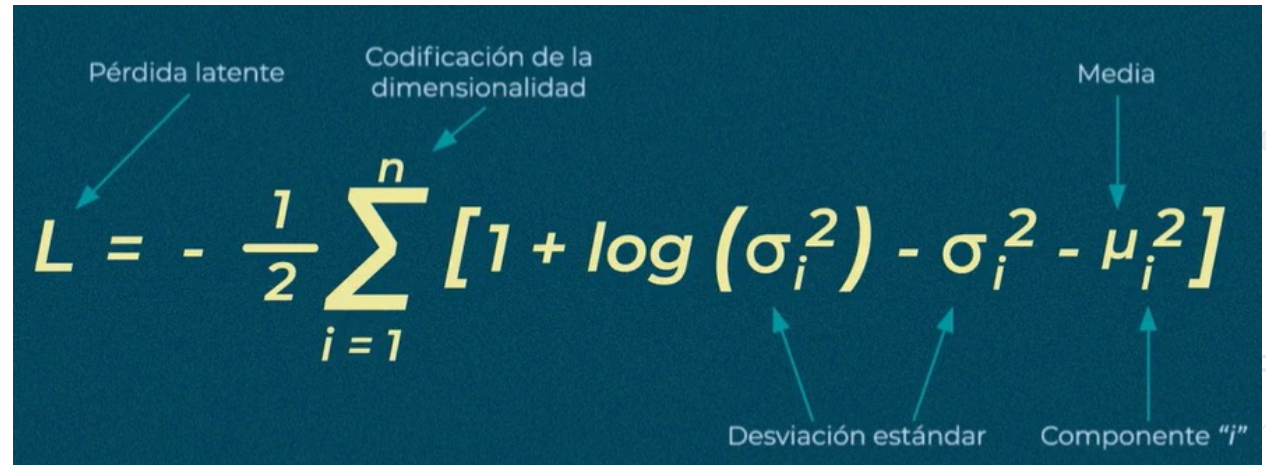


Diagrama de la ecuación de pérdida latente con anotaciones:

- Pérdida latente (apunta a L)
- Codificación de la dimensionalidad (apunta a n)
- Media (apunta a μ_i^2)
- Desviación estándar (apunta a σ_i^2)
- Componente "i" (apunta a i)

$$L = - \frac{1}{2} \sum_{i=1}^n [1 + \log(\sigma_i^2) - \sigma_i^2 - \mu_i^2]$$

1.5 Autoencoder Variacional

En esta ecuación, L es a la perdida latente, n es la codificación de la dimensionalidad, y μ y σ son la media y la desviación estándar del componente i de los códigos. Los vectores μ y σ son sacados por el enconder.

Una modificación común es hacer el autoencoder sacar $y = \log(\sigma^2)$ en vez de σ . La perdida latente se computa ahora así:

$$\gamma = \log (\sigma_i^2)$$

$$\begin{aligned} \exp(y) &= \cancel{\log(\sigma_i^2)} \\ L &= -\frac{1}{2} \sum_{i=1}^n [1 + \log(\sigma_i^2) - \sigma_i^2 - \mu_i^2] \\ L &= -\frac{1}{2} \sum_{i=1}^n [1 + \gamma_i - \exp(y_i) - \mu_i^2] \end{aligned}$$

1.6 Autoencoder Variacional (Código)

Vamos ahora a construirlo en Python. Lo usaremos para Fashion MNIST. Primero necesitamos una capa personalizada para muestrear las codificaciones, dadas μ y Σ .

Crea una capa personalizada para el muestreo.

```
class Sampling(tf.keras.layers.Layer):  
    def call(self, inputs):  
        mean, log_var = inputs  
        return tf.random.normal(tf.shape(log_var)) * tf.exp(log_var/2) + mean
```

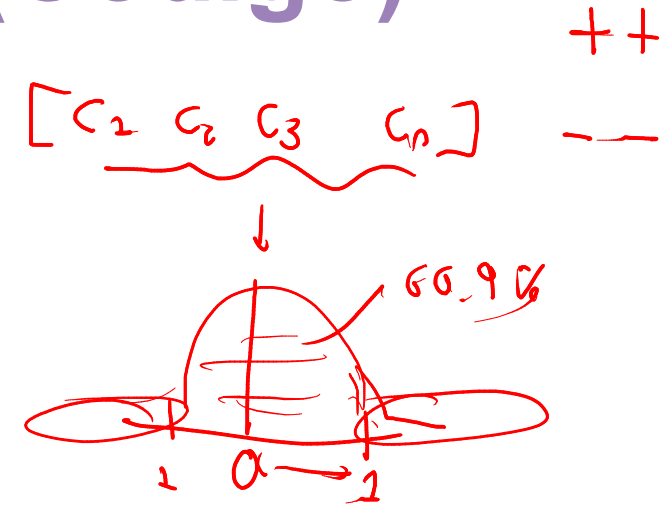
Esta capa de muestreo toma 2 entradas, el promedio y el logaritmo de la varianza. Usa la función `random_normal` para muestrear un vector aleatorio de la distribución normal, con promedio 0 y desviación estándar de 1. Luego multiplica el vector por $\exp(\Sigma/2)$, y finalmente le suma μ y regresa el resultado. Esto muestra un vector codificado de la distribución normal media μ y desviación estándar σ .

1.7 Autoencoder Variacional (Código)

Luego creamos el encoder usando la API funcional:
codings_size = 10

```
inputs = tf.keras.layers.Input(shape=[28, 28])
Z = tf.keras.layers.Flatten()(inputs)
Z = tf.keras.layers.Dense(150, activation="relu")(Z)
Z = tf.keras.layers.Dense(100, activation="relu")(Z)
codings_mean = tf.keras.layers.Dense(codings_size)(Z) #  $\mu$ 
codings_log_var = tf.keras.layers.Dense(codings_size)(Z) #  $\gamma$ 
codings = Sampling()(codings_mean, codings_log_var)
variational_encoder = tf.keras.Model(
    inputs=[inputs], outputs=[codings_mean, codings_log_var, codings])
```

Nota de las capas Densas que sacan codings_mean y codings_log_var (gamma) tienen las mismas entradas. Cuando pasamos ambos codings mean y codings log var a la capa de muestreo. Finalmente, el modelo tiene tres salidas, en la única salida que usaremos será la última.



1.8 Autoencoder Variacional (Código)

Sigue el decoder

```
decoder_inputs = tf.keras.layers.Input(shape=[codings_size])  
x = tf.keras.layers.Dense(100, activation="relu")(decoder_inputs)  
x = tf.keras.layers.Dense(150, activation="relu")(x)  
x = tf.keras.layers.Dense(28 * 28)(x)  
outputs = tf.keras.layers.Reshape([28, 28])(x)  
variational_decoder = tf.keras.Model(inputs=[decoder_inputs], outputs=[outputs])
```

Para este decodificador podríamos haber usado la API secuencial en vez de la función, ya que es una apilación sencilla de capas. Casi idénticas a los decodificadores que hemos armado hasta ahorita. Finalmente, vamos construyendo el autoencoder variacional.

```
_, _, codings = variational_encoder(inputs)  
reconstructions = variational_decoder(codings)  
variational_ae = tf.keras.Model(inputs=[inputs], outputs=[reconstructions])
```

1.9 Autoencoder Variacional (Código)

Finalmente agregamos la pérdida latente y la pérdida de reconstrucción:

```
latent_loss = -0.5 * tf.reduce_sum(  
    1 + codings_log_var - tf.exp(codings_log_var) - tf.square(codings_mean),  
    axis=-1)  
variational_ae.add_loss(tf.reduce_mean(latent_loss) / 784.)
```

La pérdida de reconstrucción se supone que es la suma de los errores de reconstrucción de píxel, pero cuando keras compute la pérdida de binary crossentropy, computa la pérdida sobre todos los 784 píxeles, en vez de la suma.

Por lo mismo la pérdida de reconstrucción es 784 veces más chica que lo que necesitamos que sea. Podríamos definir una pérdida personalizada para computar la suma en vez del promedio, pero sería más simple dividir la pérdida latente por 784

1.10 Autoencoder Variacional (Código)

¡Y finalmente entrenamos!

Entrenamos

```
variational_ae.compile(loss="mse",  
optimizer="nadam")  
history = variational_ae.fit(X_train,  
X_train, epochs=25, batch_size=128,
```

```
validation_data=(X_valid, X_valid))
```

Revisa tus resultados.

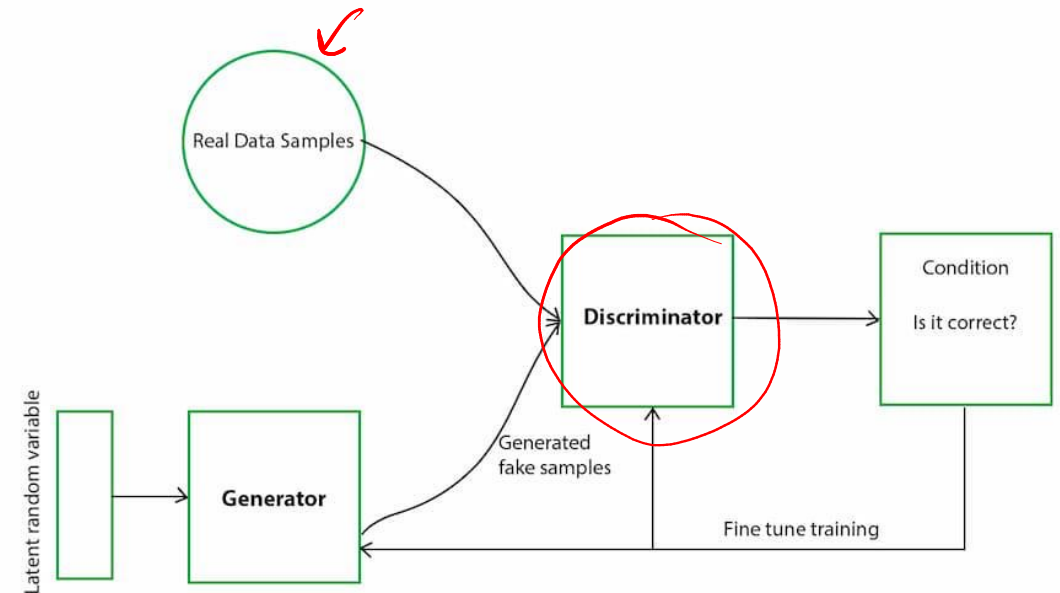
```
plot_reconstructions(variational_ae)  
plt.show()
```



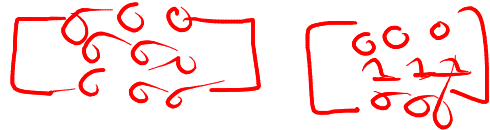
1.11 GANs – Generative Adversarial Networks o Redes generadoras Adversariales

Los GANs son lo máximo en cuanto a generación de imágenes falsas y son los responsables detrás de crímenes en contra de la humanidad como FaceApp, thisfacedoesnotexist.com, y esos sitios web rusos donde puedes subir fotos de gente y te dicen como se ve desnuda. Para fines científicos claro.

