Universidad Nacional de Rosario

Tesina de Grado

para la obtención del grado de Licenciado en Ciencias de la Computación

---

CODIFICACIÓN Y RECONSTRUCCIÓN DE ROSTROS CON REDES ADVERSARIAS GENERATIVAS

Autor:
Álvaro Zanetti

Director:
Dr. Lucas Uzal

Departamento de Ciencias de la Computación
Facultad de Ciencias Exactas, Ingeniería y Arquitectura
Av. Pellegrini 250, Rosario, Santa Fe, Argentina

Motivación

Las Redes Adversarias Generativas (GAN, por sus siglas en inglés) [Goo+14] son modelos basados en redes neuronales que han demostrado ser muy efectivos para generar imágenes sintéticas. Estos modelos se enmarcan en el área de Aprendizaje Automático, es decir que el modelo generativo aprende de los datos disponibles la distribución de probabilidad de los mismos. Para el caso concreto de colecciones de imágenes, esto significa que un modelo entrenado sobre una colección de imágenes de rostros generará una cierta diversidad de imágenes de rostros, pero el mismo modelo entrenado con una colección de imágenes de pájaros generará imágenes de pájaros. En otras palabras, los modelos de redes neuronales no están diseñados ad hoc para un tipo de imágenes en particular y en principio pueden ser aplicados a cualquier colección de imágenes (e incluso otros tipos de datos).

La particularidad de GAN respecto a otros modelos generativos basados en redes neuronales, es que el generador (que es simplemente una red neuronal multicapa), una vez entrenado, transforma determinísticamente un vector Z de ruido, que se inyecta en la entrada de la red, en una imagen en la salida de la red. Es decir que el proceso de generación no implica ningún proceso iterativo de optimización para cada imagen sino que basta una sola propagación a lo largo de la red neuronal. Toda la variabilidad en la salida de la red es capturada por el vector Z. No hay procesos estocásticos en la transformación.

En este contexto, resulta de especial interés estudiar el espacio latente aprendido por el generador: la relación semántica entre imágenes correspondientes a valores de Z cercanos, las transformaciones causadas por desplazamientos en una dada dirección en el espacio de entrada, etc. Un primer antecedente de este tipo de análisis es el realizado por Radford et al. [RMC15] quienes muestran que operaciones aritméticas básicas de suma y resta sobre los vectores Z tienen un correlato semántico en las imágenes generadas.

Se ha observado que en general es difícil obtener una función inversa del generador, es decir, una función tal que dada una imagen se obtenga el vector Z que le permita al generador reconstruir tal imagen. Esta es una tarea especialmente difícil cuando el dataset considerado presenta muy diversas categorías como por ejemplo CIFAR-10 o IMAGENET (10 y 1000 categorías respectivamente). Los resultados mejoran cuando se consideran conjuntos de imágenes más restringidos, como por ejemplo datasets de imágenes de rostros (que en particular presentan todos los rostros centrados y de frente). Sobre estos datasets se obtienen mejores resultados de generación sintética y también de inversión

del generador, sin embargo para esta última tarea los resultados no son completamente satisfactorios.

La motivación del presente trabajo es estudiar qué sucede al considerar el caso aún más restringido, utilizando un dataset de rostros conformados por una única identidad. Además, comparar los resultados con otros datasets de rostros de complejidad creciente y evaluar cómo impacta el proceso de inversión y regeneración en la preservación de la identidad.

Resumen

Se ha observado que en una arquitectura GAN en general es difícil obtener una función inversa del generador (G), es decir, una función tal que dada una imagen se obtenga el vector Z que le permita al generador reconstruir tal imagen.

El objetivo de esta tesina es estudiar este procedimiento de inversión considerando un dataset de rostros conformado por una única identidad, entendiendo que esta restricción simplificaría dicho proceso. Además, se desea comparar los resultados obtenidos con otros datasets de identidades diversas y mayor complejidad. Se desea investigar si esta simplificación del dataset permite lograr mejores reconstrucciones con el procedimiento anteriormente descripto.

