UNIVERSIDAD NACIONAL DE LA PLATA

Facultad de Informática

Trabajo Final para la obtención del Título:

ESPECIALISTA EN INTELIGENCIA DE DATOS ORIENTADA A BIG DATA

REDES GENERATIVAS ANTAGÓNICAS Y SUS APLICACIONES

LIC. LAURA R. CALCAGNI

Dirección
DR. FRANCO RONCHETTI

La Plata, 2020

Redes generativas antagónicas y aplicaciones

Resumen:

En este trabajo se estudian las Redes Neuronales Generativas Antagónicas (GANs) y sus aplicaciones a través de una extensa revisión de bibliografía teórica y de los últimos artículos científicos publicados sobre el tema.
Se describen los modelos estadísticos generativos en contraposición con los discriminativos y se estudian las partes que componen las GANs, su entrenamiento y dificultades.
Este trabajo contempla, además, la implementación de una Red Generativa Antagónica de Super-Resolución (SRGAN) propuesta en [1], mediante las librerías *Keras* y *Tensorflow*. Se utiliza para su entrenamiento el entorno *Google Colaboratory*, que permite la utilización de una GPU NVIDIA Tesla K80.
En esta implementación se obtienen imágenes cualitativamente de alta calidad, que se pueden apreciar tanto utilizando como entrada imágenes del conjunto de prueba, como imágenes sin relación al conjunto de datos utilizado para entrenar la red.

Palabras Claves: GANs, SRGANs, Redes Generativas Antagónicas.

Generative Adversarial Networks and applications

Abstract:

In this project, Adversarial Generative Neural Networks (GANs) and their applications are studied through an extensive review of theoretical literature and the latest scientific papers published.
Generative statistical models are described in opposition to discriminatory models. The different blocks that componse a GAN are studied, as well as their training and the difficulties it involves.
This project also contemplates the implementation of a Super-Resolution Adversarial Generative Network (SRGAN) presented in [1], using the libraries *Keras* and *Tensorflow*. The Google Colaboratory environment is used for training the GAN, since the platform allows the use of a GPU NVIDIA Tesla K80. In this implementation, qualitatively high quality images are obtained, which can be appreciated both using as inputimages of the test set and images unrelated to the data set used to train the network.

Keywords: GANs, SRGANs, Generative Adversarial Networks.

Introducción

Todo comenzó en el año 2014, cuando Ian J. Goodfellow y coautores introdujeron en el artículo *Generative Adversarial Nets*[2] un novedoso modelo de generación de datos conocido como Redes Generativas Antagónicas (GANs, por su sigla en inglés.)
El modelo combinaba de forma muy original el aprendizaje profundo con la teoría de juegos y prontamente grandes referentes en el área lo denominaron como *"la idea más interesante de los últimos* 10 *años en el área de Machine Learning"*.

Las GANs consisten en dos modelos (en general, redes neuronales convolucionales), el generador y el discriminador, entrenados simultáneamente para desafiarse uno al otro, lo que explica el término *antagónicas* elegido por los autores para darle identidad a este novedoso método.
Por un lado, el generador es entrenado para generar datos falsos lo más parecidos posibles a los ejemplos reales de un determinado conjunto de entrenamiento que se selecciona. Por otro lado, el discriminador es entrenado para ser capaz de discernir los datos falsos producidos por el generador de aquellos que corresponden al conjunto de entrenamiento (los ejemplos reales).
Sucesivamente, los dos modelos tratan continuamente de superarse: cuanto mejor es el generador en la creación de datos convincentes, mejor debe ser el discriminador para distinguir los ejemplos reales de los falsos.

Cabe destacar, que las GANs no han sido el primer modelo utilizado para generar datos, sino por el contrario, distintos métodos de generación de datos han sido estudio hace años. Sin embargo, las GANs han conseguido resultatos notables y es por ello que son estudiadas extensamente en la actualidad.
Entre sus posibles aplicaciones, las más exitosas hasta ahora se han dado en el área de imágenes y visión por computadora, como la super-resolución de imágenes, la síntesis y manipulación de imágenes y el procesamiento de vídeo.

En este trabajo, se hace una introducción a los modelos generativos y se estudian las GANs, sus fundamentos, arquitecturas y aplicaciones. Se desarrolla con particular detalle la aplicación de las GANs en el área de super-resolución (SR). La SR de imágenes es un problema ampliamente estudiado en el campo de visión por computadora. El objetivo es generar una o más imágenes de alta resolución a partir de una o más imágenes de baja resolución.
El modelo SRGAN (Super Resolution Generative Adversarial Network) fue propuesto en 2016 por un grupo de investigadores de la empresa *Twitter* [1] y es el modelo que se implementa en este trabajo utilizando las librerías de Python *Keras* y *TensorFlow*. Para el entrenamiento de la SRGAN se utiliza en este trabajo el entorno *Google Colaboratory* (también conocido como *Colab*), que es un servicio en la nube gratuito que provee una Jupyter Notebook y permite la utilización de una GPU NVIDIA Tesla K80.

Contribuciones

I L. Calcagni
Repositorio GitHub
Super Resolution Generative Adversarial Network (**2020**) `https://github.com/lcalcagni/Super-resolution-Generative-Adversarial-Network`

Índice general

1

Modelos Generativos

En este capítulo se introducen los modelos generativos y se explican las diferencias que existen entre éstos y los modelos discriminativos. Se describen brevemente algunos algoritmos generativos, como los Clasificadores de Bayes Ingenuos y los dos más estudiados en el último tiempo: los Autocodificadores Variacionales y las las Redes Generativas Antagónicass.

Índice

1.1 Conceptos preliminares

Dentro del campo del aprendizaje automático, un importante área la constituyen los modelos generativos. Éstos suelen tener variables latentes, es decir, variables que no son observadas explícitamente sino que se infieren a partir de los datos observados. Luego, las variables latentes se utilizan para un objetivo posterior, como la clasificación.
Los modelos generativos permiten aproximar la distribución de los datos de interés a partir su distribución conjunta. Son al menos útiles en dos aspectos:

por un lado, en el análisis de los datos observados en términos de variables latentes y por otro lado, en la generación de datos "falsos pero realistas" a partir de datos reales u observados.

En esta sección se desarrollan los conceptos preliminares de estadística que permiten explicar los modelos generativos.

Espacio muestral, variable aleatoria y probabilidad

Se considera un conjunto S, denominado *espacio muestral*, que consiste en un determinado número de elementos cuya interpretación no es estrictamente necesaria de definir por el momento. A cada uno de los subconjuntos A de S se le asigna un número real $P(A)$, denominado probabilidad, que se define de acuerdo los axiomas de Kolmogorov formulados en 1933 en su libro *Fundamentos de la Teoría de Probabilidades*.

Una variable que toma un valor específico para cada elemento del conjunto S se denomina *variable aleatoria*.

Probabilidad condicional

Si se tienen dos subconjuntos A y B contenidos en el espacio muestral S y en cual $P(B) \neq 0$, entonces se puede definir la *probabilidad condicional* $P(A|B)$ como:

$$P(A|B) = \frac{P(A \cap B)}{P(B)}. \tag{1.1}$$

Además, dos subconjuntos A y B son independientes si

$$P(A \cap B) = P(A)P(B) \tag{1.2}$$

Teorema de Bayes

A partir de lo anterior, es posible demostrar que

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}. \tag{1.3}$$

Esta ecuación, que relaciona las probabilidades condicionales $P(A|B)$ y $P(B|A)$ es conocida como *Teorema de Bayes*.

En el enfoque probabilístico *Bayesiano*, en contraposición con el *Frecuentista*, los elementos del espacio muestral corresponden a hipótesis, que son verdaderas o falsas. El concepto de probabilidad $P(A)$, se interpreta como el grado de creencia de que la hipótesis A es verdadera. En este caso, el espacio muestral debe ser construído de tal manera que sólo una de las hipótesis es verdadera y un subconjunto con varias hipótesis es verdadero si cualquiera de las hipótesis de ese subconjunto es verdadera. Como una de las hipótesis necesariamente debe ser verdadera, entonces $P(S) = 1$.

La interpretación Bayesiana está íntimamente relacionada con el teorema de Bayes 1.3. En este caso, el subconjunto A puede ser interpretado como la hipótesis de que cierta teoría es verdadera y el subconjunto B como la hipótesis de que un experimento dará un resultado particular o se obtengan determinados datos. De esta forma, el Teorema de Bayes toma la siguiente forma:

$$P(teoría|datos) \propto P(datos|teoría) \cdot P(teoría). \tag{1.4}$$

$P(teoría)$ se denomina en inglés *prior probability* y representa la probabilidad de que la teoría sea verdadera, antes de que los datos obtenidos sean observados. La probabilidad $P(datos|teoría)$ se conoce como *likelihood* y es la probabilidad, bajo la asumisión de que la teoría es verdadera, de observar los datos que fueron obtenidos. Finalmente, la probabilidad $P(teoría|datos)$ es conocida en inglés como *posterior probability* y representa la probabilidad de que la teoría sea verdadera dados los datos que han sido obtenidos.

La *prior probability* de los datos $P(datos)$ no aparece explícitamente en 1.4 y es por esto que la ecuación se expresa como una regla de proporcionalidad.

En la estadística Bayesiana, la probabilidad $P(teoría)$ es subjetiva y no hay una regla fija de cómo debe asignarse. Lo importante es que, una vez asignada, la ecuación 1.4 permite determinar cómo cambia el grado de creencia de que la teoría sea verdadera a la luz de los datos experimentales obtenidos.

1.2 Modelos discriminativos y generativos

Se denomina enfoque discriminativo (o modelos a los que hacen uso de este enfoque) al reconocimiento de patrones que se hace a través de la estimación de la *posterior probability* $P(teoría|datos)$.

En contraposición, el reconocimiento de patrones que se realiza a través de la estimación del *likelihood* $P(datos|teoría)$ y la *prior probability* $P(teoría)$ se denomina enfoque generativo (o modelos a los que hacen uso de este enfoque). Estos modelos, al poder estimar estas probabilidades, pueden generar nuevas muestras que son indistinguibles de los datos previamente existentes.

Los modelos generativos son los más utilizados en el aprendizaje automático estadístico, siendo que conocer el modelo generador de los datos es equivalente a conocer prácticamente toda la información sobre los datos y esto permite realizar todo tipo de análisis de datos.
En los modelos generativos no importa el etiquetado de las observaciones (necesario en el aprendizaje supervisado). En su lugar, se intenta estimar la probabilidad de que ocurra la observación en su totalidad.

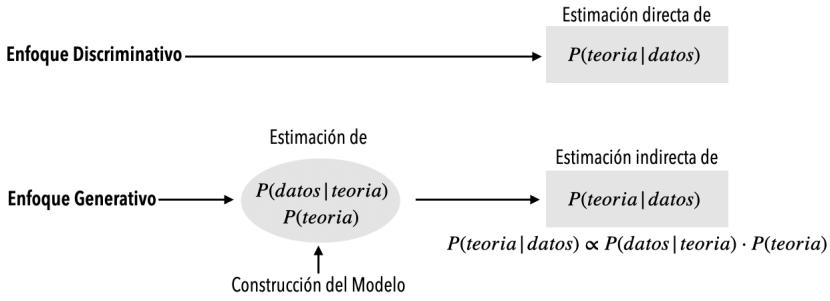

Figura 1.1: Enfoque generativo y disciminativo.

Reconocimiento de patrones

El objetivo del reconocimiento de patrones es clasificar un determinado patrón $\boldsymbol{x}$ a una clase previamente especificada y.

En el problema de reconocimiento de patrones, el subconjunto B (o los datos) suele representarse como un vector $\boldsymbol{x}$ y es llamado patrón, vector de características o variable de entrada. De modo equivalente, el subconjunto A (o la hipótesis) suele representarse como una variable y y se denomina variable de salida, *target* o categoría. Es la clase a la cual el patrón $\boldsymbol{x}$ pertenece.

Tomando en cuenta lo anterior, en el enfoque discriminativo se estima la probabilidad de una determinada variable de salida y, o clase, dado el patrón $\boldsymbol{x}$, mientras que en los enfoques generativos se estima la probabilidad de observar un determinado patrón $\boldsymbol{x}$, luego de especificada una clase y.