Los GANs fueron propuestos en el 2014 por un señor llamado Ian Goodfellow de Montreal y compañía. La idea era buena en principio, pero traía montón de problemas incluidos. Así que no despegaron hasta hace poco por el tema de resolver todos esos issues.



1.12 GANs – Generative Adversarial Network

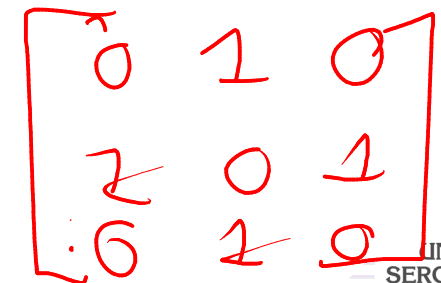
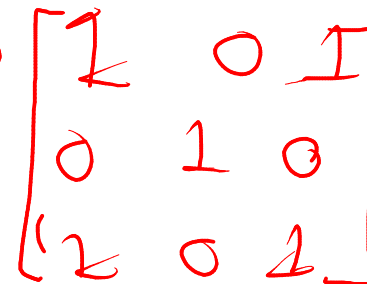
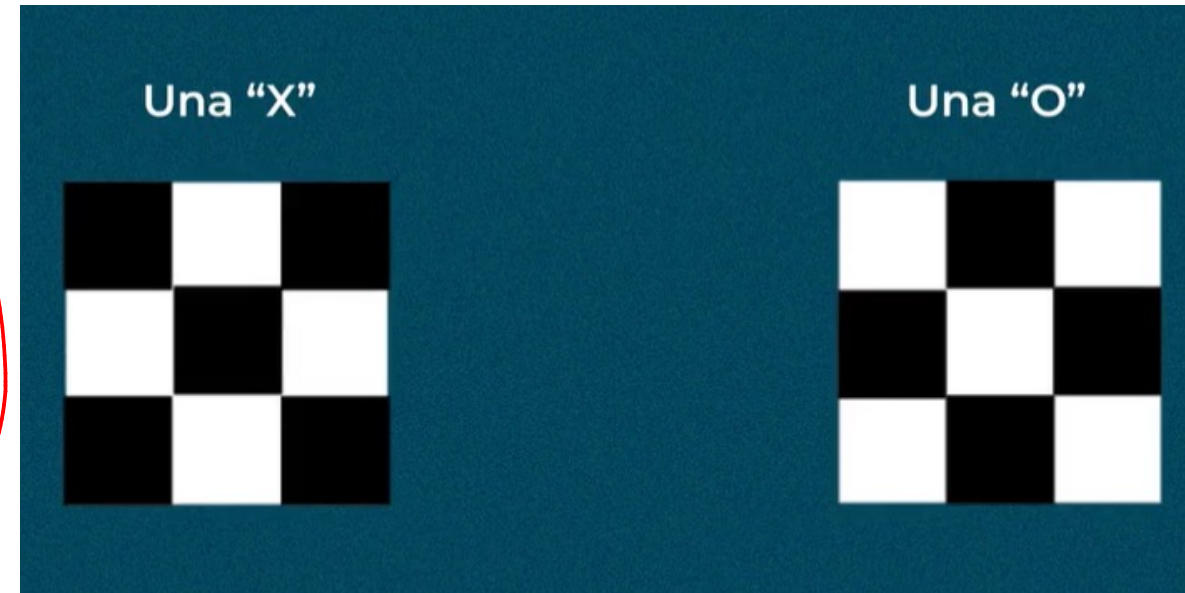


Vamos planteando el ejemplo de imagen más sencillo posible para que los números no se nos compliquen. Queremos armar un GAN que nos genere los símbolos del juego de gato en una pantalla de 3 x 3 pixeles. Es decir, no está muy complicado, este es el resultado final que estamos esperando, una X o una O.

Entonces vamos a tener 2 posibles resultados aceptables y ya. Vamos enfocándonos solo en la X y ya.

Vamos pegándole números, los cuadros perfectamente rellenos serán 1s y los vacíos serán 0s.

Ok, entonces si yo te paso una imagen de 3x3 de la X en escala de grises que se ve así con sus valores.



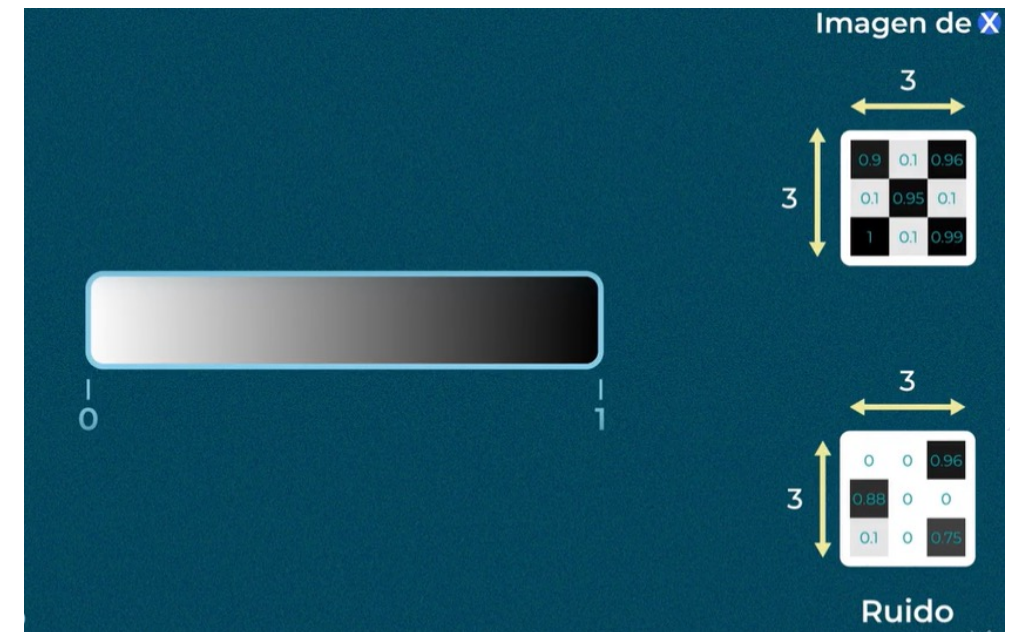
1.13 GANs – Generative Adversarial Network

Y si yo te paso una imagen que no es una X, es puro ruido, simplemente una figura amorfa sin aspiraciones ni porvenir, con sus valores, se vería así.

Este tipo de imágenes aleatorias serían el resultado de los primeros intentos de la red generadora de armar X que simplemente pues... resultaron siendo abortos.

La meta de la red es distinguir las X perfectas de este tipo de imágenes ruidosas.

Ok, vamos armando nuestras redes.



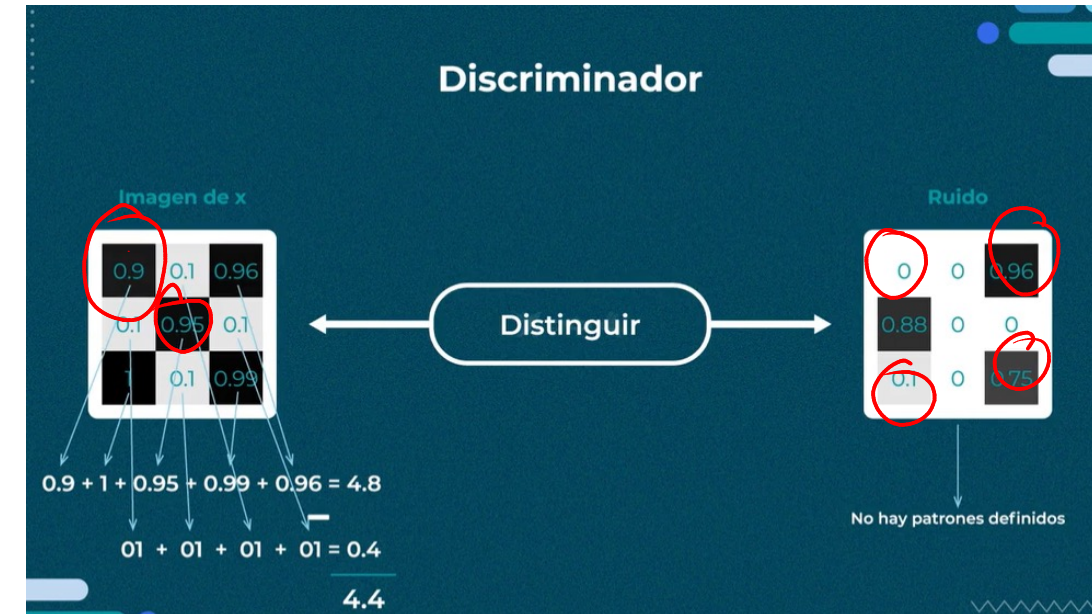
1.14 GANs – Generative Adversarial Network

la primera pregunta es como voy a distinguir las X de las no X. Fácil, nota que las X los pixeles de las esquinas y el del centro tienen valores altos, cercanos a 1 por que deberían de estar oscuros.