Por otro lado, e independientemente de lo anterior, el entrenamiento adversario de un generador que permita obtener imágenes de una determinada identidad abre la posibilidad de diversas aplicaciones como la generación y animación de rostros sintéticos de una determinada persona, representación visual de chatbots, y en términos de seguridad informática, suplantación de identidad en sistemas de validación por reconocimiento facial en video. Por lo tanto, un objetivo secundario, es evaluar el potencial de estas técnicas para las mencionadas aplicaciones.

Agradecimientos

Para Flavia, quien me acompaña hace 10 años, y desde un principio fue un sostén imprescindible para realizar este viaje. Sin ella no hubiera podido llegar a la meta.

Mi familia, de sangre y política, por el apoyo incondicional y ese esfuerzo incalculable para que nunca me faltara nada de lo que necesitaba.

Mis amigos de la carrera y de la vida, en especial Dipra, Feli, Juli y Mauro. Cada uno merece un pedacito de este logro.

Para Lucas, mi director, quien dedicó largas horas de trabajo y me ofreció su invaluable conocimiento.

A la familia de ProfitWell, tanto Rosario como Boston, por darme mi primer oportunidad laboral, el apoyo constante y toda la flexibilidad que me ofrecieron durante este último año.

Toda aquella persona que forma parte, de una forma u otra, de esta gran comunidad de LCC.

A todos los mencionados y los que seguramente falte mencionar, les agradezco de corazón por todo lo que recibí.

Índice general

Índice de figuras

Lista de Abreviaturas

IA	**I**nteligencia **A**rtificial
AA	**A**prendizaje **A**utomático
AP	**A**prendizaje **P**rofundo
RNC	**R**edes **N**euronales **C**onvolucionales
GAN	**G**enerative **A**dversarial **N**etwork
DCGAN	**D**eep **C**onvolutional **G**enerative **A**dversarial **N**etwork
ResNet	**Res**idual **Net**work
VAE	**V**ariational **A**uto**e**ncoder
WGAN	**W**asserstein **G**enerative **A**dversarial **N**etwork
WGAN-GP	**W**asserstein **G**enerative **A**dversarial **N**etwork - **G**radient **P**enalty

Capítulo 1

Marco Teórico

1.1. Inteligencia Artificial, Aprendizaje Automático y Aprendizaje Profundo

Como puntapié inicial, vamos a explicar a qué nos referimos cuando decimos Inteligencia Artificial. ¿Qué es la Inteligencia Artificial (IA), el Aprendizaje Automático (AA) y el Aprendizaje Profundo (AP)? ¿Cómo se relacionan entre ellos? Un pequeño adelanto se observa en la Figura 1.1.

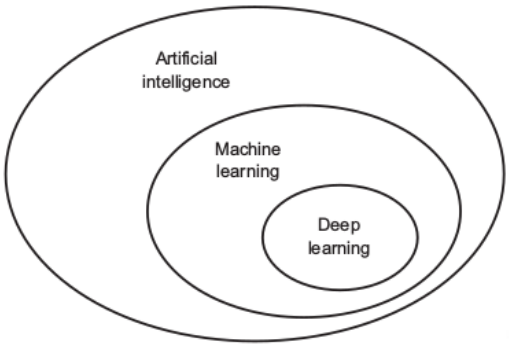


FIGURA 1.1: Inteligencia Artificial, Aprendizaje Automático y Aprendizaje Profundo [Cho17].

1.1.1. Inteligencia Artificial (IA)

El término Inteligencia Artificial fue acuñado en 1956 cuando Allen Newell, Herbert Simon, John McCarthy, Marvin Minsky y Arthur Samuel se convirtieron en los fundadores y líderes de la investigación en el campo de IA.

Una de las tantas definiciones para IA es la siguiente: *una rama de las Ciencias de la Computación que tiene como objetivo simular comportamiento inteligente en una computadora.* Por lo tanto, la Inteligencia Artificial es un campo más general que incluye al Aprendizaje Automático y al Aprendizaje Profundo,

pero que también incluye otros enfoques que no implican ningún aprendizaje. Por ejemplo, los primeros programas de ajedrez estaban compuestos por una gran cantidad de reglas escritas por programadores, lo cual no califica como AA.