Puede ser bastante sencillo de estimar la *prior probability* $P(\boldsymbol{x})$ a través del cociente entre el número de muestras que corresponden a la clase y, n_y, y el total de muestras que se tienen n.

$$\hat{P}(y) = \frac{n_y}{n}, \tag{1.5}$$

Por otro lado, la probabilidad $P(\boldsymbol{x}|y)$ es en general una densidad de probabilidad[I] de una alta dimensión cuya estimación no resulta tan simple y para lo cual existen varios enfoques o procedimientos. Los distintos métodos para estimar densidades de probabilidad se clasifican en *paramétricos* y *no paramétricos*.

Los métodos paramétricos buscan la mejor aproximación de la función de densidad de probabilidad a una familia de funciones (o modelo paramétrico[II]) de densidad de probabilidad conocida. De esta forma, una vez elegido el modelo, el problema se reduce a encontrar los mejores parámetros que representan los datos. Para conseguirlo, existen varias estrategias y entre ellas una de las más conocidas es la estimación de máximo *likelihood*, que consiste sencillamente en encontrar los parámetros del modelo paramétrico seleccionado que maximizan la probabilidad de obtener los datos con los cuales se cuenta.

Por otro lado, los métodos no paramétricos no estiman la función de densidad de probabilidad aproximándola a familias de funciones ya conocidas. Se requiere recurrir a este método cuando la densidad de probabilidad a estimar es de una naturaleza más compleja y no responde a un modelo parámetrico conocido al cual se pueda ajustar. El método no paramétrico más sencillo consiste en

[I] La función de densidad de probabilidad es una función $p(\boldsymbol{x})$ que mapea un conjunto $\boldsymbol{x}$ del espacio muestral a un número entre 0 y 1. Además, la suma de la función de densidad correspondientes a todos los conjuntos en el espacio muestral es 1.

[II] Un modelo paramétrico $p_\theta(\boldsymbol{x})$, es una familia de funciones de densidad de probabilidad que se pueden describir a través de un número finito de parámetros θ.

generar un histograma a partir de los datos y a partir del mismo la densidad de probabilidad.

1.3 Algunos modelos generativos

Los modelos generativos más populares incluyen: los clasificadores de Bayes ingenuos o *Naive Bayes*, los Autodecodificadores Variacionales o *variational autoencoders* y las Redes Generativas Antagónicas o *Generative Adversarial Networks*, que son en los modelos en los que se centra este trabajo.

Clasificador de Bayes ingenuo - Naive Bayes

Es un modelo paramétrico basado en el teorema de Bayes que asume una premisa simple que permite que el número de parámetros a estimar se reduzca drásticamente. Lo que se asume en este modelo es que cada característica x_i es independiente de cualquier otra x_j. Al ser así, la función de densidad de probabilidad se puede expresar como el siguiente producto:

$$p(\boldsymbol{x}) = \prod_{k=1}^{N} p(x_k). \tag{1.6}$$

De esta manera, el problema se reduce a estimar los parámetros de cada característica por separado y luego multiplicarlas para encontrar la probabilidad de cualquier posible combinación.

Autocodificadores Variacionales - *Variational Autoencoders*

Los Autocodificadores variacionales (VAE, por su sigla en inglés) constituyen uno de los modelos generativos de *deep learning* fundamentales. Los fundamentos de estos modelos, fueron presentados en el año 2013 por D. P. Kingma, *et. al.*[3]. Los VAE utilizan el método variacional Bayesiano para lograr su objetivo, que es un conjunto de técnicas que se utilizan para aproximar integrales no-tratables relativas a las distribuciones de probabilidad utilizando inferencia Bayesiana y algoritmos de aprendizaje automático.

Básicamente, los VAE son redes neuronales que tienen dos partes en su arquitectura:

- Codificador o *encoder*: Su objetivo es capturar información de alto nivel sobre los datos de entrada y reducir la dimensión de ellos (compresión). Corresponde a una o varias capas de convolución.
- Decodificador o *decoder*: Su objetivo es retornar los datos de entradas (del espacio latente) al dominio original (decompresión). Corresponde a una o varias capas de deconvolución que, a partir de las características de alto nivel permiten producir nuevos datos.

Cuentan con un punto de conexión entre el codificador y el decodificador que es el *espacio latente*. Este espacio lo constituyen las representaciones de alto nivel extraídas de los datos de entrada por el codificador.

Los VAE pertenecen al grupo de los Autocodificadores [4] y al igual que estos, tienen como objetivo reconstruir los datos de entrada. Su principal diferencia radica justamente en el espacio latente de los VAE. Mientras que los Autocodificadores típicos siguen un enfoque frecuentista, los VAE están basados en el aprendizaje automático Bayesiano. Es por ello que los VAE intentan determinar los parámetros de la distribución de probabilidad del espacio latente (usualmente gaussiana) y los Autocodificadores típicos presentan aleatoriedad en su espacio latente.

En los VAE, la red es entrenada para encontrar los pesos del codificador y decodificador que minimicen la pérdida entre la entrada original y la entrada reconstruída luego de pasar por el codificador y decodificador.

Una aplicación de los VAEs, por mencionar un ejemplo, es la de reducir el ruido de imágenes, siendo que el codificador aprende cuál es la información de la imágen que resulta irrelevante para poder representarla en un espacio de dimensión menor. Esto deja afuera al ruido aleatorio de las imágenes.

Redes Generativas Antagónicas - *Generative Adversarial Networks*

Consisten en dos modelos entrenados simultáneamente: uno (el generador) entrenado para generar datos falsos, y el otro (el discriminador) entrenado para discernir los datos falsos de los ejemplos reales [2].

El término *redes* indica cuál es el tipo de modelo de aprendizaje automático que usualmente se utiliza para construir el generador y el discriminador, que en

este caso son las redes neuronales. Se pueden utilizar distintos tipos de redes neuronales, dependiendo del problema, aunque usualmente se utilizan redes neuronales convolucionales.
El término *generativo* indica el propósito general del modelo: crear nuevos datos.
Finalmente, el término *antagónicas* se refiere a la dinámica competitiva que se establece entre los dos modelos, el generador y el discriminador. Por un lado, el generador tiene por objetivo crear nuevos datos que sean indistinguibles del conjunto de entrenamiento, mientras que el discriminador debe poder ser capaz de distinguir cuáles son los datos creados y cuáles los que corresponden al conjunto de entrenamiento. De esta manera, iterativamente estos dos modelos intentan desafiarse uno a otro y es así como los parámetros se van ajustando para producir datos que se parezcan con gran acierto a los reales.

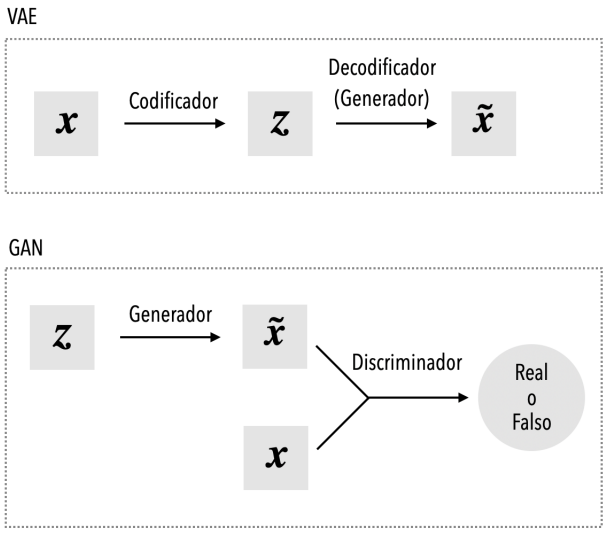

Figura 1.2: Diferencia entre VAEs y GANs.

Las diferencias fundamentales entre los VAEs y las GANs son que para los primeros existe una única función de pérdida (ver la sección 2.1 del capítulo 2), mientras que para las redes GANs se tienen dos funciones de pérdida, una para el generador y otra para el discriminador.
Por otro lado, para los VAEs se tiene una función de pérdida que se pretende

optimizar (minimizar), pero en el caso de las redes GANs, no se cuenta con una métrica explícita que se pretenda optimizar. En su lugar, se tienen dos objetivos que compiten entre ellos y que no se pueden traducir en una única función a optimizar.

En la figura 2.3 se ilustran las diferencias entre ambos enfoques. En el caso de los VAEs, los datos $\boldsymbol{x}$ originales son comprimidos a una representación latente $\boldsymbol{z}$ y luego, a partir de esta, se logra la reconstrucción $\tilde{\boldsymbol{x}}$.
En las GANs, un vector de ruido aleatorio $\boldsymbol{z}$ se utiliza para generar los datos $\tilde{\boldsymbol{x}}$. El discriminador intenta discernir entre los datos producidos por el generador, $\tilde{\boldsymbol{x}}$, y los que son muestrados de la distribución real de datos ($\boldsymbol{x}$).
Tomando en cuenta lo anterior, puede establecerse una analogía entre el decodificador de los VAEs y el generador de las GANs.

2

Redes Generativas Antagónicas

En este capítulo se describe el sustento teórico sobre el cual se basan las GANs, sus componentes, entrenamiento y dificultades asociadas al mismo.

Índice

2.1 Conceptos preliminares

Como se ha mencionado en el capítulo anterior, usualmente para la construcción del generador y discrminador que conforman las Redes Generativas Antagónicas se utilizan, generalmente, redes neuronales convolucionales profundas. A continuación se hace un repaso de los conceptos asociados a estos modelos.

Redes neuronales profundas

La gran mayoría de los sistemas de aprendizajes profundos son redes neuronales y es por este motivo que, en general, el término *Deep Learning* hace alusión a redes neuronales profundas. Sin embargo, cabe destacar que cualquier sistema que emplee múltiples capas (como por ejemplo, las máquinas de Boltzmann) para poder aprender la representaciones complejas de los datos de entrada también se considera *Deep Learning*.

Una red neuronal profunda es una red neuronal con múltiples capas entre las capas de entrada y de salida que son llamadas capas ocultas o *hidden layers*. Cada capa contiene unidades, que están conectadas a las unidades de la capa anterior a través de un conjunto de pesos.

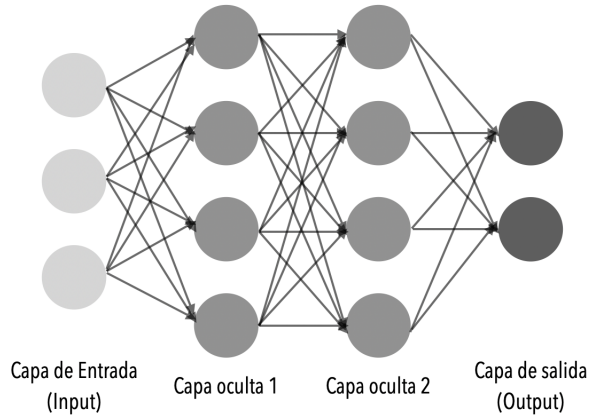

Figura 2.1: Ejemplo de red neuronal con dos capas ocultas densas.

Existen muchos tipos distintos de capas, pero uno de los más comunes son las capas densas, que conecta todas las unidades de la capa directamente a cada unidad de la capa anterior. El propósito de estas capas ocultas radica en que, al combinar las distintas capas, las unidades de cada capa posterior pueden representar aspectos cada vez más sofisticados de la entrada original.

El entrenamiento de la red es el proceso mediante el cual se busca cuál es el conjunto de pesos de cada capa que permite establecer predicciones más precisas.

Funciones de activación

La salida de una determinada unidad y es la suma pesada de las entradas x_i que recibe de capas anterior. Esta suma es la variable de entrada de una función de activación f_{act} no lineal antes de ser enviada a la siguiente capa. Esto permite que la red aprenda patrones complejos, evitando que la salida sea una sencilla combinación lineal de las entradas que recibe.

$$y = f_{act}\left(\sum_i (w_i x_i + b)\right), \tag{2.1}$$

donde w_i representan los pesos de las conexiones y b representa un valor de sesgo o *bias*. Existen varias funciones de activación, como por ejemplo la sigmoidea, *ReLu* y *softmax*, cuyos detalles y propiedades se pueden encontrar en libros básicos de aprendizaje automático.

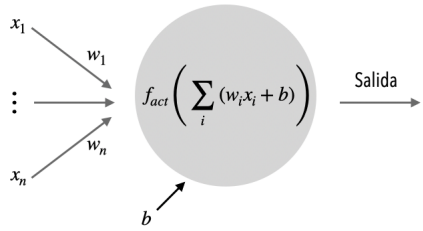

Figura 2.2: Salida de una unidad determinada.