Por otro lado, las imágenes ruidosas pues no tienen ningún patrón definido, hay pixeles de todos los valores y en todos lados de la imagen.

Entonces podemos sumar los valores de las esquinas y el centro y restar los valores de los demás pixeles y este resultado debería estar muy cercano a 5 cuando se trata de una X casi perfecta.

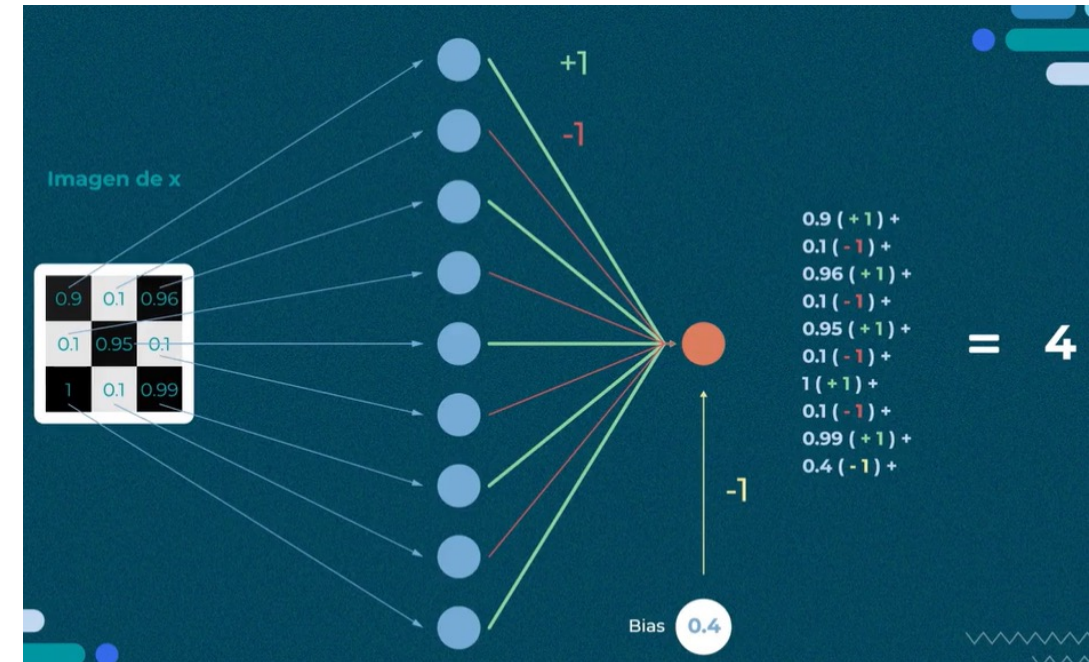
En este ejemplo el valor de los pixeles de la X es 4.8 y el valor de los pixeles extra es de 0.4, así que el resultado final es 4.4.



1.15 GANs – Generative Adversarial Network

Entonces la mejor idea sería definir un umbral de por ejemplo 4 y decir, cualquier puntaje mayor a 4 es una X bien hecha, y cualquier puntaje menor a 4 es una abominación y merece ser aventada a los fuegos del monte de la perdición.

En representación de red neuronal, así se verían las cosas, los valores de nuestros 9 pixeles se multiplican por +1 o -1 según donde están y luego le restamos el bias. Le sumamos estos 9 números y si el puntaje es 4 o mayor, entonces la imagen es clasificada como una X, y si es menos que 4 pues entonces no es una X.

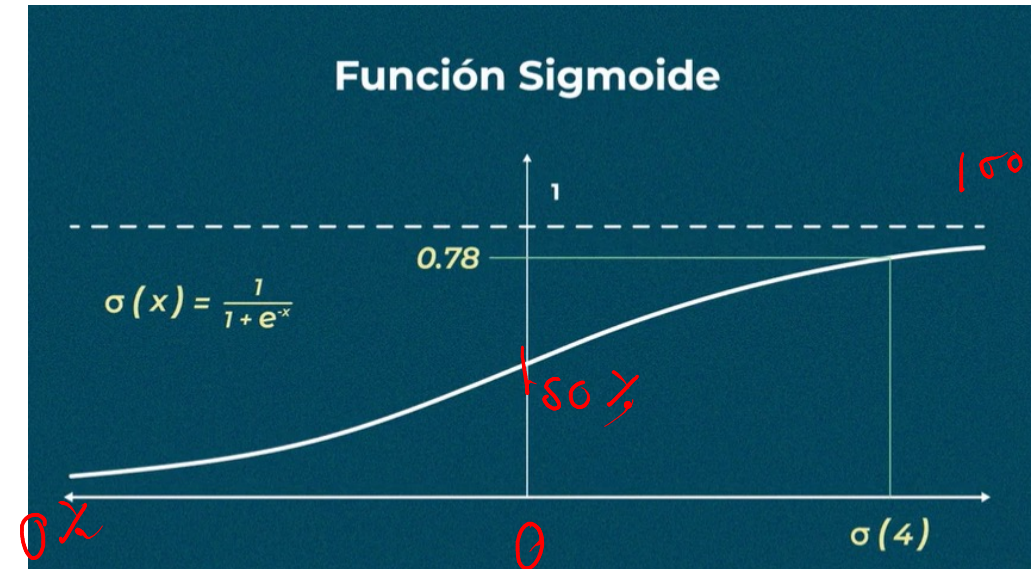


$[] \rightarrow \in X \text{ o } \notin X$

1.16 GANs – Generative Adversarial Network

También podemos usar la función de activación sigmoide – en este caso los valores más altos se van ir cercanos a 1 y los valores más bajos se van ir cercanos a 0.

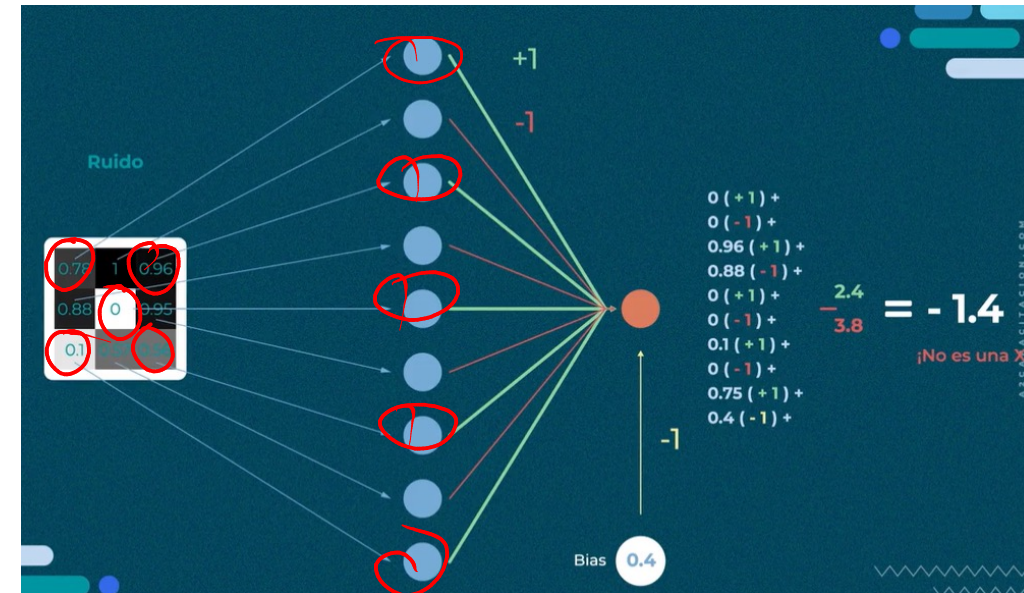
Si pasamos una imagen, la metemos a la función sigmoide, y nos resulta un 0.78 de resultado, nos van a devolver que esta imagen tiene un 78% de probabilidad de ser una X. Si nosotros decidimos aceptar este 78% como aceptable, entonces pues es cosa de nosotros como seres humanos. La red neuronal cumplió con su deber, el umbral lo tenemos que definir nosotros.



1.17 GANs – Generative Adversarial Network

Nota que, en esta red neuronal, las orillas gruesas son positivas y las delgadas son negativas, esta es una convención que usaremos en este video.

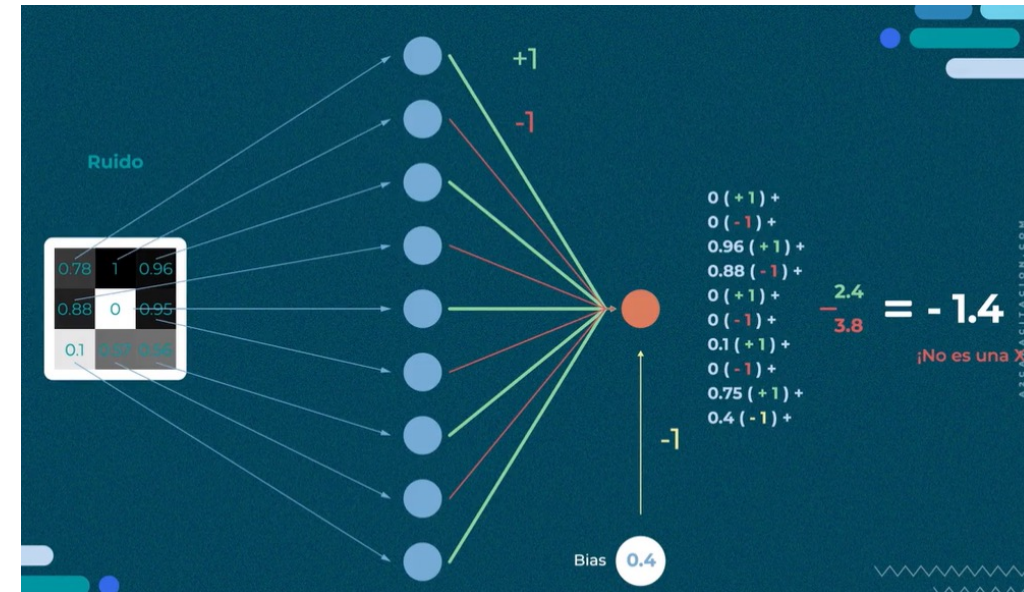
Si le metemos esta imagen que es puro ruido y no es una X, entonteces el discriminador va a hacer el mismo calculo y le va a dar un resultado positivo de 2.4 y un resultado negativo de 3.8, con un total de -1.4. O si se lo pasamos a la función sigmoide, nos dará un resultado de 0.13% y eso nos dejará que sea muy baja la probabilidad de que sea una X.