Entre las décadas del '50 y '80 se creyó que con un conjunto de reglas suficientemente grande se podría lograr una inteligencia al nivel de los humanos. Esta creencia se manifiesta en el auge que tuvieron los sistemas expertos en la década del '80.

Estos sistemas son muy efectivos en problemas bien definidos y lógicos pero resulta intratable definir las reglas para solucionar problemas más complejos como visión por computadora, reconocimiento automático de voz o traducción automática. El Aprendizaje Automático aparece en escena para intentar resolver estos problemas.

1.1.2. Aprendizaje Automático (AA)

El Aprendizaje Automático -o Machine Learning, en inglés- surge de las preguntas:

"¿Puede una computadora ir más allá de lo que sabemos ordenarle y aprender por sí misma cómo realizar una determinada tarea?"

"En vez de tener que crear a mano un conjunto de reglas de procesamiento para los datos, ¿puede una computadora aprender automáticamente estas reglas analizando un repositorio de datos?"

De la mano de estas preguntas surge un nuevo paradigma de programación (Figura 1.2). En la programación clásica, los humanos escriben un conjunto de reglas (un programa) y proveen datos para ser procesados mediante las mismas, obteniendo las respuestas de este procesamiento. En el Aprendizaje Automático, en cambio, se proveen los datos y las respuestas esperadas de estos datos a una serie de algoritmos particulares de AA, y se obtienen reglas que luego pueden ser aplicadas a nuevos ejemplos para obtener respuestas desconocidas.

Se dice que un sistema de AA es *entrenado*, en vez de programado. Se le presentan una gran cantidad de ejemplos relevantes para una determinada tarea y encuentra una estructura estadística en los datos que le permiten al sistema generar reglas para automatizar dicha tarea.

Supongamos el siguiente problema muy simple, de clasificar una serie de puntos en un par de ejes de coordenadas, entre blancos y negros:

En dicho problema, se realiza una transformación, específicamente un cambio del sistema de coordenadas para lograr una representación más sencilla de los

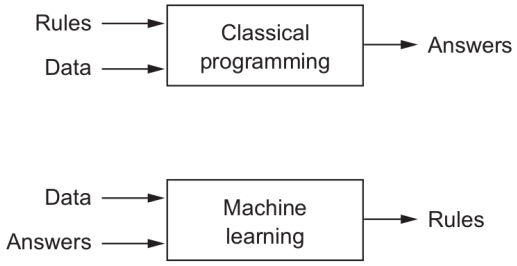


FIGURA 1.2: Aprendizaje Automático: un nuevo paradigma de programación [Cho17].

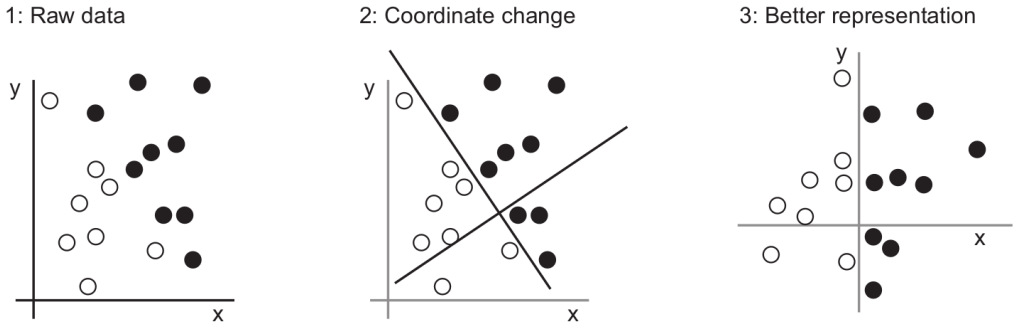


FIGURA 1.3: Clasificación de puntos. Cambio en el sistema de coordenadas [Cho17].

datos que permite clasificarlos con la simple regla: "Los puntos blancos tienen su componente $x < 0$ mientras que los negros $x > 0$".

En este caso el cambio de coordenadas se hizo manualmente, pero si dicha búsqueda es realizada por un algoritmo, de forma sistemática, probando los diferentes cambios de coordenadas y utilizando como retroalimentación el porcentaje de puntos correctamente clasificados, estaríamos haciendo Machine Learning.