Las redes neuronales profundas pueden ser formuladas a través de una función $f_{\boldsymbol{\theta}}(\boldsymbol{x})$ parametrizada por $\boldsymbol{\theta}$, que contiene todos los parámetros de todas las unidades de la red.

Entrenamiento de la red

La red es entrenada con el fin de encontrar el conjunto de parámetros $\boldsymbol{\theta}$ óptimo. El problema de aprendizaje de la red neuronal se formula en términos de la minimización de una función de pérdida $\mathcal{L}$ o *loss function* que compara la salida predicha por la red con los datos reales.

El proceso de entrenamiento consta de una secuencia iterativa de pasos que podrían resumirse de la siguiente forma:

- *Propagación hacia adelante:* La red recibe muestras de datos del conjunto de entramiento y luego de ser procesados por cada unidad de la red los resultados se propagan hacia adelante hasta llegar a un *output* o valor/es final/es.

- *Medida del error:* En este paso se calcula la función de pérdida $\mathcal{L}$. Esta función retorna un valor promedio por cada ejemplo de entrenamiento y cuanto mayor sea, indica que peor se ha comportado la red para esa observación.

  Distintas funciones de pérdida darán lugar a distintos errores para la misma predicción y, por lo tanto, tendrá un impacto significativo en la evaluación del comportamiento de la red. Tres de las funciones de pérdida más comunes son el error cuadrático medio, la entropía cruzada categórica y la entropía cruzada binaria. La elección de cada una depende de cada caso en particular.

- *Retropropagación:* El algoritmo de retropropagación permite calcular el gradiente de la función de pérdida media respecto a los parámetros de la red, permitiendo encontrar los valores de los parámetros que minimicen la función de pérdida.

  De esta forma, Sabiendo que el vector gradiente da la dirección de máximo crecimiento de la función se puede entonces determinar cómo ajustar los pesos que minimicen la función de pérdida.

- *Actualización de los valores de los parámetros:* Una vez se pasan todos los ejemplos de entrenamiento a la red y se obtienen los gradientes de las funciones de pérdida respecto de cada parámetro de la red, éstos son promediados a través de todos los ejemplos de entrenamiento.

  Finalmente, se actualizan los valores de los parámetros de tal manera de minimizar la función de pérdida, sabiendo que el vector gradiente apunta en la dirección de máximo crecimiento una función. Así, la actualización de los parámetros se establece como:

$$\theta \leftarrow \theta - \eta g, \tag{2.2}$$

donde η es el parámetro *learning rate* que denota en qué cantidad debe moverse el valor del parámetro a ser actualizado en la dirección de minimización de la función de la función de pérdida y g es el promedio de los gradientes respecto al parámetro a actualizar a lo largo de todo el conjunto de entrenamiento:

$$g = \frac{1}{n} \sum_{i}^{n} \nabla_\theta \mathcal{L}_i. \tag{2.3}$$

Como deben ajustarse muchos parámetros resulta importante poder entrenar los modelos rápidamente. Para ello existen los *optimizadores*, que son algoritmos que se utilizan para actualizar los pesos de la red neuronal basado en el gradiente de la función pérdida. Existen muchos optimizadores, entre ellos: Gradiente descendente estocástico, RMSprop, Adam, Adagrad, Adamax, etc.

Capas convolucionales y filtros

Las capas convolucionales permiten tener en cuenta la estructura espacial de las imágenes de entrada. En estas capas, se toman grupos de pixeles cercanos de la imagen de entrada y se aplica el producto escalar contra una matriz pequeña denominada filtro o *kernel*. Esto se hace sucesivamente moviendo el filtro a lo largo de toda la imagen de entrada y así el resultado es un mapa de características. Las características extraídas corresponden a cada posible ubicación del filtro en la imagen original y permite distinguir propiedades dentro de la imagen.

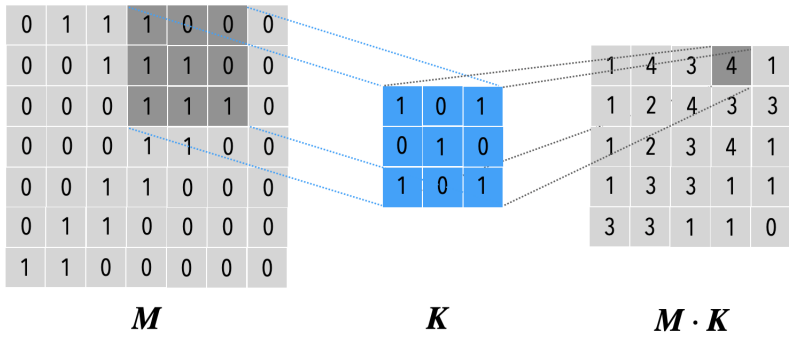

Figura 2.3: Producto escalar entre la matriz de entrada M y el filtro K.

En la práctica no se aplica un sólo filtro sino varios, intentando determinar diferentes propiedades en las imágenes. Los valores de los filtros son pesos que la red aprende a través del entrenamiento. Inicialmente, estos valores son aleatorios y a medida que la red se entrena, cada filtro toma valores que permiten detectar distintas características de la matriz de entrada.

Los filtros tienen tantos canales como tenga imagen de entrada, es decir, si se trata de una imagen color de 3 canales, el filtro que se aplica sobre esa imagen también tendrá 3 canales.

Dos parámetros importantes son el parámetro *stride* (o zancada) y *padding*. El primero determina el corrimiento del filtro sobre la variable de entrada durante la operación convolución. Incrementar el parámetro *stride* reduce la dimensión de la salida y es una técnica que se utiliza, por ejemplo, en los Autodecodificadores Variacionales cuando se construye el codificador. El *padding* se utiliza para que la variable de salida tenga la misma dimensión que la variable de entrada. Se agregan ceros en los bordes de la imagen de entrada de forma que la salida tenga exactamente la misma dimensión.

Capas de *pooling*

Las capas de *pooling* tienen la función de reducir la dimensionalidad de los mapas de características obtenidos al aplicar los filtros, pero manteniendo la información más importante [16]. De esta forma, se reduce el número de parámetros en la red, controlando el posible sobre-entrenamiento y haciendo la red más pequeña en memoria.

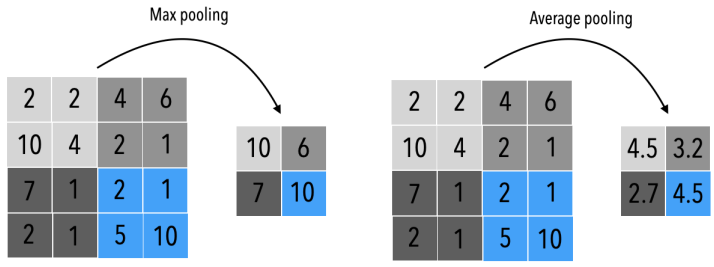

Figura 2.4: Ejemplo de aplicación de las funciones *max pooling* y *average pooling*.

Existe una variedad de funciones de *pooling* y dentro de las más comunes se encuentran las funciones de *average pooling* y *max pooling*. En el primer caso, la matriz de dimensionalidad reducida se calcula promediando un conjunto de valores de la matriz original para producir el resultado de cada celda final. En el segundo caso, el valor de las celdas de la matriz reducida se obtienen tomando el máximo de un conjunto de valores de la matriz original, tal y como se muestra en el ejemplo de la figura 2.4.

Capas convolucionales transpuestas y capas de *upsampling*

Las capas convolucionales permiten reducir a la mitad el tamaño de la matriz o el tensor de entrada si se configura el parámetro *stride*= 2. Por el contrario, las capas convolucionales transpuestas tienen el comportamiento opuesto. Es decir, al configurar el parámetro *stride*= 2 duplican el tamaño de la matriz o tensor de entrada. Para lograr esto, se agrega un *padding* de ceros entre los píxeles de la imagen original antes de realizar las operaciones de convolución con el filtro seleccionado.

Otra forma de aumentar el tamaño de la matriz o tensor original es a través de las capas de *upsampling*. Estas capas repiten cada fila y columna de su entrada de manera de duplicar su tamaño.

Ambos métodos pueden utilizarse para lograr el tamaño de salida que se requiera, la elección de cada uno dependerá del problema en particular.

Capas de normalización o normalización por lotes

A medida que el error se retropropaga a través de la red, el cálculo del gradiente en las primeras capas puede, en ocasiones, volverse exponencialmente grande, causando altas fluctuaciones en los pesos calculados.
Para evitar esto, se debe asegurar que los pesos de la red se mantengan en un rango de valores razonables. Una manera de hacerlo es incorporar una capa de normalización que calcula la media y desviación estándar de la entrada y lo normaliza restando la media y diviendo por la desviación estándar.
Lo ideal es que la normalización se haga usando la media y desviación estándar de todo el conjunto de entrenamiento, pero también en algunos casos es conveniente hacer una normalización por lotes, en la que se utiliza la media y desviación estándar de cada lote de entrada.

Regularización

Si un algoritmo se comporta bien cuando se utiliza el conjunto de datos de entrenamiento, pero no lo hace con el conjunto de datos de prueba, es muy probable que la red haya sufrido un sobreajuste u *overfitting*. Para contrarrestrar este problema se utilizan técnicas de regularización, cuyo objetivo es penalizar el modelo si comienza a sobreajustarse.

En *Deep Learning*, una de las formas más usuales de regularizar el modelo es agregando en la función de pérdida un término que penaliza la complejidad del mismo. Dependiendo de cómo sea determinada la complejidad del modelo se tienen distintos tipos de regularización.
Por ejemplo, en la regularización Lasso, también llamada L1, la complejidad del modelo se mide como la media del valor absoluto de los coeficientes del modelo. O bien, en la regularización Ridge, también llamada L2, la complejidad del modelo se mide como la media del cuadrado de los coeficientes del modelo. Otros métodos de regularización combinan las regularizaciones L1 y L2. La selección de un método u otro depende del problema en concreto que se quiera resolver.

Además, pueden incluirse lo que se conocen como capas *dropout*. Durante el entrenamiento, cada capa *dropout* selecciona un conjunto aleatorio de unidades de la capa anterior y fuerza su salida a 0. De esta manera, el modelo no se vuelve sobredependiente de determinadas unidades y la generalización es mayor, reduciendo el sobreajuste.
En general, las capas *dropout* son utilizadas luego de una capa densa, pero también pueden verse luego de capas convolucionales.

2.2 GANs

Las Redes Generativas Antagónicas (GANs, por su sigla en inglés), fueron introducidas en el año 2014 por Ian J. Goodfellow y colaboradores en el artículo *Generative Adversarial Nets* [2]. El modelo combina de forma muy original el aprendizaje profundo con la teoría de juegos y permite la generación de datos a partir de ejemplos con muy buenos resultados.
A continuación se describirán sus componentes, entrenamiento y las dificultades asociadas al mismo.

2.2.1 Componentes

Las GANs consisten en dos modelos (en general, redes neuronales convolucionales) entrenados simultáneamente: uno (el generador) entrenado para generar datos, y el otro (el discriminador) entrenado para discernir los datos que provienen del generador (falsos) de los ejemplos reales [2].

Al comienzo del proceso, la salida del generador es ruidosa y el discriminador predice de manera aleatoria, pero a medida que se van entrenando ambas redes neuronales, el generador mejora para eludir al discriminador y por tanto, el discrinador también mejora para poder discernir entre datos verdaderos y falsos, los producidos por el generador.

Formalmente, tanto el generador como el discriminador se representan por funciones diferenciables $G(\boldsymbol{z}, \theta_G)$ y $D(\boldsymbol{x}, \theta_D)$, respectivamente, cada una con su propia función de pérdida y siendo θ_G y θ_D los parámetros a entrenar (pesos) de las dos redes. Cada red ajustará sus propios parámetros al ser entrenada.

El generador

El propósito del generador (G) es "engañar" al discriminador (D) haciendo pasar como datos del conjunto de entrenamiento, aquellos que fueron producidos a partir de un vector de ruido aleatorio.
Debe capturar características del conjunto de entrenamiento de tal forma que las muestras generadas sean indistinguibles del mismo.
El generador podría pensarse como un reconocedor de patrones a la inversa. Es decir, en lugar de reconocer los patrones, el generador aprende a producirlos.