1.18 GANs – Generative Adversarial Network

Nota que, en esta red neuronal, las orillas gruesas son positivas y las delgadas son negativas, esta es una convención que usaremos en este video.

Si le metemos esta imagen que es puro ruido y no es una X, entonteces el discriminador va a hacer el mismo calculo y le va a dar un resultado positivo de 2.4 y un resultado negativo de 3.8, con un total de -1.4. O si se lo pasamos a la función sigmoide, nos dará un resultado de 0.13% y eso nos dejará que sea muy baja la probabilidad de que sea una X.



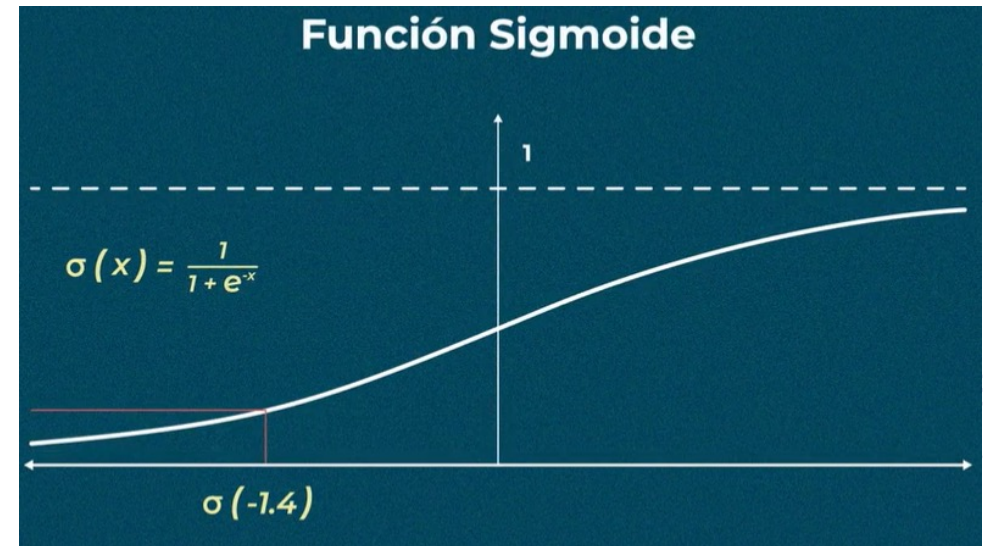
1.19 GANs – Generative Adversarial Network

Ya tenemos el discriminador, ahora vamos a construir un generador. El generador va a ser la red neuronal que va a sacar las X. Para construir el generador de nuevo hay que tomar en cuenta que las X bien hechas tienen valores cercanos a 1 en las esquinas y el centro.

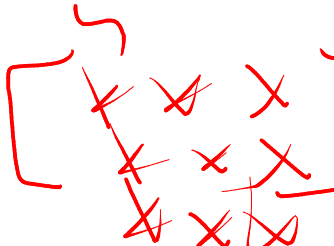
Entonces, así va a funcionar, primero empezamos eligiendo una entrada, que será un número aleatorio entre 0 y 1, en este caso digamos 0.8. En general la entrada será un vector que viene de alguna distribución fija.

$U(0, 1)$

Ahora construyamos una red neuronal. Lo que en realidad queremos es tener algunos valores más grandes y más chicos. Los más grandes son estas líneas gruesas y los más chicos son estas orillas pequeñas.



0 - 1



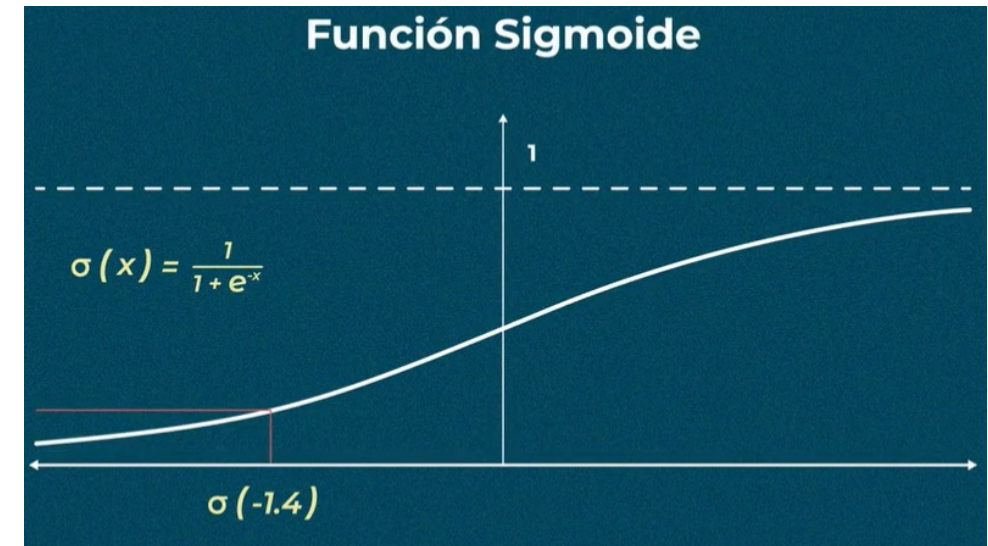
A hand-drawn diagram in red ink showing a 3x3 grid of 'x' marks. The grid is enclosed in large square brackets. The 'x' marks are arranged in a pattern that suggests a 3x3 matrix or a set of data points.

1.20 GANs – Generative Adversarial Network

Recuerda que queremos grandes valores para las esquinas y el centro, y valores chicos para los demás píxeles. así que como las salidas de las orillas tienen que ser grandes, queremos que los pesos de estos nodos sean grandes, por lo mismo, vamos a agregarles un bias de +1.

Así que, de resultado final, ahorita, le metemos un 0.8 y nos va a sacar 1.8 esta neurona por su bias.

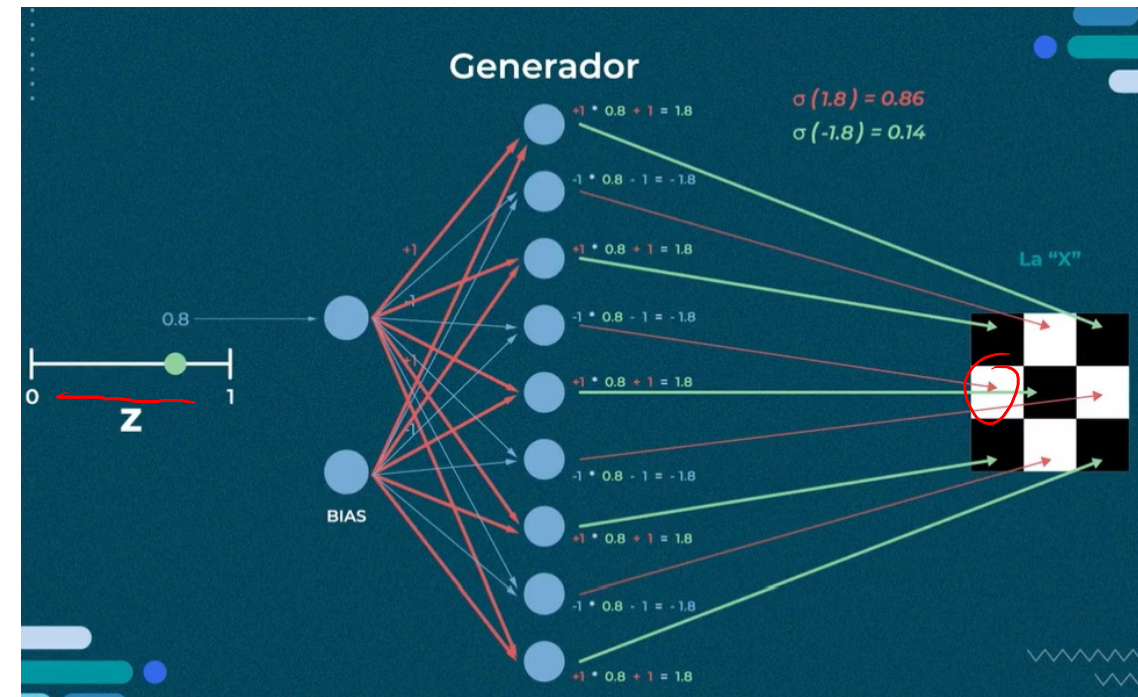
Ahora, para este píxel queremos que los valores sean chicos, así que para forzarlo vamos a ponerle bias de -1. Así que entrando el valor de 0.8, la neurona nos va a escupir -0.2.



1.21 GANs – Generative Adversarial Network

Lo mismo hacemos para los demás píxeles y nos acaba escupiendo esta imagen gracias a los bias que le metimos.