Es decir, el *aprendizaje* en el contexto de AA se define como una búsqueda automática -dentro del espacio de hipótesis- de una transformación que convierta los datos en otra representación más útil para la tarea a resolver.

La definición más citada se puede encontrar en el libro de Tom M. Mitchell [Mit97, capítulo 1, p. 2] que reproducimos a continuación: "Un algoritmo aprende de la experiencia E, con respecto a una tarea T y medida de rendimiento P, si su rendimiento al realizar la tarea T (medido por P) aumenta con la experiencia E."

Entre los algoritmos más conocidos y utilizados en el campo del Aprendizaje Automático podemos mencionar: árboles de decisión, algoritmos genéticos, máquinas de vectores de soporte, algoritmos de agrupamiento -clustering, en inglés- y redes neuronales. Éstas últimas son las responsables de los grandes

avances de la última década y con las que trabajaremos en la presente tesina.

El AA surgió en los '90 y rápidamente se convirtió en el subcampo más popular y exitoso de la Inteligencia Artificial, una tendencia que se potencia día a día por la creciente disponibilidad de datos y el aumento del poder de cómputo.

El AA sienta sus bases en la matemática y la estadística pero debemos tener en cuenta que debido a la complejidad de los datos (datasets de millones de imágenes que consisten en decenas de miles de píxeles cada una) los análisis estadísticos clásicos resultan imprácticos. Por lo tanto, el Aprendizaje Automático -y más aún el Aprendizaje Profundo- son disciplinas prácticas donde las ideas se prueban empíricamente con mayor frecuencia que de manera teórica.

1.1.3. Aprendizaje Profundo (AP)

El Aprendizaje Profundo es un subcampo del AA. Es un nuevo enfoque en el cuál se pone énfasis en aprender sucesivas *capas* que generan representaciones cada vez más significativas y con mayor nivel de abstracción. Además, estas capas se aprenden de manera *conjunta*, cada una siendo modificada para cumplir con las necesidades de representación de la capa superior como así también las de la capa inferior.

Muchas veces se encuentran declaraciones como "el aprendizaje profundo modela el funcionamiento de nuestro cerebro". Esto no es cierto, y si bien las redes neuronales surgieron con cierta inspiración de la neurociencia, no existe ninguna evidencia de que el cerebro humano funcione de tal manera, y sólo genera confusión para quienes están comenzando en este campo. Debemos ver al Aprendizaje Profundo (o Deep Learning) como una familia de modelos y técnicas de AA que explotan el concepto de multiplicidad de capas de abstracción previamente mencionado para aprender representaciones a partir de datos.

A la cantidad de capas que componen un determinado modelo se le llama la *profundidad* del modelo. Actualmente, existen modelos de decenas (*GoogleNet*) e incluso centenas (*Microsoft ResNet*) de capas.

En la Figura 1.4 se muestra de manera visual como evolucionan estas representaciones al atravesar las diferentes capas de una red profunda entrenada para clasificar dígitos del dataset MNIST [LC10].

El AP revolucionó el campo del Aprendizaje Automático con resultados impresionantes en problemas de percepción, como la vista y la audición, que resultan naturales e intuitivos para los humanos pero fueron siempre un problema de gran complejidad para las computadoras.

En particular, podemos nombrar los siguientes avances:

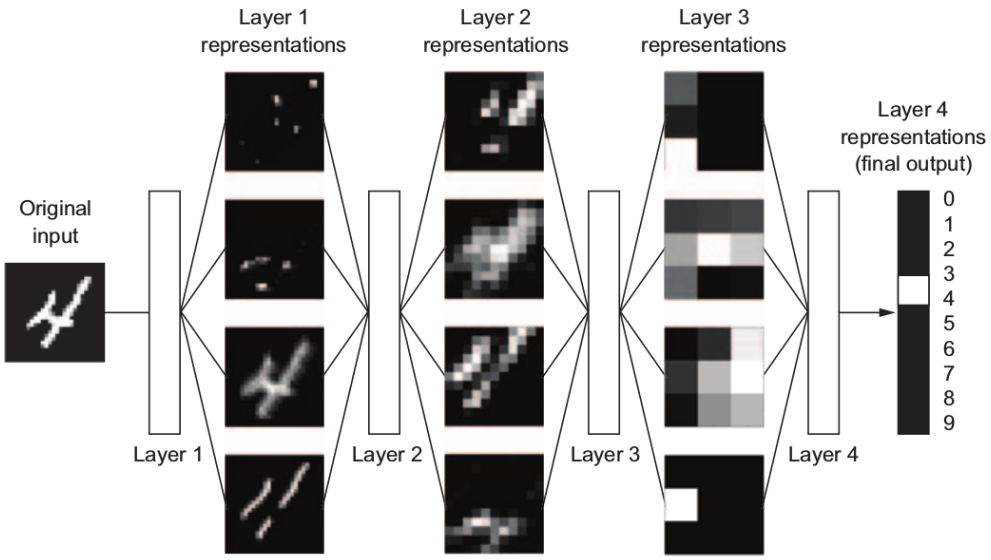


FIGURA 1.4: Representaciones en las diferentes capas de una red profunda aprendidas por un modelo de clasificación entrenado con datos del dataset MNIST [LC10] [Cho17].

- Clasificación de imágenes a nivel humano, o incluso superior en ciertos casos.
- Reconocimiento del habla a nivel casi humano.
- Transcripción de escritura a nivel casi humano.
- Conducción autónoma a nivel casi humano.
- Mejora de la traducción automática.
- Conversión de texto a voz.
- Asistentes digitales como Google Now y Amazon Alexa.
- Mejora en la focalización de anuncios, utilizada principalmente por Google, Baidu y Bing.
- Mejores resultados de búsqueda en la web.
- Capacidad para responder preguntas en lenguaje natural.
- Destreza sobrehumana para jugar Go.

Además se continúan expandiendo los usos de estas técnicas a diferentes campos. En caso de tener éxito, podríamos estar frente a un panorama en el

cuál el AP ayude a los seres humanos en campos como la ciencia, ingeniería, arte, desarrollo de software, y más.

El éxito del Aprendizaje Profundo no sólo se debe a la calidad de sus resultados. También facilita la resolución de problemas debido a que automatiza lo que siempre fue uno de los pasos más cruciales del Aprendizaje Automático: la ingeniería de características -del inglés, feature engineering-.

Las técnicas previas del AA , transformaban los datos de entrada en uno o dos espacios de representación sucesivos, generalmente a través de transformaciones simples tales como proyecciones no lineales de alta dimensión (SVM) o árboles de decisión. Sin embargo, las representaciones necesarias para problemas complejos de percepción como los que hablamos no pueden ser alcanzadas por dichas técnicas por lo que los ingenieros debían transformar los datos de entrada previamente de manera que puedan ser procesados por estos algoritmos. A esto le llamamos *ingeniería de características*. Este pre-procesamiento usualmente requiere de una nueva solución para cada problema que se desea resolver.

El AP por otro lado, automatiza este paso, descubriendo automáticamente las representaciones para los datos de entrada en vez de tener que diseñarlas manualmente, lo que facilita enormemente el flujo de trabajo. Es una característica distintiva de AP poder entrenar en general con los datos crudos sin requerir pre-procesamientos sofisticados.

1.1.4. Formas de aprendizaje automático

Aprendizaje Supervisado

En el Aprendizaje Supervisado tenemos disponibles un conjunto de pares $\langle entrada, salida \rangle$ para utilizar durante la fase de entrenamiento de nuestro algoritmo. Un ejemplo muy utilizado para explicar este tipo de problemas es el de reconocimiento de escritura. En este caso, nuestros pares $\langle entrada, salida \rangle$ se corresponden con imágenes de dígitos escritos a mano como *entrada* y la etiqueta correcta para dicho dígito como *salida*. Justamente se lo llama Aprendizaje Supervisado porque se puede pensar como un proceso donde un "maestro" supervisa el proceso de aprendizaje. El algoritmo produce una salida y se la compara con la respuesta correcta que ya conocemos, ajustando el algoritmo hasta obtener un desempeño aceptable.