Formalmente, el generador toma un vector de ruido aleatorio $\boldsymbol{z}$ muestreado a partir de una distribución $p_z(\boldsymbol{z})$ (en general una distribución Gaussiana) para producir una muestra de datos falsos $\tilde{\boldsymbol{x}}$, es decir:

$$G(\boldsymbol{z}) = \tilde{\boldsymbol{x}}. \tag{2.4}$$

El discriminador

El objetivo del discriminador es predecir si los datos corresponden al conjunto de entrenamiento (reales) o fueron producidos por el generador (ficticios o falsos).
Formalmente, discrimina entre $\tilde{\boldsymbol{x}}$, la muestra falsa, y $\boldsymbol{x}$, los datos muestreados de la distribución real de datos $p_{datos}(\boldsymbol{x})$.

La arquitectura que suele usarse para construir el discriminador es la que se utiliza para un problema de clasificación de imágenes supervisado, siendo que el discriminador opera como tal.
En general, la arquitectura del discriminador es aproximadamente simétrica a la del generador (ver, por ejemplo, las figuras 3.1 y 3.2 del siguiente capítulo). Esto permite que la convergencia del entrenamiento sea alcanzada más fácilmente.

2.2.2 Entrenamiento

En la figura 2.5 se ilustra el proceso de entrenamiento de una GAN. Como se puede anticipar, el entrenamiento del generador es bastante más complejo que el entrenamiento del discriminador, que no es más que un clasificador binario.

El discriminador puede ser entrenado con un conjunto de entramiento dentro del cual algunas entradas son reales (seleccionadas aleatoriamente) y otras, que provienen del generador, se etiquetan como falsas. La respuesta del discriminador debería ser, entonces, 1 si los datos de entrada son reales, o 0 si esos datos son falsos, creados por el generador.

En el caso del generador, debe entrenarse teniendo como única condición que los datos que se generarán deben "eludir" al discriminador. Es decir, debe entrenarse con el fin de minimizar minimizar $log(1 - D(G(\boldsymbol{z})))$ que es equivalente a $log(1 - D(G(\tilde{\boldsymbol{x}})))$.
Para poder lograr esto, la salida del generador debe ser la entrada del discriminador, de tal manera que la salida de este modelo combinado nos da la probabilidad de que los datos generados son reales, de acuerdo con el discriminador.

Así, el generador obtiene *feedback* del discriminador, que utiliza para generar datos más parecidos al conjunto de entrenamiento.

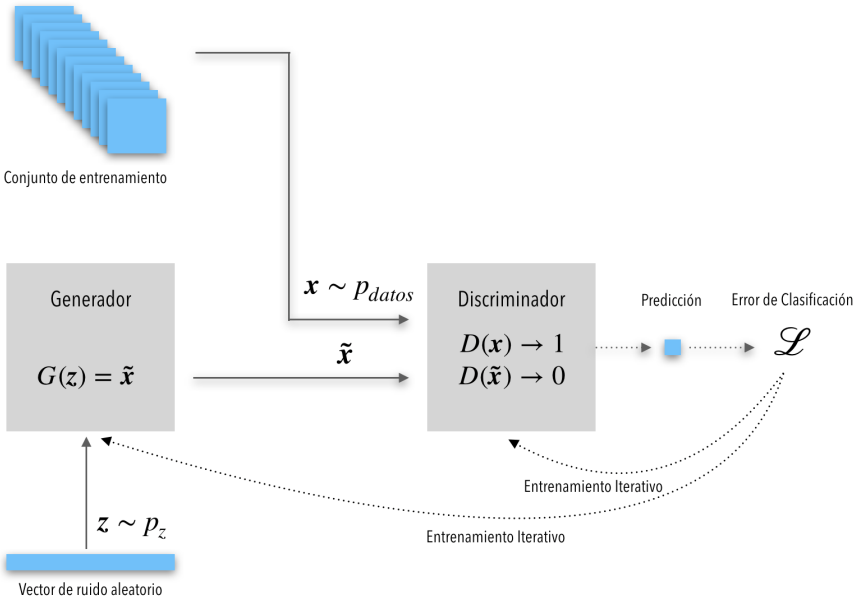

Figura 2.5: Entrenamiento de una red GAN.

En las sucesivas iteraciones, cuando se entrena el discriminador, se actualizan sus pesos para maximizar la precisión de clasificación. En cambio, cuando se entrena el generador (utilizando el modelo combinado), se actualizan los pesos del generador de manera tal que maximicen la probabilidad de que el discriminador clasifique a los datos generados a través del vector de ruido aleatorio como reales.

Es fundamental que cuando se entrena el modelo combinado, los pesos del discriminador queden fijos, de forma que sólo se actualicen los del generador. En caso contrario, el discriminador aumenta las probabilidades de predecir los datos del generador como *reales*, pero no por mérito del generador, que es lo que se espera.

Equilibrio de Nash y fin del entrenamiento

En una GAN, se tienen dos redes que pretenden alcanzar objetivos contrapuestos, es decir, cuando una red mejora, la otra empeora. En este contexto, es difícil pensar en un criterio inmediato para darle fin al entrenamiento. La manera más sencilla de formular el aprendizaje de una GAN es como si fuera un *juego de suma cero*.

Dentro de la matemática aplicada, particularmente en la teoría de juegos, se llaman *juegos de suma cero*, a aquellos sistemas en los cuales la ganancia o pérdida de un jugador, se equilibra con exactitud con las pérdidas y ganancias del otro jugador. Es decir, si se suman las ganancias y las pérdidas totales, esto suma cero.

En 1928, John von Neumann desarrolló el teorema *minimax* [5], que establece que en los juegos de suma cero, donde cada jugador conoce de antemano la estrategia de su oponente y sus consecuencias, existe una estrategia que permite a ambos jugadores minimizar la pérdida máxima esperada. Esta estrategia, o método, es denominada *minimax*.
Las GANs utilizan en su entrenamiento el método de decisión minimax para minimizar la pérdida máxima esperada.En el contexto de las GANs, puede pensarse entonces que el generador y el discriminador juegan un juego de suma cero con función objetivo $\mathcal{L}(G, D)$[2]:

$$\min_G \max_D \mathcal{L}(G, D) = \mathbf{E}_{\boldsymbol{x} \sim p_{datos}}[logD(\boldsymbol{x})] + \mathbf{E}_{\boldsymbol{z} \sim p_z}[log(1 - D(\tilde{\boldsymbol{x}}))]$$

En el contexto de juegos de suma cero, la solución *minimax* es equivalente a lo que se denomina *Equilibrio de Nash*, que es el punto en el cual ningún jugador puede mejorar su situación a través de sus acciones.
En las GANs, si se logra el Equilibrio de Nash es el momento oportuno para darle fin al entrenamiento.

En su artículo original, Goodfellow et. al. [2] mostraron que el Equilibrio de Nash se alcanza cuando se cumplen dos condiciones:

- El generador produce muestras que son indistinguibles de los datos del conjunto de entramiento ($p_g = p_{datos}$).
- El discriminador puede, como mucho, adivinar aleatoriamente cuando una muestra es real o falsa.

Es decir, si las muestras generadas son indistinguibles del conjunto de datos, el discriminador, al no poder diferenciarlas, si recibe para analizar la mitad de muestras reales y la otra mitad falsas, lo mejor que puede hacer es adivinar aleatoriamente, con un 50 % de probabilidades de acierto.

En [2] también queda demostrado que el algoritmo descripto de manera ilustrativa en la sección 2.2.2, optimiza la función 2.5 obteniendo el resultado deseado.

Cuando se alcanza el equilibrio se dice que la red converge. La convergencia de las redes GAN no es un problema trivial y constituye una de las mayores dificultades que presentan estos modelos.

2.2.3 Dificultades

Las GANs presentan varios desafíos a eludir durante su entrenamiento [6] y que son actualmente temas de investigación. Entre ellos, los problemas más usuales que surgen en el entrenamiento de una GAN son:

- **No convergencia**
  El generador y el discriminador no logran alcanzar un equilibrio. La función de pérdida del generador y discriminador empiezan a oscilar sin poder lograr a largo plazo una estabilidad.
  Si bien es común en las GAN que en un comienzo las funciones de pérdida oscilen, a medida que transcurre el entrenamiento el objetivo es que se logre una estabilidad. Cuando esto no ocurre, las muestras son producidas por el generador, pero su calidad no mejora.

- **Colapso modal**
  Esto ocurre cuando el generador produce muestras similares aunque las entradas sean de muy diversas características. El generador encuentra que un conjunto pequeño de muestras engañan al discriminador y entonces no es capaz de producir otras. En estos casos, el gradiente de la función de pérdida queda estancado en un valor cercano a 0.

- **Pérdida no informativa**
  Aunque parezca natural pensar que cuanto menor sea la pérdida del generador, mayor será la calidad de las muestras que produce, esto no resulta tan inmediato. La pérdida del generador debe ser comparada con la del discriminador, que se encuentra en constante mejora. Por lo tanto, no es tan sencillo evaluar la mejora del modelo. El generador podría estar produciendo muestras cada vez de mayor calidad, aun cuando la función de pérdida se vaya incrementando.

En los últimos años se han ido desarrollando técnicas que tienen el objetivo de sortear las dificultades que presentan para su entrenamiento las GAN.

Por ejemplo, las Wasserstein GAN (WGAN) o las Wasserstein GAN-Gradient Penalty (WGAN-GP) [7, 8] son dos enfoques que permiten mejorar la inestabilidad de las GAN y lidiar parcialmente con el problema del colapso modal. Sin embargo, este es un tema de investigación abierto en la actualidad.

3

GANs y aplicaciones

En este capítulo se describen algunas de las distintas arquitecturas de GANs que se han implementado hasta el momento y las aplicaciones que estas variedades (y otras) permiten.

Índice

3.1 Distintas arquitecturas de GANs

Desde que fueron propuestas las redes GANs, muchas arquitecturas distintas han sido sugeridas para encarar diversos problemas. En [9] se hace una extensa revisión sobre múltiples arquitecturas de redes GANs, entre ellas: cGAN, BiGAN, infoGAN, AC-GAN, etc.

Cada semana, diversos artículos científicos sobre GANs son publicados en los que nuevas arquitecturas (con nombres nuevos) son propuestas. Siendo así, la revisión de todas las arquitecturas de GANs que han sido propuestas hasta el momento excede ampliamente el objetivo de este trabajo y es por esto que, a modo de ejemplo, en esta sección se han seleccionado algunas arquitecturas

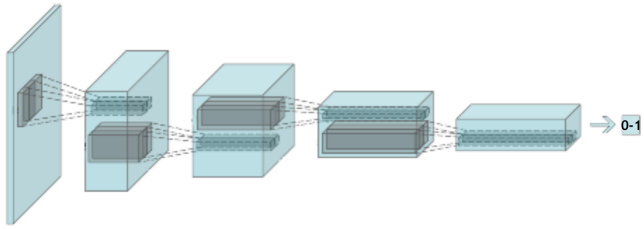

Figura 3.2: Ejemplo de arquitectura de un Discriminador, teniendo en cuenta el Generador de la figura 3.1

para ser descriptas, como las DCGAN, cGAN, cycleGAN y SRGAN, que se han comportado bien en la práctica de acuerdo a las últimas publicaciones.

DCGAN

Las DCGAN (por *Deep Convolutional Generative Adversarial Network*) [10] fueron las primeras redes generativas antagónicas capaces de crear imágenes de buena resolución utilizando un único generador.

En esta arquitectura, tanto el generador como el discriminador tienen una arquitectura similar, pero opuesta, basada en una serie de capas convolucionales, como puede verse en las figuras 3.1 y 3.2.

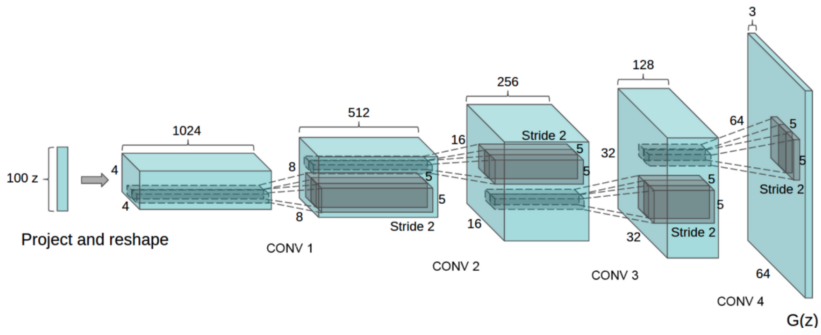

Figura 3.1: Ejemplo de arquitectura de un Generador, tomado de [10]

En el generador se produce un aumento progresivo de la dimensión del vector de ruido aleatorio original, que se logra a través de proyecciones, reajustes de tamaños, capas de convolución transpuestas o capas de *upsampling*.