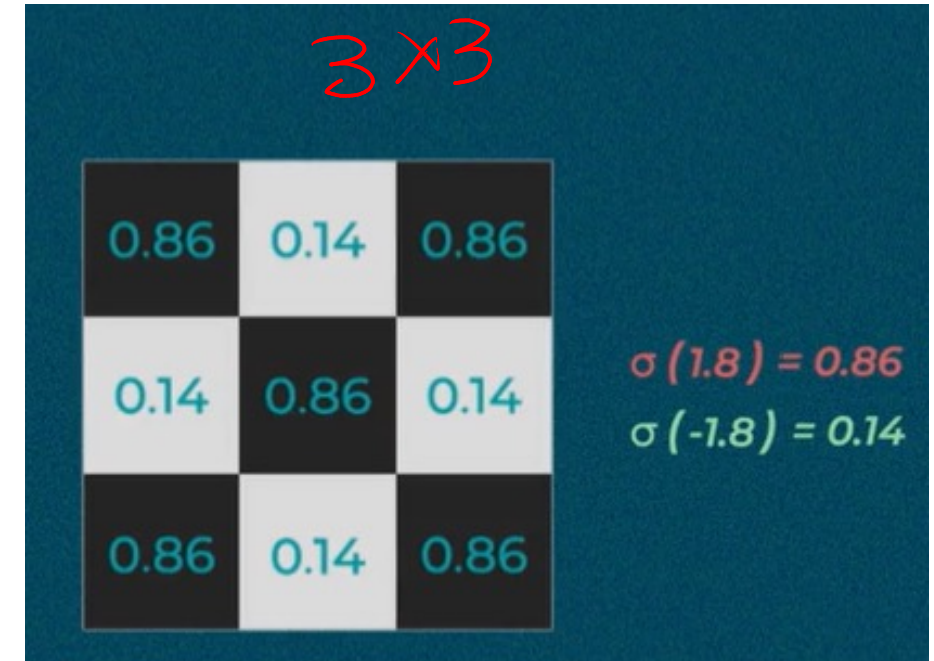
Ahora, esos son solo los puntajes que necesitamos para aplicar nuestra función de activación sigmoide. Par encontrar las probabilidades, le metemos nuestros valores a la sigmoide y nos devuelve esto.



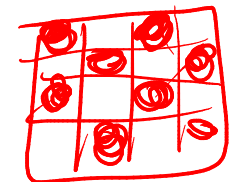
1.22 GANs – Generative Adversarial Network

Estos valores son los que van a entrar a nuestros pixeles – y fíjense, nuestra imagen se ve mucho como una X, que al final es lo que queremos. Esta red neuronal siempre va a generar grandes valores para las esquinas y el centro, y valores chicos para los demás pixeles.

Por lo mismo, esta red siempre nos va a estar pasando Xs, lo que significa que es un buen generador. Claro esta es una tarea hiper sencilla y básicamente pudimos definir los pesos y biases de pura vista. Ahora imagina que estas construyendo caras fotorrealistas en imágenes de 1080 x 1080 pixeles y el problema se vuelve mucho más difícil.



1080x1080

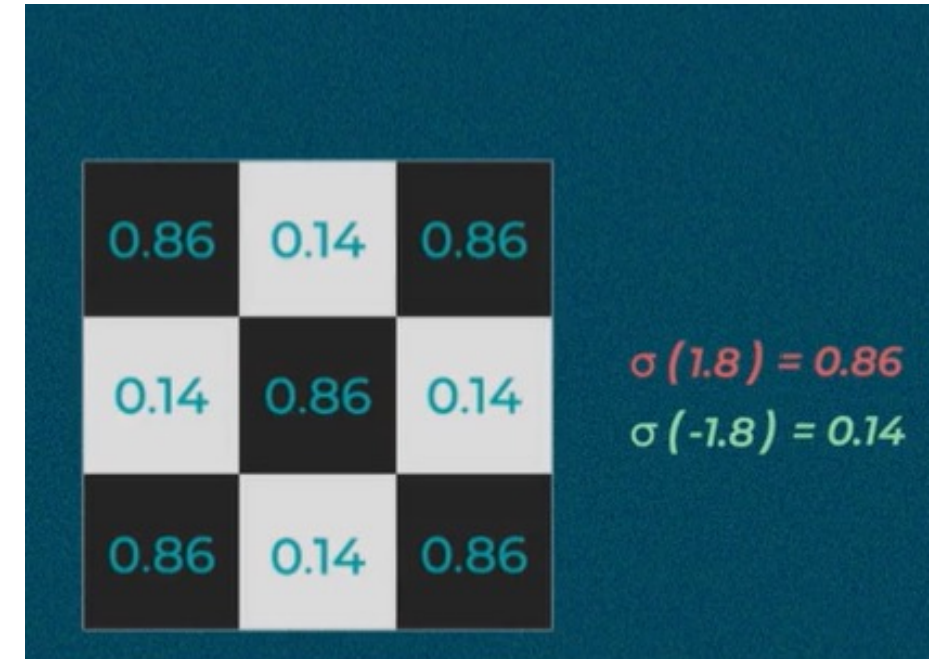


1.23 GANs – Generative Adversarial Network

Para eso vamos a tener que conseguir los mejores pesos posibles para nuestra red, y pues ya saben, hay que aplicar una función de error de verdad y echar a andar el backpropagation.

¿Cómo funciona en el caso de los GANs? Igual que en todas las demás redes neuronales.

El chiste en este caso va a ser poner las fórmulas de error correctas en los lugares correctos.

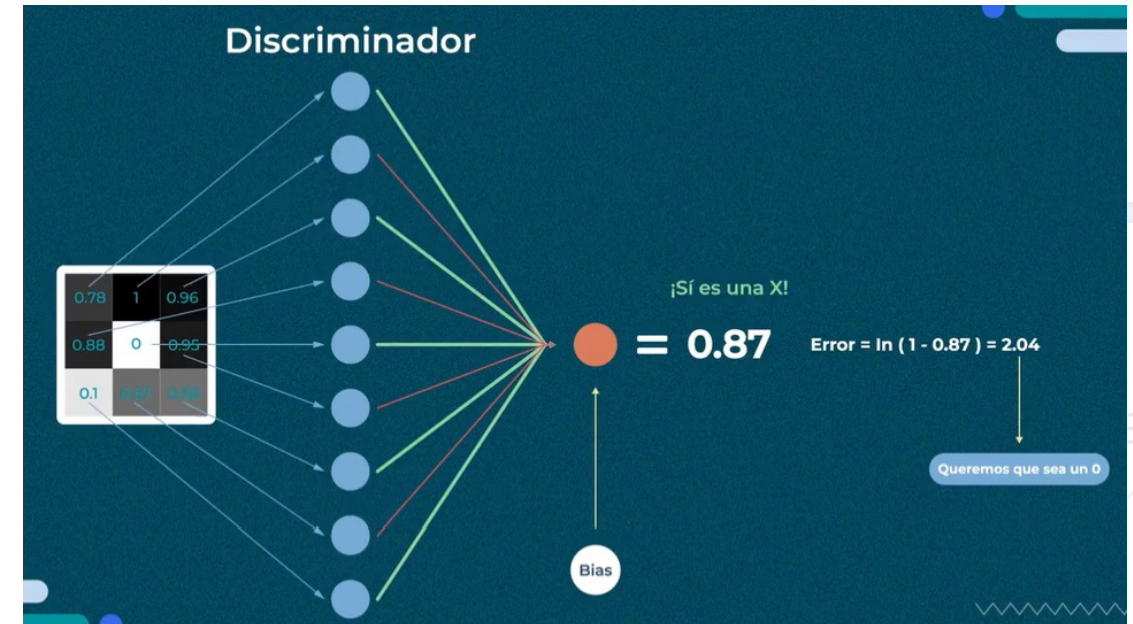


1.24 GANs – Generative Adversarial Network

Aquí esta nuestra red neuronal generadora, nota que los pesos todavía no están definidos, así que vamos a comenzar definiéndolos como números aleatorios. Ahora, vamos a seleccionar algunos números entre 0 y 1 que van a servir de entrada para el generador.

En el pase hacia adelante nos va a escupir una imagen que, lo más probable, no sea una X. Recuerda que comenzamos con números aleatorios.

Ok, este es el resultado, obviamente no es una X. Ahora sí, vamos a pasar esta imagen generada a través del discriminador para que nos diga si es una X o no es.

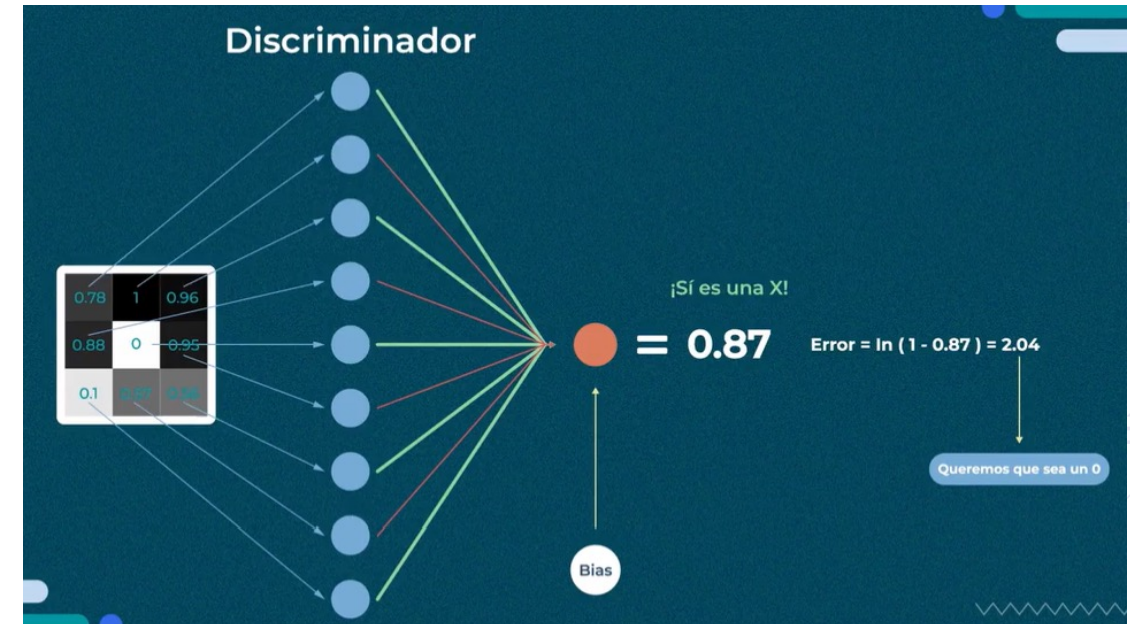


1.25 GANs – Generative Adversarial Network

Aquí esta nuestra red neuronal generadora, nota que los pesos todavía no están definidos, así que vamos a comenzar definiéndolos como números aleatorios. Ahora, vamos a seleccionar algunos números entre 0 y 1 que van a servir de entrada para el generador.