Así, nuestro algoritmo puede descubrir los diferentes patrones que asocian una imagen con su etiqueta. O bien, en el caso general, aprender una función que asocie las entradas con su correspondientes salidas. El objetivo es que esta

función luego nos sirva para predecir las salidas sobre un nuevo conjunto de entradas nunca antes visto.

Sin embargo, existe una tarea previa crucial: sólo podemos utilizar algoritmos de aprendizaje supervisado si disponemos de un gran dataset previamente etiquetado de manera correcta, lo que significa que en algún punto un humano debe clasificar todas las imágenes disponibles en dicho dataset de entrenamiento. Este es un trabajo laborioso y en muchos casos inviable, pero cuando los datos existen, los algoritmos de aprendizaje supervisado pueden ser extremadamente efectivos en una amplia gama de tareas.

Los algoritmos de Aprendizaje Supervisado se pueden dividir en 2 subgrupos: regresión y clasificación.

En un problema de **regresión** se intenta estimar o predecir un valor continuo. ¿Cuál será el valor de una determinada acción de bolsa dentro de un mes? ¿Cuál es el precio de una casa en una determinada ciudad/barrio? ¿Cuántos de nuestros clientes se irán a un competidor este año? Estos son ejemplos de preguntas que caerían en el campo de la regresión. Para resolver estos problemas en un marco de aprendizaje supervisado, recopilaríamos ejemplos anteriores de pares de entrada/salida de "respuestas correctas" que tratan el mismo problema. Como entradas, identificaríamos las características que creemos que son predictivas de los resultados que deseamos predecir. Por ejemplo, para responder la primer pregunta podríamos recopilar los valores históricos de la acción en cuestión y generar ventanas de una determinada longitud como entrada, junto con el valor siguiente a dicha ventana como salida.

En un problema de **clasificación**, en cambio, se trata de asignar etiquetas discretas a las que llamamos *clases* en vez de predecir un valor continuo. El caso más simple es el de clasificación binaria: ¿Tiene un paciente determinado la enfermedad x? ¿Es un email spam o no? Por otro lado, el ejemplo de clasificación de dígitos antes mencionado es un problema donde debemos asignar una de entre k categorías. En muchos casos, estos problemas se pueden pensar como un conjunto de clasificadores de la forma: *detector-0, detector-1,..., detector-k* donde se devuelve la etiqueta cuyo clasificador tiene la mayor certeza.

Aprendizaje No Supervisado

En el caso del Aprendizaje No Supervisado disponemos únicamente de datos de entrada, sin ninguna información de la salida correcta. El objetivo es modelar la estructura subyacente o la distribución de los datos para aprender más sobre ellos.

Se le llama "no supervisado" pues, a diferencia del caso anterior, no existen "respuestas correctas" y por lo tanto, el algoritmo no es guiado sino que debe descubrir la estructura propia del dataset por sus propios medios.

Podemos identificar los siguientes problemas dentro de este grupo:

- **Agrupamiento — o clustering:** Por ejemplo, cuando se desea descubrir las agrupaciones inherentes en los datos, como la agrupación de clientes mediante el comportamiento de compra. Entre los algoritmos más conocidos para clustering podemos mencionar k-means y agrupamiento jerárquico.
- **Asociación:** Descubrir reglas que describan grandes porciones de los datos, como por ejemplo: "las personas que compran X también tienden a comprar Y". Un ejemplo es el Algoritmo apriori.
- **Modelos Generativos:** Todos los tipos de modelos generativos tienen como objetivo aprender –de manera explícita o implícita– la verdadera distribución de los datos del conjunto de entrenamiento para generar nuevos ejemplos con algunas variaciones. En este grupo se destacan algoritmos como Variational Autoencoders (VAE) y Generative Adversarial Networks (GAN).

Aprendizaje Semi-supervisado

Por último tenemos el Aprendizaje Semi-supervisado. En estos tipos de problemas generalmente disponemos de una gran cantidad de datos de entrada pero sólo disponemos de las salidas correctas para un pequeño subconjunto de los mismos.

Por ejemplo, una gran biblioteca de imágenes donde sólo algunas de ellas están correctamente etiquetadas.