Por otro lado, el discriminador se compone de 4 capas convolucionales en las que para reducir la dimensión en cada paso se configura el parámetro *strides* $\neq 1$. En la última capa, se agrega una capa densa que proyecta la entrada de esa capa en un único número que se encontrará dentro del rango $[0, 1]$ (se interpreta como la probabilidad de que los datos procesados sean reales o no). Esto se logra utilizando una función sigmoidea como función activadora.

cGAN

Las cGANs o GANs condicionales fueron propuestas en el año 2014 en el trabajo *Conditional Generative Adversarial Nets* [11]. Son GANs en las cuales el generador G y el discriminador D están condicionados a cierta información adicional c. Esta información condicional se refiere en general a etiquetas en los datos.

El generador, utiliza el vector de ruido aleatorio $\boldsymbol{z}$ y la información adicional c para generar muestras falsas $G(\boldsymbol{z}, c) = \tilde{\boldsymbol{x}}|c$. Su objetivo es producir muestras lo más parecidas a las reales posibles dada una cierta condición.

El discriminador, recibe las muestras reales etiquetadas $(\boldsymbol{x}, c)$ y las muestras falsas producidas por el generador $(\tilde{\boldsymbol{x}}|c, c)$ y aprende a discriminar pares de datos reales.

En este caso, la función de pérdida se define de la siguiente manera:

$$\min_G \max_D \mathcal{L}(G, D) = \mathbf{E}_{\boldsymbol{x} \sim p_{datos}}[logD(\boldsymbol{x}, c)] + \mathbf{E}_{\boldsymbol{z} \sim p_z}[log(1 - D(G(\boldsymbol{z}, c)))]$$

CycleGAN

Las *CycleGANs*, o GANs cíclicas, surgieron en el año 2017 al ser publicado el trabajo *Unpaired Image-to-Image Translation using Cycle-Consistent Adversarial Networks* [12] por un grupo de investigadores del *Berkeley AI Research (BAIR) laboratory.*

Las *CycleGANs* permiten transformar imágenes desde un dominio a otro. Para lograrlo usan una arquitectura que se componen de dos GANs, es decir, cuentan con un total de dos generadores y dos discriminadores.

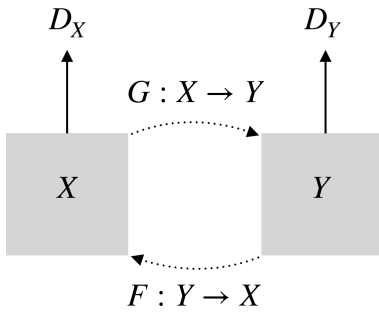

Figura 3.3: Componentes de una red *CycleGAN*

El primer generador G convierte imágenes desde un dominio X a un dominio Y, mientras que, a la inversa, segundo generador F produce imágenes desde el dominio Y al X:

$$G : X \to Y \tag{3.2}$$
$$F : Y \to X \tag{3.3}$$

Por otro lado, el primer discriminador D_X se entrena para poder determinar imágenes reales del dominio Y de aquellas que son producidas por el generador G . El segundo discriminador D_Y se entrena para poder distinguir imágegnes reales del dominio X de aquellas que son producidas por el Generador F.

$$D_X : \text{ Distingue } X \text{ de } F(Y) \tag{3.4}$$
$$D_Y : \text{ Distingue } Y \text{ de } G(X) \tag{3.5}$$

3.2 Aplicaciones de GANs

El descubrimiento de aplicaciones de las GANs es un campo de investigación activo. En esta sección se describen algunas aplicaciones de estas redes que

aparecen en la literatura científica.

- **Generación de imágenes**

  Las GANs pueden ser utilizadas para generar imagenes realistas a partir de un conjunto de entrenamiento de imagenes de ejemplo. Esto tiene múltiples aplicaciones, como por ejemplo, la creación de logos [13], de ejemplos para incrementar el tamaño de un *dataset*, lo que se conoce como *Data Augmentation* [14], creación de personajes de caricaturas [15], de rostros de humanos [16], entre muchas otras.

  Un ejemplo de aplicación de generación de imagenes con GANs en la ciencia, más exactamente en el área de Física de Altas Energías (HEP, por sus siglas en inglés) es la de creación de imagenes de *jets* de partículas [17], que son representaciones 2D de la deposición de la energía de las particulas producidas en un acelerador cuando interactuan con los calorímetros del mismo. Esto permite hacer simulaciones computacionales rápidas, que de otro modo requieren un gran poder computacional.

  También se han relizado trabajos en los que GANs se utilizan para completar partes faltantes en imágenes, en lugar de crear imágenes completas [18, 19].

- **Síntesis de imagenes a partir de texto**

  Al mezclar los campos de procesamiento natural de lenguage y el de visión por computadora, un problema desafiante que surge es el de dada una descripción de una figura, crear la imagen correspondiente. [20, 21].

  Reed et al. [21] fueron los primeros en proponer una solución con resultados prometedores al problema de síntesis de imagen a partir de texto. La propuesta fue dividir el problema en dos subproblemas: por un lado, encontrar una manera visual y discriminativa para representar las descripciones de imagenes y por otro lado, utilizar esta representación para poder crear imagenes realistas.

  Si bien hasta el momento no se logran producir imagenes muy fieles a las descripciones, es una aplicacion de GANs que está siendo ampliamente explorada y se espera que esto cambie en un futuro cercano.

- **Conversión imagen-imagen**

  Usualmente se utilizan *CycleGANs* para convertir imágenes de un dominio a otro. En el trabajo [12] se muestran estas conversiones mediante los ejemplos de imágenes de zebras a imágenes de caballos, fotografías a pinturas del estilo de Monet y fotos de invierno a fotos de verano.

  Otro uso común de las *CycleGANs* en este tipo de aplicaciones es el de transformar un mismo rostro a través de distintas edades [22]. Esto podría tener varias aplicaciones, como facilitar la búsqueda de niños desaparecidos, construir sistemas de reconocimiento facial más robustos que no dependan de las edades de las personas, entretenimiento, etc.

- **Generación de imágenes de alta resolución**

  Las GANs pueden ser entrenadas para, a partir de imágenes de baja resolución, generar su contraparte de alta resolución. Esto puede tener aplicaciones en varios campos, como en imágenes médicas para permitir mejor detección de patologías [23, 24, 25, 26, 27], en imágenes astronómicas, en determinadas escenas para permitir el reconocimiento de objetos, en imágenes forenses, etc.

  También se han utilizado GANs de super-resolución para aumentar la resolución en videojuegos viejos. Algunos juegos que han implementado esta tecnología son: *Final Fantasy VIII*, *Resident Evil Remake HD Remaster* y *Max Payne* [28].

  Este tema es tratado con mayor detalle en el siguiente capítulo.

Además de las aplicaciones antes mencionadas, existen muchas otras de enorme interés que han sido y continúan siendo estudiadas, como por ejemplo la síntesis de videos [29], que podría servir para generar contenido en un menor tiempo al precisado si se hace la tarea de forma manual, la predicción de cuadros de videos [30, 31, 32] o el mapeo de imágenes 3D a partir de cortes en 2D [33], etc, etc.

4

GANs de super-resolución

En este capítulo se implementa en código, a través de las librerías de Python Keras y TensorFlow, la Red Generativa Antagónica de Super-resolución(SRGAN) propuesta en [1]. El entrenamiento de esta SRGAN se lleva a cabo en el entorno Google Colaboratory.

Índice

4.1 Introducción

La resolución espacial de una imagen se corresponde al número de pixels[I] que se utilizan para construir dicha imagen. Así, una imagen de baja resolución (LR) contiene una baja densidad de pixels que no permiten evidenciar detalles de la misma.

[I]Menor unidad homogénea que conforma una imagen.

La super-resolución (SR) de imágenes es un problema ampliamente estudiado en el campo de visión por computadora. El objetivo es generar una o más imágenes de alta resolución a partir de una o más imágenes de baja resolución.

Este tema ha recibido gran antención en el último tiempo debido a la gran cantidad de aplicaciones que tiene, como la mejora del desempeño de sistemas reconocedor de patrones en visión por computadora, en sistemas de seguridad donde podría requerirse determinar con mayor precisión algún punto en una escena, en el área médica, en la cual adquirir imágenes de alta resolución suele ser una tarea complicada que involucra someter al paciente a mayores tiempos en la adquisición de datos mientras se hace el diagnóstico, etc.

Los métodos de SR se puede clasificar en dos categorías: Super-resolución a partir de multi-imágenes o imagen única de super-resolución. En la super-resolución a partir de multi-imágenes, se utiliza más de una imagen de baja resolución para generar la imágen final de alta resolución. Estas imágenes pueden ser de distintos ángulos.
Por otro lado, las imágenes únicas de super-resolución (SISR, por sus siglas en inglés) son imágenes de alta resolución generadas a partir de una única imagen de baja resolución. El problema de generar una imágen de alta resolución a partir de una de baja resolución es un problema indeterminado, que no presenta solución única.

4.2 SRGANs

El modelo SRGAN (*Super Resolution Generative Adversarial Network*) fue propuesto en 2016 por un grupo de investigadores de la empresa Twitter [1].
La principal innovación de este modelo es su función de pérdida, llamada *función de pérdida perceptual*, que permite mejorar el realismo de la imagen de salida. El modelo presentado corresponde a la generación de imágenes de alta resolución a partir de una única imagen de baja resolución.

En esta sección se implementará una SRGAN siguiendo la arquitectura presentada en [1] mediante la librería de alto nivel de Python para el desarrollo de redes neuronales: *Keras*.

Keras tiene una API[II] muy fácil de usar y proporciona numerosos bloques de construcción útiles que pueden ser conectados para crear arquitecturas de aprendizaje profundo altamente complejas.
Para el entrenamiento de las redes, *Keras* utiliza una de las tres librerías como *backend* para este propósito: *TensorFlow*, *CNTK*, o *Theano*. En este trabajo se utiliza *TensorFlow*, que es una librería de Python de código abierto para el aprendizaje automático desarrollada por Google.

La arquitectura de la SRGAN presentada en [1] es compleja y consta de varios bloques, tanto en la construcción del generador como del discriminador.
A continuación se describen cada una de las partes que conforman cada red y su implementación en código a través de *Keras*. Posteriormente, se analiza la función de pérdida que caracteriza a este tipo de GANs y se analizan los resultados obtenidos mediante la red implementada.

4.2.1 Arquitectura

En esta sección se describe la arquitectura del generador y discriminador de la SRGAN de [1] y cómo se pueden implementar utilizando *Keras*.

El en Apéndice A se encuentra el código completo de la arquitectura del generador y del discriminador. Además, este código se puede encontrar en el repositorio de *GitHub* asociado a este proyecto [34].

Generador

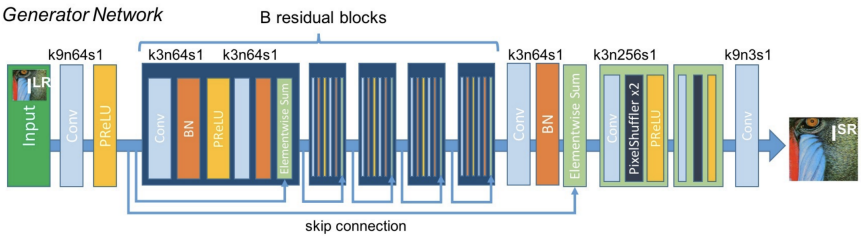

Figura 4.1: Arquitectura del generador de la SRGAN, donde k es el tamaño del filtro, n es la dimensión del mapa de características (capa convolucionada) y s indica el valor del parámetro *stride*. Imagen tomada de [1].

[II]API: *Application Programming Interface.*

El generador (figura 4.1 toma como input una imagen de baja resolución de dimensión $64 \times 64 \times 3$ y, luego de una serie de bloques, genera una imagen de super-resolución de dimensión $256 \times 256 \times 3$. Cuenta con la siguiente estructura:

- **Capa de entrada:** capa de entrada para tomar como *input* la imagen de baja resolución de dimensión $64 \times 64 \times 3$.