En el pase hacia adelante nos va a escupir una imagen que, lo más probable, no sea una X. Recuerda que comenzamos con números aleatorios.

Este es el resultado, obviamente no es una X. Ahora sí, vamos a pasar esta imagen generada a través del discriminador para que nos diga si es una X o no es.



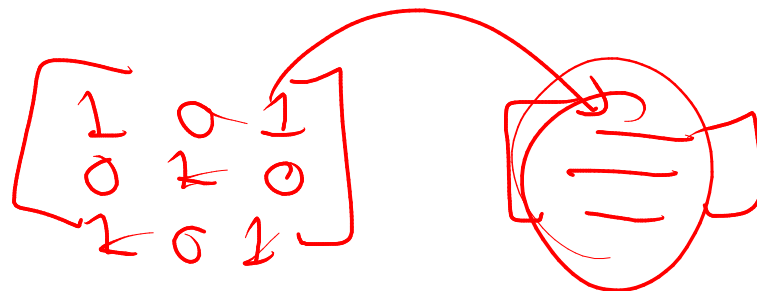
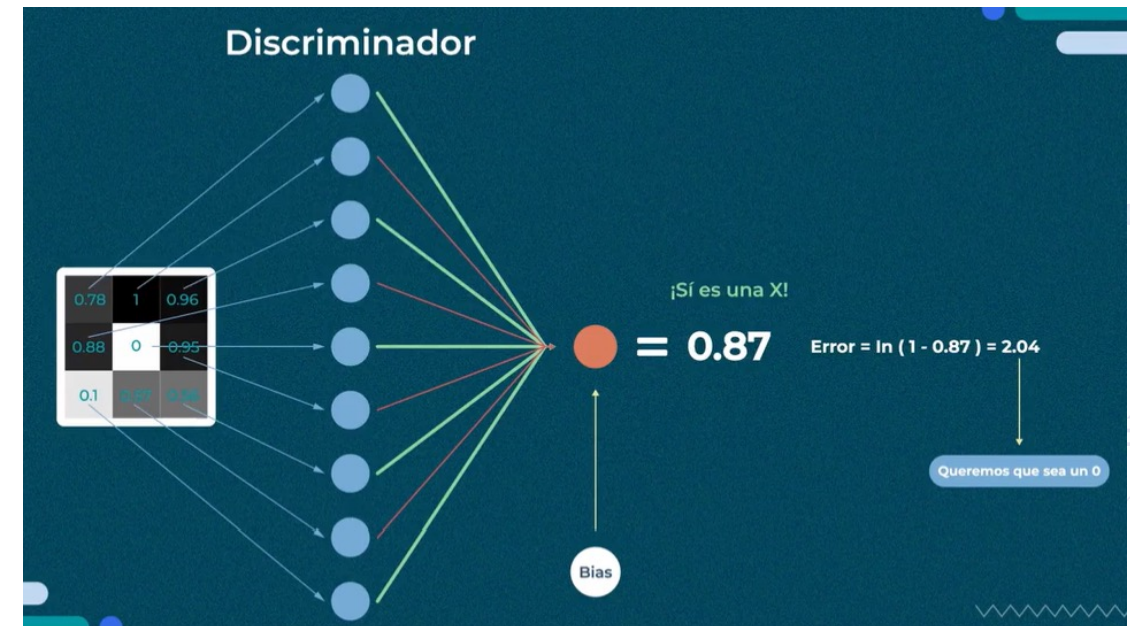
1.26 GANs – Generative Adversarial Network

El discriminador va a sacar una probabilidad, digamos, de 0.87% de que SI es una X. Y aquí es donde va a entrar nuestra función de pérdida al rescate.

Obviamente no es una X, así que el discriminador está bien equivocado. Debería estarnos dando un 0, no un 87%. En este caso vamos a estar usando esta función de error, que para que nos devuelva un 0 queremos que nos dé el $-\ln(1 - \text{la predicción})$.

Error

Este es un error que nos va a ayudar a entrenar al discriminador. Ahora que sabemos que quiere el discriminador, vamos al generador.



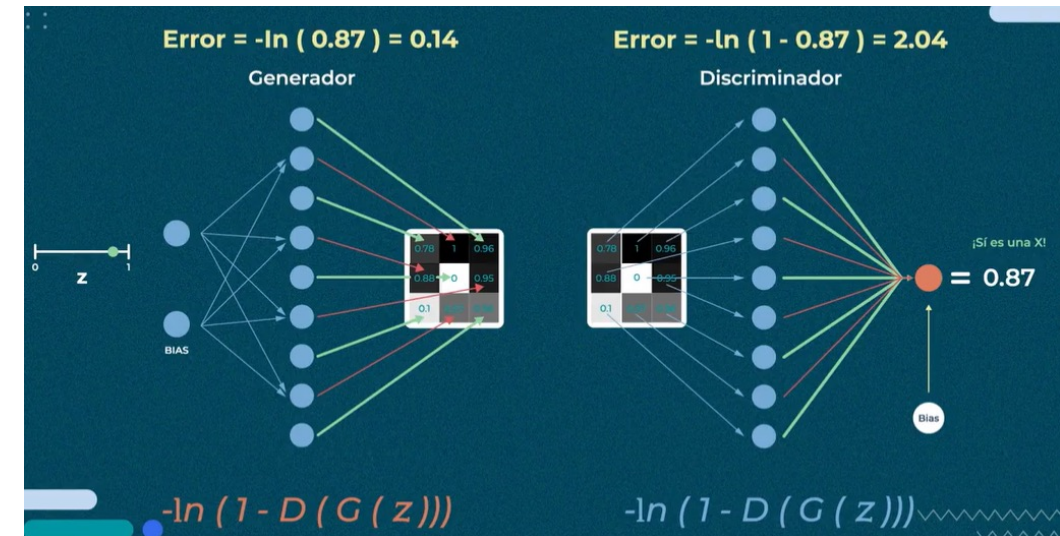
Discriminador + loss
 $loss = -\ln(Diff)$

1.27 GANs – Generative Adversarial Network

¿Qué quiere el generador?

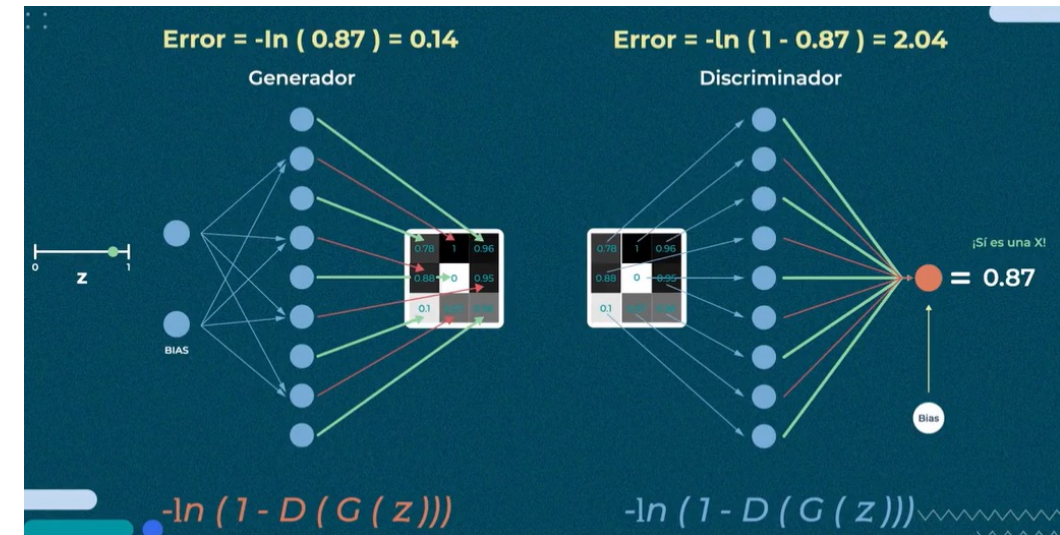
El generador quiere generar una X tan perfecta que el discriminador la clasifique de real. Ósea, quiere que el discriminador reciba la imagen y saque un 1.

Eso significa que la función de error del generador es el logaritmo negativo de la predicción, así que esa es la función de error que nos va a ayudar a entrenar los pesos del generador. En otras palabras, si G de Z es la salida del generador, y Derivada de G de Z es la salida del discriminador, entonces la función de error para el generador es el $-\ln(\text{derivada de } G(Z))$ y la función de error del discriminador es negativo logaritmo de 1 menos DGZ.



1.28 GANs – Generative Adversarial Network

Las derivadas de estos 2 son lo que nos va a ayudar nuestros pesos de ambas redes neuronales para mejorar la predicción particular. Nota que estas 2 funciones de error pelean contra la otra, pero no hay problema porque la función real de error solo cambia los pesos del generador y la función de error azul solo cambia el discriminador.



1.29 GANs para MNIST

Primero, necesitamos construir el generador y el discriminador. El generador es similar a un decodificador de un autoencoder, y el discriminador es simplemente un clasificador regular binario (la imagen sirve o no sirve, al final de cuentas).