Muchos problemas del mundo real se incluyen en esta categoría debido al hecho ya mencionado sobre el costo de etiquetar un dataset completo (a veces incluso siendo necesarios expertos del área), mientras que los datos sin etiquetas pueden ser recopilados y almacenados de forma sencilla y barata.

Por lo tanto, podemos usar aprendizaje no supervisado para descubrir la estructura subyacente en los datos de entrada, y luego utilizar los datos etiquetados para mejorar nuestro predictor. Por ejemplo, en [Lee13], utilizan un algoritmo no supervisado para hacer las mejores predicciones posibles sobre ejemplos no etiquetados, retro-alimentan esos resultados en el algoritmo de aprendizaje supervisado como datos de entrenamiento y luego utilizan el modelo para hacer predicciones sobre nuevos datos nunca vistos.

1.2. Redes Neuronales

Habiendo explicado las relaciones entre los diferentes subcampos de la Inteligencia Artificial, veamos con mayor profundidad de qué se tratan las Redes Neuronales.

Las subsecciones 1 a 4 (incluida) están basadas en el libro *Machine Learning de Tom M. Mitchel* [Mit97, capítulo 4]. En las mismas, se repasan conceptos clásicos de redes neuronales pero que son de relevancia para los métodos actuales.

Las redes neuronales artificiales están compuestas de un gran conjunto de unidades simples densamente interconectadas. Cada una de estas unidades toma un determinado número de entradas de valores reales (posiblemente la salida de unidades previas) y produce un único valor real de salida (que podría convertirse en entrada para otras unidades).

1.2.1. Perceptrones

Uno de los primeros tipos de redes neuronales estaban basados en unidades llamadas *perceptrones*. Estos toman un vector de valores reales como entrada, calculan una combinación lineal de dichos valores y producen un 1 como salida si el resultado es mayor a determinado umbral, o -1 en caso contrario. Más precisamente, dado un vector de entrada n-dimensional $x_1, ..., x_n$, la salida $o(x_1, ..., x_n)$ calculada por el perceptrón es:

$$o(x_1, ..., x_n) = \begin{cases} 1 & \text{si } w_0 + w_1 x_1 + w_2 x_2 + ... + w_n x_n > 0 \\ -1 & \text{caso contrario} \end{cases}$$

donde cada w_i es una constante real, o *peso*, que determina cuánto contribuye la entrada x_i a la salida producida por el perceptrón. Notar que el valor $(-w_0)$ es el umbral que la sumatoria $w_1 x_1 + w_2 x_2 + ... + w_n x_n$ debe superar para que la salida del perceptrón sea 1. Para simplificar la notación, muchos autores introducen una constante de entrada adicional $x_0 = 1$ que nos permite escribir la inecuación anterior de la forma $\sum_{i=0}^{n} w_i x_i > 0$, o en forma vectorial $\vec{w} \cdot \vec{x} > 0$, dando como resultado:

$$o(\vec{x}) = sgn(\vec{w} \cdot \vec{x})$$

donde

$$sgn(y) = \begin{cases} 1 & \text{si } y > 0 \\ -1 & \text{caso contrario} \end{cases}$$

Por lo tanto, el espacio de hipótesis considerado al entrenar un perceptrón es el conjunto de todos los posibles *vectores de pesos* de valores reales:

$$\{H = \vec{w} \,|\, \vec{w} \in \Re^{(n+1)}\}$$

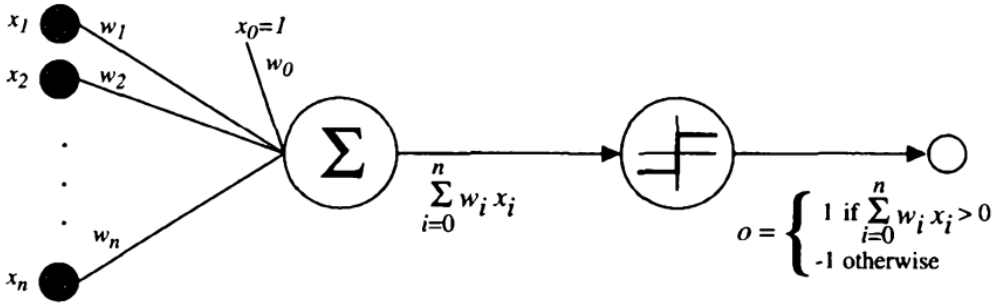


FIGURA 1.5: Representación gráfica de un perceptrón [Mit97].