  La implementación en *Keras* queda:

```
input_shape = (64, 64, 3)      #Low Resolution image
↪  dimension

#Input Layer for the Low Resolution image
input_lr = Input(shape=input_shape)
```

- **Bloque pre-residual:** Este bloque contiene una capa convolucional con $stride = 1$ y tamaño del filtro 9. Utiliza la función $PReLU$[III] como función activadora.

  La implementación en *Keras* de este bloque puede escribirse de la siguiente manera:

```
    #Pre-Residual Block
    pre_res = Conv2D(filters=64, kernel_size=9, strides=1,
    ↪  padding='same')(input_lr)
    pre_res = PReLU(shared_axes=[1,2])(pre_res)
```

- **Bloque de redes residuales:** Las redes residuales o *ResNet* fueron propuestas en 2015 por He et al. [35] y se utilizan para evitar el problema

[III] La función activadora *PReLU (Parametric Rectified Linear Unit)* $f(y_i)$ se define de la siguiente manera:

$$f(y_i) = \begin{cases} y_i & \text{si } y_i > 0. \\ a_i y_i & \text{si } y_i \leq 0. \end{cases}$$

Donde y_i es la entrada en el canal i y a_i es un parámetro ajustable correspondiente a una pendiente negativa.

del desvanecimiento del gradiente[IV]. La redes residuales se inspiran en el hecho biológico de que algunas neuronas (las piramidales) se conectan con neuronas en capas no necesariamente contiguas. Así, la particularidad que tienen las *ResNet* es que establecen conexiones salteándose algunas capas.

Cada bloque de redes residuales se compone por dos redes convolucionales seguidas por una capa de normalización por lotes. Finalmente, hay una capa de adición que calcula la suma de la entrada al bloque y la salida de la última capa de normalización.

La implementación en *Keras* de este bloque se puede escribir mediante una función *residual_block*:

```
#Residual Block
def residual_block(input):

    res = Conv2D(64, kernel_size=3, strides=1,
    ↪  padding='same')(input)
    res = BatchNormalization(momentum=0.8)(res)
    res = PReLU(shared_axes=[1,2])(res)
    res = Conv2D(64, kernel_size=3, strides=1,
    ↪  padding='same')(res)
    res = BatchNormalization(momentum=0.8)(res)
    res = Add()([res, input])

    return res

res_blocks = 16                 #Number of Residual
↪  Blocks

#Residual Blocks
res = residual_block(pre_res)
for i in range(res_blocks-1):
    res = residual_block(res)
```

[IV]El problema del desvanecimiento del gradiente surge cuando en una red existen muchas capas y es probable que el valor del gradiente al llegar a las primeras capas sea muy próximo a cero, por lo que la variación de los pesos es muy pequeña impidiendo entrenar bien las primeras capas, cuyos errores se arrastran a toda la red, y consumiendo mucho tiempo de entrenamiento.

- **Bloque post-residual:** El bloque post-residual contiene una capa de convolución seguida de una capa de normalización por lotes y una capa de adición. La implementación en *Keras* es entonces:

```
#Post-Residual Block
post_res = Conv2D(64, kernel_size=3, strides=1,
↪  padding='same')(res)
post_res = BatchNormalization(momentum=0.8)(post_res)
post_res = Add()([post_res, pre_res])
```

- **Bloque de *upsampling*:** El generador contiene dos bloques de *upsampling* en los que cada uno están compuestos por una capa de *upsampling* y una capa de convolución que utiliza una función *PReLU* como activadora.

```
#Upsampling Blocks
upsamp_1 = UpSampling2D(size=2)(post_res)
upsamp_1 = Conv2D(filters=256, kernel_size=3, strides=1,
↪  padding='same')(upsamp_1)
upsamp_1 = PReLU(shared_axes=[1,2])(upsamp_1)

upsamp_2 = UpSampling2D(size=2)(upsamp_1)
upsamp_2 = Conv2D(filters=256, kernel_size=3, strides=1,
↪  padding='same')(upsamp_2)
upsamp_2 = PReLU(shared_axes=[1,2])(upsamp_2)
```

- **Capa de convolución final:** La última capa de convolución utiliza una función *tanh* como función activadora y tiene como *output* una imagen de dimensiones $(256, 256, 3)$

```
#Output Layer for the High Resolution image
output_hr = Conv2D(filters=3, kernel_size=9, strides=1,
↪  padding='same', activation='tanh')(upsamp_2)
```

El código completo del generador se encuentra en el Apéndice A.1.

Discriminador

La arquitectura del Discriminador puede verse en la figura 4.2. Cuenta con la siguiente estructura:

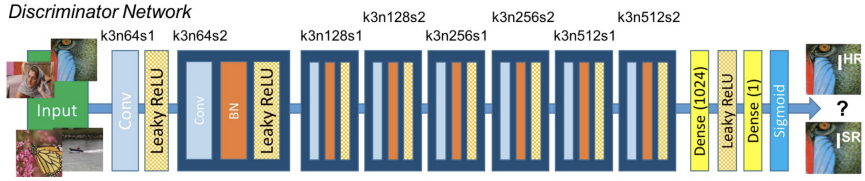

Figura 4.2: Arquitectura del Discriminador de la SRGAN, donde k es el tamaño del filtro, n es la dimensión del mapa de características (capa convolucionada) y s indica el valor del parámetro *stride*. Imagen tomada de [1].

- **Capa de entrada:** La imagen de alta resolución producida por el generador es tomada como *input* para ser discriminada.

```
input_shape = (256, 256, 3)      #High Resolution image
↪  dimension

#Input Layer for the High Resolution image
input_hr = Input(shape=input_shape)
```

- **Bloque de capas convolucionales:** Se extraen las características de la imagen mediante una serie de 8 capas convolucionales. La implementación en *Keras* puede hacerse definiendo una función *conv_block*:

```
#Convolution Block
def conv_block(input, filter, strides=1, batchnorm=True):
    conv = Conv2D(filters=filter, kernel_size=3,
    ↪  strides=strides, padding='same')(input)
    conv = LeakyReLU(alpha=0.2)(conv)

    if batchnorm:
        conv = BatchNormalization(momentum=0.8)(conv)
    return conv

#Convolution blocks
```

```
conv = conv_block(input_hr, filter, batchnorm=False)
conv = conv_block(conv, filter, strides=2)
conv = conv_block(conv, filter*2)
conv = conv_block(conv, filter*2, strides=2)
conv = conv_block(conv, filter*4)
conv = conv_block(conv, filter*4, strides=2)
conv = conv_block(conv, filter*8)
conv = conv_block(conv, filter*8, strides=2)
```

- **Capa Densa y capa de salida:** Se tiene una capa densa de dimensón 1024 y finalmente la salida es juzgada a través de una función sigmoidea:

```
#Dense layer
dense_layer = Dense(filter*16)(conv)
dense_layer = LeakyReLU(alpha=0.2)(conv)

#Output classification layer
output_class = Dense(1, activation='sigmoid')(dense_layer)
```

El código completo del discriminador se encuentra en el Apéndice A.2.

4.2.2 Función de pérdida perceptual

La función de pérdida perceptual es lo que caracteriza al modelo SRGAN y lo diferencia de otros modelos para super-resolución que han surgido.

La función de pérdida perceptual contiene un factor que pesa la pérdida de reconstrución (o contenido) y otro que pesa la pérdida antagónica:

$$l^{SR} = l_X^{SR} + 10^{-3} l_{Gen}^{SR}, \tag{4.1}$$

donde l_X^{SR} es la pérdida debido al contenido, l_{Gen}^{SR} corresponde a la pérdida antagónica y l^{SR} es la pérdida total que tiene en cuenta ambos factores.

- **Pérdida de contenido**

  La pérdida de contenido se define como la pérdida de error cuadrático medio debido a diferencias entre pixeles (*Pixel-wise Mean Squared Error loss*) y la pérdida debida a diferencias entre las representaciones de las características de alto nivel de la imagen, que se pueden extraer a través de redes convolucionales VGG[V] pre-entrenadas.

$$l_X^{SR} = l_{MSE}^{SR} + l_{VGG/i,j}^{SR}, \tag{4.2}$$

  donde la pérdida debida a la diferencia entre pixeles viene dada por

$$l_{MSE}^{SR} = \frac{1}{r^2 WH} \sum_{x=1}^{rW} \sum_{y=1}^{rH} (I_{x,y}^{HR} - G_{\theta_G}(I^{LR})_{x,y})^2. \tag{4.3}$$

  Siendo r un factor de escala, H la altura de la imagen y W el ancho de la misma. $G_{\theta G}$ corresponde al Generador de la red e I^{HR}, I^{LR} se refieren a la imagen de alta y baja resolución, respectivamente. Es decir, $G_{\theta_G}(I^{LR})$ representa la imagen de alta resolución generada por el generador tomando como entrada la imágen de baja resolución.

  Por otro lado, la pérdida debido a la diferencia de las características de alto nivel puede ser escrita como:

$$l_{VGG/i,j}^{SR} = \frac{1}{W_{i,j} H_{i,j}} \sum_{x=1}^{w_{i,j}} \sum_{y=1}^{H_{i,j}} (\phi_{i,j}(I^{HR})_{x,y} - \phi_{i,j}(G_{\theta_G}(I^{LR}))_{x,y})^2. \tag{4.4}$$

  Donde ϕ_i, j indica el mapa de características obtenido por la $j-$ésima convolución (luego de la activación) antes de la capa $i-$ésima de *max-pooling*[VI] de la red VGG pre-entrenada con el conjunto de datos *ImageNet*[VII].
  $W_{i,j}$ y $H_{i,j}$ se refieren a las dimensiones de un mapa de características particular de la red VGG.
  La pérdida $l_{VGG/i,j}^{SR}$ representa la distancia Euclídea entre el mapa de característricas de la imagen que es generada y la imagen real.

[V] Las redes VGG constituyen una arquitectura de red neuronal convolucional y fue propuesta en el año 2014 por K. Simonyan y A. Zisserman del Instituto de Robótica de Oxford [36]

[VI] Las capas de *pooling* o de reducción se utilizan usualmente luego de capas convolucionales para disminuir la cantidad de parámetros, reduciendo las dimensiones de espaciales de la imagen de entrada, esto permite reducir un posible sobreajuste y evitar la sobrecarga de cálculo en la red. Las capas de *max-pooling* dividen la imagen de entrada en un conjunto de regiones conservando solamente el máximo valor de cada subregión.

[VII] ImageNet es una de las mayores bases de datos de imágenes disponibles. Para más información: http://www.image-net.org/ .

Para entrenar una red SRGAN pueden utilizarse los dos factores que contribuyen a la pérdida de contenido, tal y como muestra la ecuación 4.2, o bien sólo el factor $l^{SR}_{VGG/i,j}$, como se hace en el artículo original [1].

- **Pérdida antagónica**

  Las pérdida antagónica se calcula teniendo en cuenta las probabilidades que devuelve el discriminador.

$$l^{SR}_{Gen} = \sum_{n=1}^{N} -logD_{\theta_D}(G_{\theta_G}(l^{LR})). \tag{4.5}$$

  $G_{\theta_G}(l^{LR})$ representa la imagen producida por el generador y $D_{\theta_D}(G_{\theta_G}(l^{LR}))$ es la probabilidad de que la imagen generada sea real, según el discriminador.

De esta manera, al ir decreciendo la pérdida perceptual 4.1, el generador comienza a generar imágenes más realistas.

4.2.3 Métodos y Resultados

Una vez que se cuenta con la implementación en código de la arquitectura de la GAN de la sección 4.2.1 , resta implementar el procesamiento de las imágenes de entrada, la red VGG pre-entrenada y la sección que permite el entrenamiento de la GAN tomando en cuenta la pérdida perceptual de la sección 4.2.2.
El código completo puede verse en el repositorio de *GitHub* asociado a este trabajo [34] .

Datos y procesamiento inicial

El conjunto de datos utilizado para el entrenamiento consistió en imágenes de flores y fue descargado de [37].