Para la segunda fase de cada iteración de entrenamiento, también necesitamos el modelo GAN completo que contiene el generador seguido del discriminador.

```
codings_size = 30
```

```
Dense = tf.keras.layers.Dense  
generator = tf.keras.Sequential([  
    Dense(100, activation = "relu", kernel_initializer="he_normal"),  
    Dense(150, activation="relu", kernel_initializer="he_normal"),  
    Dense(28 * 28, activation="sigmoid"),  
    tf.keras.layers.Reshape([28, 28])  
])
```

1.30 GANs para MNIST

Arma el discriminador.

```
discriminator = tf.keras.Sequential([  
    tf.keras.layers.Flatten(),  
    Dense(150, activation = "relu", kernel_initializer = "he_normal"),  
    Dense(100, activation="relu", kernel_initializer="he_normal"),  
    Dense(1, activation="sigmoid")  
])
```

Arma el GAN.

```
gan = tf.keras.Sequential([generator,discriminator])
```

Ahora necesitamos compilar estos modelos. Como el discriminador es un clasificador binario, podemos usar perdida binaria de entropía cruzada. El generador solo será entrenado mediante el modelo de GAN, así que no necesitamos compilarlo.

1.31 GANs para MNIST

Compila el discriminador.

```
discriminator.compile(loss="binary_crossentropy", optimizer="rmsprop")
discriminator.trainable = False
```

Compila el GAN.

```
gan.compile(loss="binary_crossentropy", optimizer="rmsprop")
```

Contru

Prepara tus datos en batches.

```
batch_size = 32
dataset = tf.data.Dataset.from_tensor_slices(X_train).shuffle(1000)
dataset = dataset.batch(batch_size, drop_remainder=True).prefetch(1)
```

Importante, el discriminador no debería hacer entrenamiento durante la segunda etapa, así que hay que hacerlo no entrenarlo antes de compilar el modelo de GAN.

1.32 GANs para MNIST

Vamos a necesitar un ciclo de entrenamiento personalizado.

```
def train_gan(gan, dataset, batch_size, codings_size, n_epochs):
    generator, discriminator = gan.layers
    for epoch in range(n_epochs):
        print(f"Epoch {epoch + 1}/{n_epochs}")
        for X_batch in dataset:
            noise = tf.random.normal(shape=[batch_size, codings_size])
            generated_images = generator(noise)
            X_fake_and_real = tf.concat([generated_images, X_batch], axis=0)
            y1 = tf.constant([[0.]] * batch_size + [[1.]] * batch_size)
            discriminator.train_on_batch(X_fake_and_real, y1)
            #Etapa 2
            noise = tf.random.normal(shape=[batch_size, codings_size])
            y2 = tf.constant([[1.]] * batch_size)
            gan.train_on_batch(noise, y2)
        plot_multiple_images(generated_images.numpy(), 8)
        plt.show()
```

1.33 GANs para MNIST

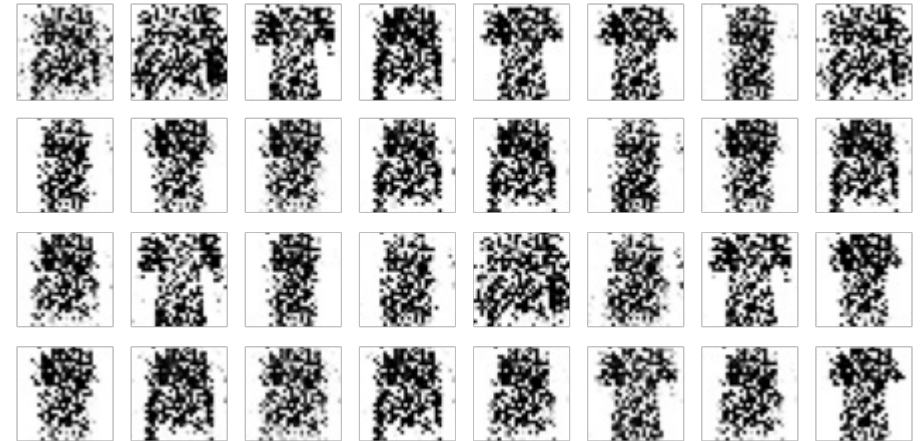
Y ahora vamos a armar la función de entrenamiento.

```
train_gan(gan, dataset, batch_size, codings_size, n_epochs=50)
```

Como discutimos anteriormente, puedes ver 2 fases en cada iteración:

- En la primera fase le alimentamos ruido Gaussiano al generador para producir imágenes falsas, y luego completamos este bache concatenando un número igual de imágenes reales. Los objetivos y_1 están puesto para 0 para imágenes falsas y 1 para imágenes reales. Luego entrenamos el discriminador en este bache.
- En la segunda fase, alimentamos el GAN un poco de ruido gaussiano. El generador va a empezar a producir imágenes falsas, y luego el discriminador va a intentar adivinar si estas imágenes son falsas o reales. Queremos que el discriminador crea que las imágenes falsas son reales, así que los objetivos y_2 se ponen en 1.

Y listo, si muestras las imágenes generadas, veras que al final de la primera época, medio se ven como imágenes de MNIST moda.

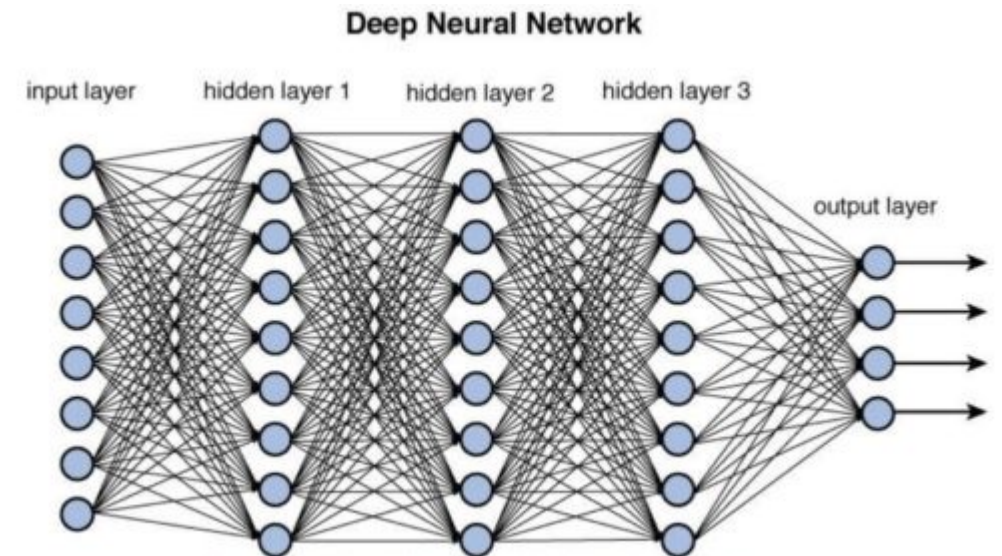


1.34 GANs convolucionales profundas

Las GANs construidas usando redes convolucionales han sido problemáticas. Se propusieron desde el primer momento, en el 2014, pero siempre acababan con gradientes inestables. Fue hasta finales del 2015 que Alec Radford tuvo éxito y bautizo su creación **DCGAN – Deep Convolutional GAN**.

La arquitectura de DCGAN se basa en los siguientes principios:

- Reemplazar cualquier capa de pooling con convoluciones con zancada en el discriminador, y convoluciones transpuestas (en el generador).
- Usar normalización de Batch en el generador y discriminador, excepto en la capa de salida del generador y la capa de entrada del discriminador.
- Quitar las capas densas para las arquitecturas más profundas.
- Usar activación RELU en el generador para todas las capas excepto la de salida, que podría usar tanh.
- Usar RELU que gotea en el discriminador para todas las capas.

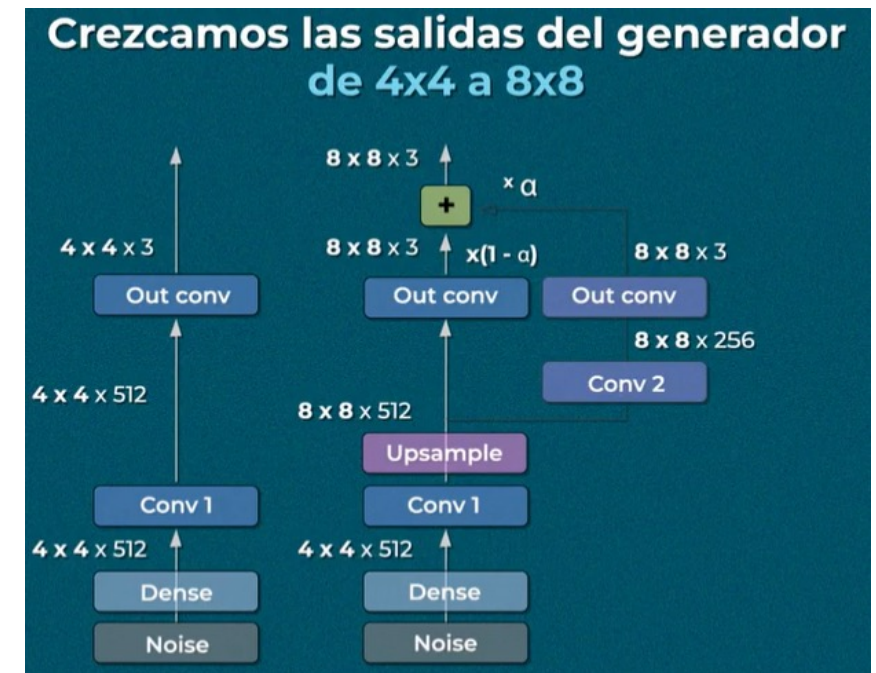


1.35 GANs que crecen progresivamente

Una técnica importante fue propuesta en el 2018 por Tero Karras de Nvidia – generar imágenes pequeñas al principio del entrenamiento, y luego agregar gradualmente capas convolucionales a el generador y el discriminador para producir imágenes más y más grandes.

Las capas extras se suman al final del generador y al inicio del discriminador, y las capas previamente entrenadas permanecen entrenables.

Por ejemplo, vamos a crecer las salidas del generador de 4x4 a 8x8 – para eso vamos a agregar una capa de upsampling a la capa convolucional existente para que saque mapas de 8x8, que luego se alimentan a la nueva capa convolucional.