Este conjunto de datos fue procesado mediente el siguiente *script* con dos fines:

1. Descartar imágenes de resolución menor a 384×384.
2. Normalizar todas las imágenes a una resolución final de 384×384.

```
from PIL import Image
import os, sys

path = "dataset/"
dirs = os.listdir(path)

def resize():
    for item in dirs:
        if os.path.isfile(path+item):
            im = Image.open(path+item)

            width, height = im.size
            if width >= 384 and height >= 384:
                f, e = os.path.splitext(path+item)
                imResize = im.resize((384,384),
                ↪  Image.ANTIALIAS)
                imResize.save(item + ' resized.jpg', 'JPEG',
                ↪  quality=90)
    print("done")
resize()
```

Luego del procesamiento se obtuvieron aproximadamente 6500 imágenes que son utilizadas para el entrenamiento del modelo.

Entrenamiento en Google Colaboratory

El entrenamiento de una red neuronal profunda suele consumir recursos computacionales que no siempre se tienen a disposición. Las GANs no son la excepción y, en general, lo ideal es hacer el entrenamiento utilizando una GPU en lugar de la CPU.
Google Colaboratory es una herramienta desarrollada por Google, que permite ejecutar código Python en la nube, con la posibilidad de hacer uso de sus GPUs de manera gratuita.
Además del mencionado, otro beneficio de Google Colaboratory es que permite compartir el código entre distintos usuarios, si es preciso trabajar en equipo.

El entrenamiento de esta SRGAN se realizó utilizando Google Colaboratory y la GPU NVIDIA Tesla K80 y se hizo a lo largo de 3000 épocas[VIII].

En el apéndice A.3 se describe la configuración utilizada para entrenar el modelo en este entorno.

Resultados

A continuación se puede ver la función de pérdida del generador (figura 4.3) y del discriminador (figura 4.4) en función de la cantidad de épocas durante el entrenamiento.

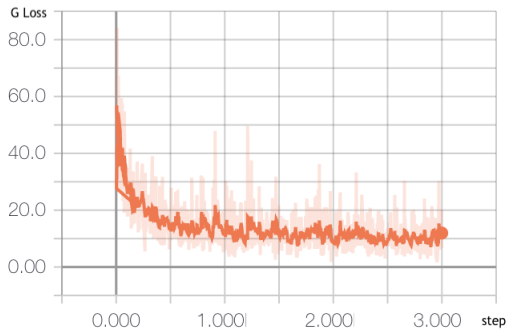

Figura 4.3: Pérdida del generador en función de la cantidad de épocas durante el entrenamiento de la SRGAN.

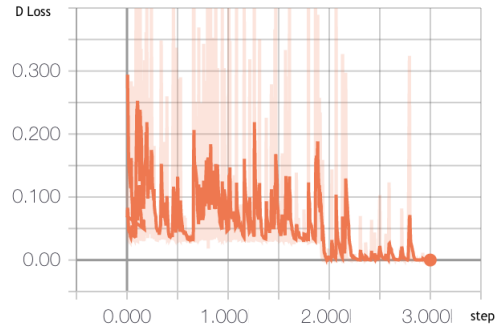

Figura 4.4: Pérdida del discriminador en función de la cantidad de épocas durante el entrenamiento de la SRGAN.

[VIII] Cabe aclarar que en este caso, una época no corresponde al pasaje a la red del conjunto total del *dataset*, sino que se refiere más exactamente a la cantidad de veces que un lote se introdujo en la red para su entrenamiento.

En las figuras 4.5 y 4.6 se pueden ver, respectivamente, las imágenes generadas por la SRGAN a las 500 y a las 3000 épocas durante el entrenamiento. A la izquierda se encuentra la imagen de baja resolución, que es la entrada del generador. La imagen original se muestra en el medio y a la derecha se observa la imagen generada a partir de la imagen de baja resolución por la SRGAN. Se puede apreciar cómo al aumentar la cantidad de épocas mejora la calidad de la imagen producida por la SRGAN.

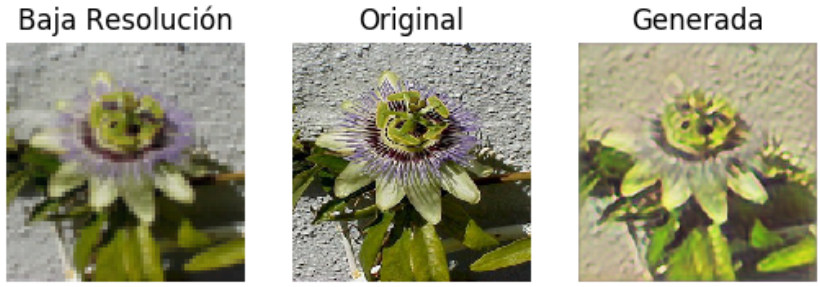

Figura 4.5: Imagen generada durante el entrenamiento de la SRGAN luego de 500 épocas.

Figura 4.6: Imagen generada durante el entrenamiento de la SRGAN luego de 3000 épocas.

La figura 4.7 muestra imágenes generadas por la SRGAN a partir del conjunto de prueba, una vez finalizado el entrenamiento. Puede apreciarse cualitativamente, que la calidad de las imagenes de testeo son muy similares a la calidad de la imagen producida durante la última época del entrenamiento, sugiriendo que no existe *overfitting*.

Figura 4.7: Imágenes generadas durante el testeo de la SRGAN, una vez finalizado el entrenamiento.

Finalmente, la figura 4.8 muestra imágenes generadas por la SRGAN a partir imágenes que no corresponden al conjunto de testeo, sino que se trata de dos fotos que no tienen relación con el conjunto de datos con el que fue entrenada la red. Puede observarse que aún en este caso las imágenes producidas por la SRGAN son cualitativamente de alta calidad.

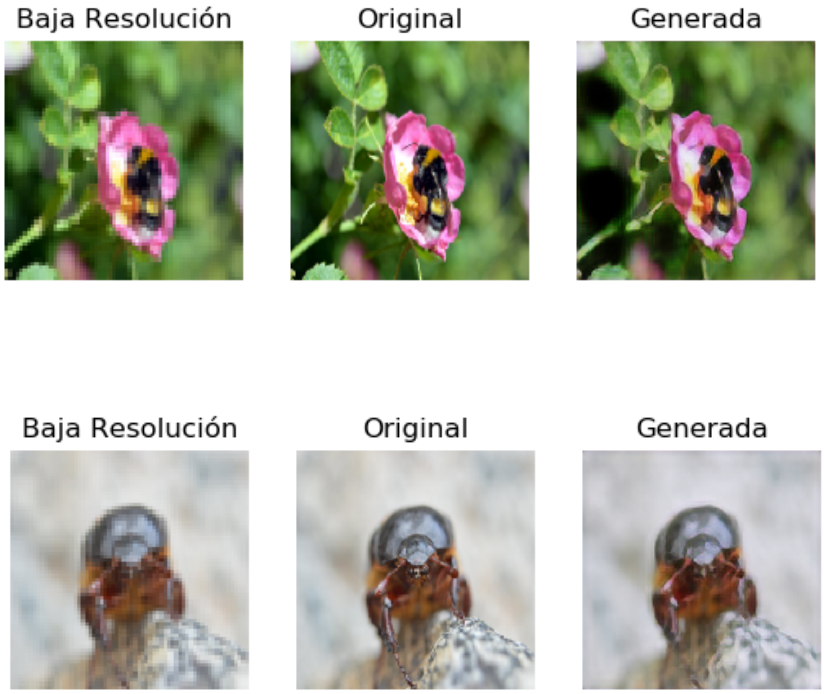

Figura 4.8: Imágenes generadas durante el testeo de la SRGAN, a partir de fotos tomadas por Luis A. Calcagni en San Carlos de Bariloche, Argentina.

5

Conclusiones

No cabe duda de la gran importancia en el área de Inteligencia Artificial que tienen y que tendrán las Redes Generativas Antagónicas. Las GANs constituyen un tipo de modelo generativo muy poderoso y el más investigado en la actualidad.
Desde que surgieron, las GANs han sido ampliamente estudiadas debido a su enorme potencial de aplicaciones. Cada semana se publica una considerable cantidad de artículos científicos relacionados con este estema, en los cuales se implementan arquitecturas ya publicadas o se proponen nuevas para un determinado fin. Como tal, es un campo de investigación en auge.

Dentro del gran abanico de aplicaciones que permiten las GANs, la super-resolución (SR) de imágenes es un problema interesante y muy relevante en el campo de visión por computadora. La generación de imágenes de alta resolución a partir de su contraparte de baja resolución tiene aplicaciones en el campo científico, de diagnóstico por imágenes y en sistemas de seguridad, entre muchos otros.

La implementación de la SRGAN propuesta en [1] mediante las librerías *Keras* y *TensorFlow* generó imágenes cualitativamente de alta calidad. Esto se pudo apreciar, tanto utilizando como entrada imágenes del conjunto de prueba, como imágenes sin relación con el conjunto de datos utilizado para entrenar la red.
Como es de esperar, el entrenamiento de la red muestra un comportamiento oscilante, pero estabilizado a largo plazo. El hecho de que la calidad de las imágenes que son producidas utilizando el conjunto de datos de prueba sea cualitivamente comparable con la imagen producida durante la última época en el entrenamiento, sugiere que no ha ocurrido *overfitting*.

Como proyecto futuro, se recomienda añadir en este análisis métricas cuantitativas que permitan determinar conclusiones más rigurosas.

Bibliografía

[1] Christian Ledig, Lucas Theis, Ferenc Huszár, José Antonio Caballero, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, and Wenzhe Shi. Photo-realistic single image super-resolution using a generative adversarial network. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 105–114, 2016.

[2] M. Mirza B. Xu D. Warde-Farley S. Ozair A. Courville I. J. Goodfellow, J. Pouget-Abadie and Y. Bengio. Generative adversarial nets. *In Advances in Neural Information Processing Systems (NIPS)*, abs/1406.2661, 2014.

[3] Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. *ICLR*, abs/1312.6114, 2013.

[4] D. E. Rumelhart and J. L. McClelland. *Learning Internal Representations by Error Propagation*, pages 318–362. 1987.

[5] J. von Neumann. Zur theorie der gesellschaftsspiele. *Mathematische Annalen*, 100:295–320, 1928.

[6] A. Creswell, T. White, V. Dumoulin, K. Arulkumaran, B. Sengupta, and A. A. Bharath. Generative adversarial networks: an overview. *IEEE Signal Process. Mag., 35(1):53–65*, 2018.

[7] M. Arjovsky, S. Chintala, and L. Bottou. Wasserstein gan. *eprint arXiv:1701.07875*, 2017.

[8] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A.C. Courville. Improved training of wasserstein gans. In *NIPS*, 2017.

[9] K. Wang, C. Gou, Y. Duan, Y. Lin, and X. Zheng. Generative adversarial networks: Introduction and outlook. *IEEE/CAA Journal of Automatica Sinica*, 2017.

[10] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. 2015.

[11] Mehdi Mirza and Simon Osindero. Conditional generative adversarial nets. *CoRR*, abs/1411.1784, 2014.

[12] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A. Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. *CoRR*, abs/1703.10593, 2017.

[13] A. Sage, E. Agustsson, R. Timofte, and L. Van Gool. Logo synthesis and manipulation with clustered generative adversarial networks. *CoRR*, abs/1712.04407, 2017.

[14] A. Antoniou, A.Storkey, and H. Edwards. Data augmentation generative adversarial networks. *ArXiv*, abs/1711.04340, 2017.

[15] Y. Jin, J. Zhang, M. Li, Y. Tian, H. Zhu, and Z. Fang. Towards the automatic anime characters creation with generative adversarial networks. *ArXiv*, abs/1708.05509, 2017.

[16] H.Hukkelås, R. Mester, and F. Lindseth. Deepprivacy: A generative adversarial network for face anonymization. In *ISVC*, 2019.

[17] de Oliveira L., Paganini M., and Nachman B. Learning particle physics by example: Location-aware generative adversarial networks for physics synthesis. *Computing and Software for Big Science*, 1, 01 2017.

[18] V. Chandak, P. Saxena, M. Pattanaik, and G. Kaushal. Semantic image completion and enhancement using deep learning. *2019 10th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, pages 1–6, 2019.

[19] Y. Li, S. Liu, J. Yang, and M. Yang. Generative face completion. *CoRR*, abs/1704.05838, 2017.

[20] H. Zhang, T. Xu, H. Li, S. Zhang, X. Huang, X. Wang, and D. N. Metaxas. Stackgan: Text to photo-realistic image synthesis with stacked generative adversarial networks. *CoRR*, abs/1612.03242, 2016.