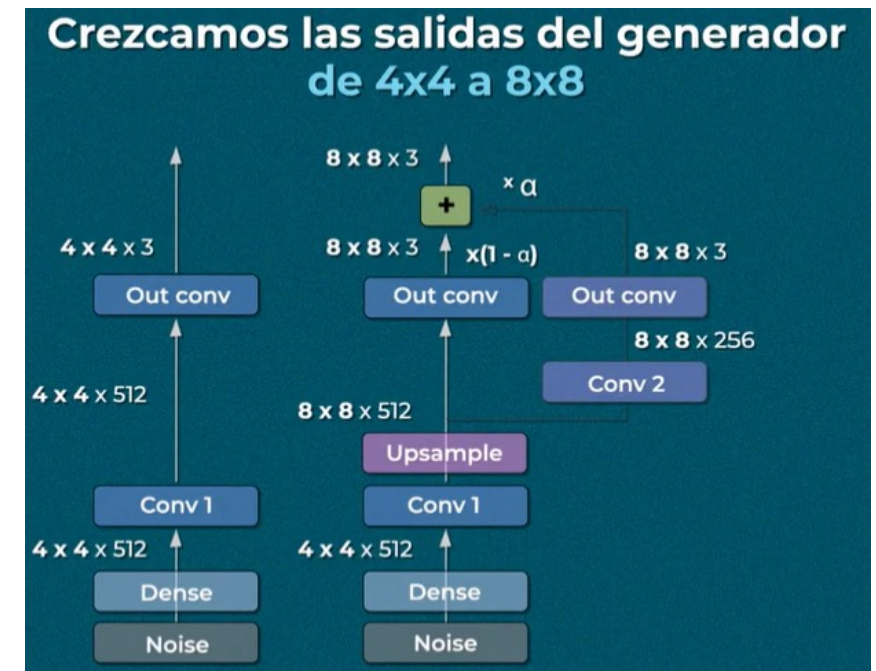


1.36 GANs que crecen progresivamente

Una técnica importante fue propuesta en el 2018 por Tero Karras de Nvidia – generar imágenes pequeñas al principio del entrenamiento, y luego agregar gradualmente capas convolucionales a el generador y el discriminador para producir imágenes más y más grandes.

Las capas extras se suman al final del generador y al inicio del discriminador, y las capas previamente entrenadas permanecen entrenables.

Por ejemplo, vamos a crecer las salidas del generador de 4x4 a 8x8 – para eso vamos a agregar una capa de upsampling a la capa convolucional existente para que saque mapas de 8x8, que luego se alimentan a la nueva capa convolucional.



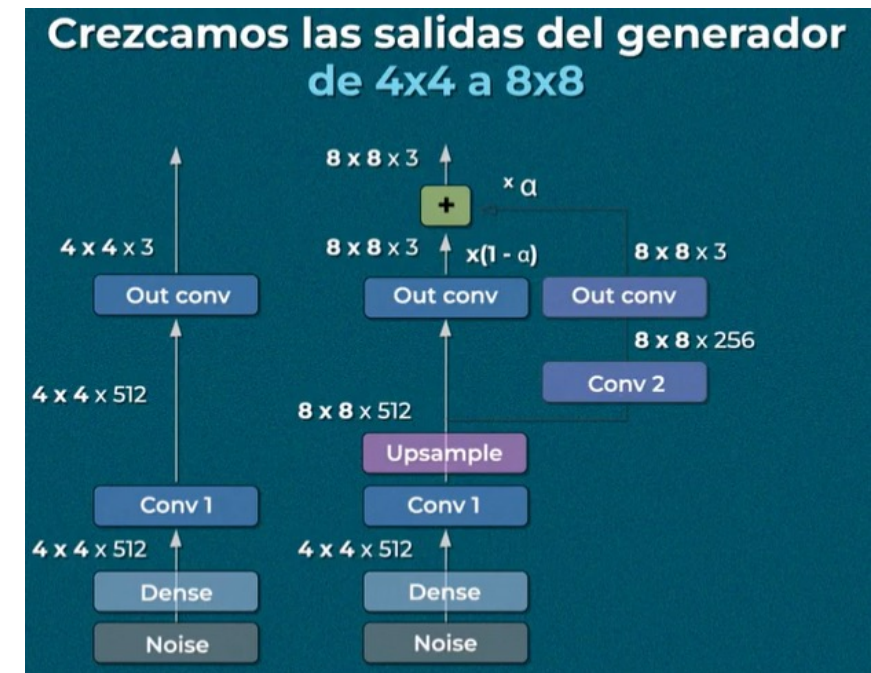
1.37 GANs que crecen progresivamente

Esta nueva capa es seguida por una nueva capa convolucional de salida: esta es una capa regular con tamaño de kernel de 1 que proyecta las salidas a través de el número deseado de canales de color.

Para evitar romper los pesos entrenados de la primer capa convolucional cuando la nueva capa convolucional se agrega, las salidas finales son una suma de la capa original de salida y la nueva capa de salida.

El peso de las nuevas salidas es alfa, mientras que el peso de las salidas originales es $1 - \alpha$, y alfa sube lentamente de 0 a 1.

En otras palabras, las nuevas capas convolucionales gradualmente entran, mientras que la capa de salida original lentamente se desvanece. Una técnica similar de entrada y salida usa cuando se agrega una nueva capa convolucional al discriminador.

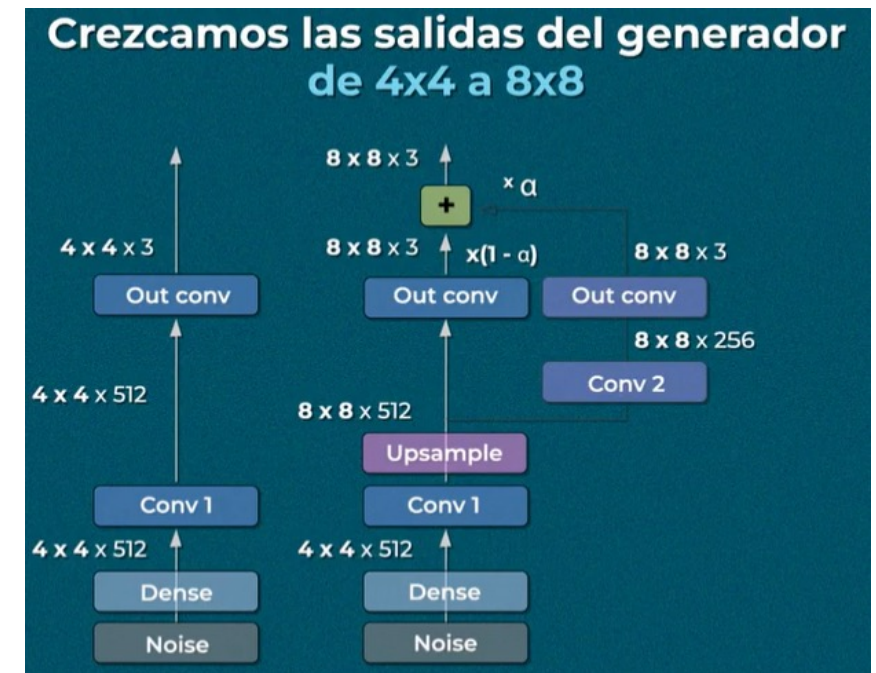


1.38 GANs que crecen progresivamente

Capa de Desviación estándar de Mini batch: Agregada cerca del final del discriminador. Para cada posición en las entradas, computa la desviación estándar a través de todos los canales y todas las instancias en el bache.

Estas desviaciones estándar luego se promedian a través de todos los puntos para obtener un solo valor.

Finalmente, se agrega un Feature Map adicional a cada instancia en el bache y se llena con el valor computado.

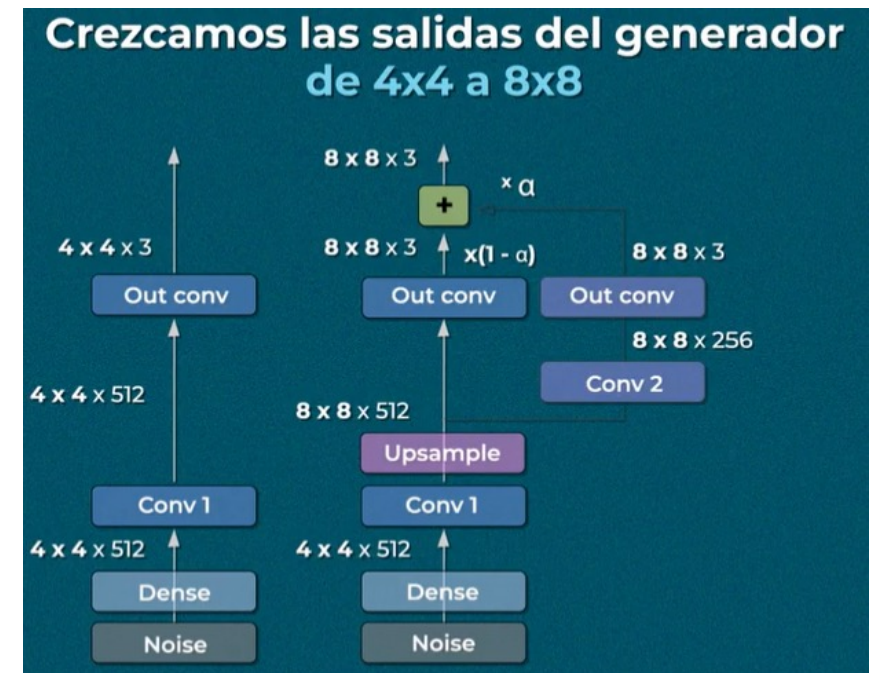


1.39 GANs que crecen progresivamente

Capa de Normalización Pixelwise: Se agrega después de cada capa convolucional en el generador. Normaliza cada activación basada en todas las activaciones en la misma imagen y ubicación, pero a través de todos los canales.

Esta técnica evita explosión en activaciones por competencia excesiva entre el generador y el discriminador.

Combinando estas técnicas sencillas, puedes generar caras extremadamente convincentes. Al menos para el ojo humano.



Preguntas