[21] S. Reed, Z. Akata, X. Yan, L. Logeswaran, B. Schiele, and H. Lee. Generative adversarial text to image synthesis. *CoRR*, abs/1605.05396, 2016.

[22] G. Antipov, M. Baccouche, and J. Dugelay. Face aging with conditional generative adversarial networks. *2017 IEEE International Conference on Image Processing (ICIP)*, pages 2089–2093, 2017.

[23] R. Sood, B. Topiwala, K. Choutagunta, R. Sood, and M. Rusu. An application of generative adversarial networks for super resolution medical imaging. *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 326–331, 2018.

[24] I. Sánchez and V. Vilaplana. Brain mri super-resolution using 3d generative adversarial networks. *ArXiv*, abs/1812.11440, 2018.

[25] Y. Chen, F. Shi, A. Christodoulou, Z. Zhou, Y. Xie, and D. Li. Efficient and accurate mri super-resolution using a generative adversarial network and 3d multi-level densely connected network. In *MICCAI*, 2018.

[26] Dwarikanath Mahapatra, Behzad Bozorgtabar, and Rahil Garnavi. Image super-resolution using progressive generative adversarial networks for medical image analysis. *Computerized Medical Imaging and Graphics*, 71:30 – 39, 2019.

[27] Tong Zheng, Hirohisa Oda, Takayasu Moriya, Shota Nakamura, Masahiro Oda, Masaki Mori, Horitsugu Takabatake, Hiroshi Natori, and Kensaku Mori. Multi-modality super-resolution loss for gan-based super-resolution of clinical ct images using micro ct image database. 2019.

[28] S. Bhattacharyya, V. Snášel, D. Gupta, and A. Khanna. *Hybrid Computational Intelligence: Challenges and Applications*. Academic Press, 2020, 2020.

[29] A. Clark, J. Donahue, and K. Simonyan. Efficient video generation on complex datasets. *CoRR*, abs/1907.06571, 2019.

[30] S. Aigner and M. Körner. Futuregan: Anticipating the future frames of video sequences using spatio-temporal 3d convolutions in progressively growing gans. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, XLII-2/W16:3–11, 09 2019.

[31] Y. Kwon and M. Park. Predicting future frames using retrospective cycle gan. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1811–1820, 2019.

[32] William Lotter, Gabriel Kreiman, and David Cox. Unsupervised learning of visual structure using predictive generative networks, 2015.

[33] B. Yang, S. Rosa, A. Markham, N. Trigoni, and H. Wen. Dense 3d object reconstruction from a single depth view. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2019.

[34] L. Calcagni. Super resolution generative adversarial network. `https://github.com/lcalcagni/Super-resolution-Generative-Adversarial-Network`, 2020.

[35] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[36] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[37] Dataset:. `https://www.robots.ox.ac.uk/~vgg/data/flowers`.

[38] Gavin Hackeling. *Mastering Machine Learning with scikit-learn*. Packt Publishing, 2014.

[39] Geoffrey I. Webb Claude Sammut. *Encyclopedia of Machine Learning*. Springer, 1st edition. edition, 2011.

[40] Sujit Pal Antonio Gulli, Amita Kapoor. *Deep Learning with TensorFlow 2.0 and Keras*. Packt, 2nd edition. edition, 2019.

[41] Tony Jebara. *Machine Learning: Discriminative and Generative*. The Kluwer International Series in Engineering and Computer Science. Kluwer Academic Publishers / Springer, 2004.

[42] Carl Doersch. Tutorial on variational autoencoders. *ArXiv*, abs/1606.05908, 2016.

[43] Nao Takano and Gita Alaghband. Generator from edges: Reconstruction of facial images. 2020.

[44] M. Gruber. Srgan-keras. `https://github.com/MathiasGruber/SRGAN-Keras`, 2019.

[45] A. Sham. Srgan keras implementation. `https://github.com/AvivSham/SRGAN-Keras-Implementation`, 2019.

[46] D. Birla. Keras-srgan. `https://github.com/deepak112/Keras-SRGAN`, 2018.

[47] K. Ahirwar. Generative-adversarial-networks-projects. `https://github.com/PacktPublishing/Generative-Adversarial-Networks-Projects`, 2018.

[48] A. Ruano. Pytorch-srgan. `https://github.com/aitorzip/PyTorch-SRGAN`, 2017.

[49] Y. Yang. Super resolution with cnns and gans. `https://github.com/yiyang7/Super_Resolution_with_CNNs_and_GANs`, 2017.

[50] D2L:. https://d2l.ai/chapter_convolutional-neural-networks/lenet.html.

[51] Yann Lecun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324, 1998.

A

Apéndice

En este apéndice se adjunta la implementación en Keras y TensorFlow de la arquitectura del generador y discriminador de la SRGAN propuesta en [1] y la configuración necesaria para poder entrenar el modelo en el entorno Google Colaboratory.

A.1 Generador

Implementación en *Keras* y *TensorFlow* de la arquitectura del generador de la SRGAN propuesta en [1].
Este código también se encuentra disponible en el repositorio de GitHub asociado a este proyecto [34].

```
from keras import Input
from keras.models import Model
from keras.layers.convolutional import Conv2D, UpSampling2D
from keras.layers import BatchNormalization, Activation,
↪  LeakyReLU, PReLU, Add, Dense

#Build the generator network
def build_generator():

    #Residual Block
    def residual_block(input):

        res = Conv2D(64, kernel_size=3, strides=1,
        ↪  padding='same')(input)
        res = BatchNormalization(momentum=0.8)(res)
        res = PReLU(shared_axes=[1,2])(res)
        res = Conv2D(64, kernel_size=3, strides=1,
        ↪  padding='same')(res)
        res = BatchNormalization(momentum=0.8)(res)
        res = Add()([res, input])

        return res

    #------
    #Parameters
    input_shape = (64, 64, 3)      #Low Resolution image
    ↪  dimension
    res_blocks = 16                #Number of Residual Blocks
```

```
    #Input Layer for the Low Resolution image
    input_lr = Input(shape=input_shape)

    #Pre-Residual Block
    pre_res = Conv2D(filters=64, kernel_size=9, strides=1,
    ↪  padding='same')(input_lr)
    pre_res = PReLU(shared_axes=[1,2])(pre_res)

    #Residual Blocks
    res = residual_block(pre_res)
    for i in range(res_blocks-1):
        res = residual_block(res)

    #Post-Residual Block
    post_res = Conv2D(64, kernel_size=3, strides=1,
    ↪  padding='same')(res)
    post_res = BatchNormalization(momentum=0.8)(post_res)
    post_res = Add()([post_res, pre_res])

    #Upsampling Blocks
    upsamp_1 = UpSampling2D(size=2)(post_res)
    upsamp_1 = Conv2D(filters=256, kernel_size=3, strides=1,
    ↪  padding='same')(upsamp_1)
    upsamp_1 = PReLU(shared_axes=[1,2])(upsamp_1)

    upsamp_2 = UpSampling2D(size=2)(upsamp_1)
    upsamp_2 = Conv2D(filters=256, kernel_size=3, strides=1,
    ↪  padding='same')(upsamp_2)
    upsamp_2 = PReLU(shared_axes=[1,2])(upsamp_2)

    #Output Layer for the High Resolution image
    output_hr = Conv2D(filters=3, kernel_size=9, strides=1,
    ↪  padding='same', activation='tanh')(upsamp_2)

    #------
    #Create the model
    model = Model(inputs=[input_lr], outputs=[output_hr],
    ↪  name='generator')
    return model
```

A.2 Discriminador

Implementación en *Keras* y *TensorFlow* de la arquitectura del discriminador de la SRGAN propuesta en [1].
Este código también se encuentra disponible en el repositorio de GitHub asociado a este proyecto [34].

```
from keras import Input
from keras.models import Model
from keras.layers.convolutional import Conv2D, UpSampling2D
from keras.layers import BatchNormalization, Activation,
↪  LeakyReLU, PReLU, Add, Dense

#Build the discriminator network
def build_discriminator():

    #Convolution Block
    def conv_block(input, filter, strides=1, batchnorm=True):
        conv = Conv2D(filters=filter, kernel_size=3,
        ↪  strides=strides, padding='same')(input)
        conv = LeakyReLU(alpha=0.2)(conv)

        if batchnorm:
            conv = BatchNormalization(momentum=0.8)(conv)
        return conv

    #------
    #Parameters
    input_shape = (256, 256, 3)      #High Resolution image
    ↪  dimension
    filter=64                        #Initial filter

    #Input Layer for the High Resolution image
    input_hr = Input(shape=input_shape)

    #Convolution blocks
```

```
    conv = conv_block(input_hr, filter, batchnorm=False)
    conv = conv_block(conv, filter, strides=2)
    conv = conv_block(conv, filter*2)
    conv = conv_block(conv, filter*2, strides=2)
    conv = conv_block(conv, filter*4)
    conv = conv_block(conv, filter*4, strides=2)
    conv = conv_block(conv, filter*8)
    conv = conv_block(conv, filter*8, strides=2)

    #Dense layer
    dense_layer = Dense(filter*16)(conv)
    dense_layer = LeakyReLU(alpha=0.2)(conv)

    #Output classification layer
    output_class = Dense(1, activation='sigmoid')(dense_layer)

    #------
    #Create the model
    model = Model(inputs=input_hr, outputs=output_class,
    ↪  name='discriminator')
    return model
```

A.3 Entrenamiento en Google Colaboratory

Una vez clonado el repositorio [34] y subido a la cuenta de Google Drive dentro de un directorio denominado *Colab Notebooks*, se suben también las imágenes que serán necesarias para el entrenamiento en el directorio *input/train_data*.

Dentro de una sesión de Google Colaboratory, en principio se chequea la disponibilidad de la GPU:

```
from tensorflow.python.client import device_lib
import tensorflow as tf
print(device_lib.list_local_devices())
tf.test.is_gpu_available()
```

Luego, se acceden a los datos de Google Drive y al directorio donde se encuentra el proyecto:

```
from google.colab import drive
drive.mount('/content/gdrive')

%cd gdrive/My Drive/Colab
↪  Notebooks/Super-resolution-Generative-Adversarial-Network
```

A continuación, se instalan los paquetes necesarios para ejecutar el código y para utilizar la GPU con la versión de *Tensorflow* correspondiente:

```
!pip install -r requirements.txt

!wget http://developer.download.nvidia.com/compute/cuda/
repos/ubuntu1604/x86_64/cuda-repo-ubuntu1604_9.0.176-1_amd64.deb
!wget
↪  http://developer.download.nvidia.com/compute/machine-learning/
repos/ubuntu1604/x86_64/libcudnn7_7.0.5.15-1+cuda9.0_amd64.deb
```

```
!wget
↪ http://developer.download.nvidia.com/compute/machine-learning/
repos/ubuntu1604/x86_64/libcudnn7-dev_7.0.5.15-1+cuda9.0_amd64.deb
!wget
↪ http://developer.download.nvidia.com/compute/machine-learning/
repos/ubuntu1604/x86_64/libnccl2_2.1.4-1+cuda9.0_amd64.deb
!wget
↪ http://developer.download.nvidia.com/compute/machine-learning/
repos/ubuntu1604/x86_64/libnccl-dev_2.1.4-1+cuda9.0_amd64.deb
!wget
↪ http://developer.download.nvidia.com/compute/machine-learning/
repos/ubuntu1604/x86_64/libcudnn7_7.1.4.18-1+cuda9.0_amd64.deb

!sudo dpkg -i cuda-repo-ubuntu1604_9.0.176-1_amd64.deb
!sudo dpkg -i libcudnn7_7.0.5.15-1+cuda9.0_amd64.deb
!sudo dpkg -i libcudnn7-dev_7.0.5.15-1+cuda9.0_amd64.deb
!sudo dpkg -i libnccl2_2.1.4-1+cuda9.0_amd64.deb
!sudo dpkg -i libnccl-dev_2.1.4-1+cuda9.0_amd64.deb
!sudo apt-get update
!sudo apt-get install cuda=9.0.176-1
!sudo apt-get install libcudnn7-dev
!sudo dpkg -i libcudnn7_7.1.4.18-1+cuda9.0_amd64.deb
!sudo apt-get install libnccl-dev
```

Finalmente, se procede al entrenamiento del modelo mediante el comando:

```
!python main.py
```