Podemos entender al perceptrón como un hiperplano partiendo un espacio n-dimensional de manera que produce una salida igual a 1 para instancias que se encuentran a un lado del hiperplano, y -1 al otro. Por este motivo, la efectividad de un perceptrón de una única capa -del inglés, single layer perceptron- está limitada a conjuntos de datos *linealmente separables*.

Una manera de entrenar estas unidades es partir de pesos aleatorios, aplicar iterativamente el perceptrón a los datos de entrada, y modificar los pesos cuando éstos son clasificados erróneamente. Este proceso iterativo se repite hasta que el perceptrón clasifica todos los ejemplos de manera correcta. En cada paso, los pesos son modificados de la siguiente forma:

$$w_i \leftarrow w_i + \Delta w_i$$

donde

$$\Delta w_i = \eta(t - o)x_i$$

t es el valor deseado a la salida del perceptrón para la entrada x, o es el generado por el perceptrón, y η es una constante positiva llamada *learning rate*. El rol del *learning rate* es moderar la magnitud de cambio de los pesos en cada iteración. Se usan valores pequeños, generalmente entre $1e^{-3}$ y 0,1. Aunque existen algoritmos más complejos donde éste valor varía durante el entrenamiento. La optimización del *learning rate* se realiza de manera empírica.

Se puede probar que el procedimiento anterior converge en un número finito de pasos a un *vector de pesos* que clasifica correctamente todos los ejemplos, siempre y cuando dichos ejemplos sean linealmente separables y se utilice un η

suficientemente pequeño, ver [New69]. Si los datos no son linealmente separables, no se puede asegurar la convergencia.

1.2.2. Descenso por gradiente

Sin embargo, en el caso donde los ejemplos no son linealmente separables, la convergencia no está asegurada al utilizar la regla de actualización de pesos recién presentada. Para solucionar este problema se utiliza la llamada *delta rule*. Cuando los ejemplos no son linealmente separables, esta regla de actualización converge a la mejor aproximación posible del objetivo.

La idea fundamental en la que se basa es realizar un *descenso por gradiente* para buscar en el espacio de hipótesis de vectores de pesos los que mejor se adapten a los ejemplos de entrenamiento. Esta regla es muy importante porque sienta las bases para el algoritmo de *backpropagation*, el cual se utiliza para entrenar redes neuronales con múltiples capas intermedias.

Para explicar la *delta rule*, consideremos el entrenamiento de un perceptrón sin umbral, también llamado *unidad o neurona lineal* para la cual su salida o está dada por: $o(\vec{x}) = \vec{w} \cdot \vec{x}$. Además, especifiquemos una medida para el *error de entrenamiento* de una determinada hipótesis:

$$E(\vec{w}) = \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \tag{1.1}$$

donde D es el conjunto de ejemplos de entrenamiento, t_d es el valor objetivo y o_d es la salida de la *unidad lineal* para el ejemplo d. Notar que el error está parametrizado por el vector de pesos $\vec{w}$ ya que las salidas o_d dependen de dicho vector. Obviamente este error también depende de los ejemplos de entrenamiento, pero suponemos que los mismos no cambian por lo cual no es necesario expresar el error en término de los mismos.

Para entender el algoritmo de descenso por gradiente es de gran ayuda visualizar el espacio de hipótesis de los posibles vectores de pesos y su error E asociado, como se presentan en la Figura 1.6.

Así, la técnica de descenso por gradiente determina el vector de pesos que minimiza el error E, comenzando con un vector aleatorio de pesos, y actualizándolo iterativamente. En cada actualización el vector de pesos se mueve en la dirección de mayor descenso del error hasta encontrar el mínimo global.

Para encontrar esta dirección de mayor descenso del error calculamos la derivada del error E con respecto a cada una de las componentes del vector $\vec{w}$. Este vector de derivadas recibe el nombre de *gradiente* de E con respecto a $\vec{w}$

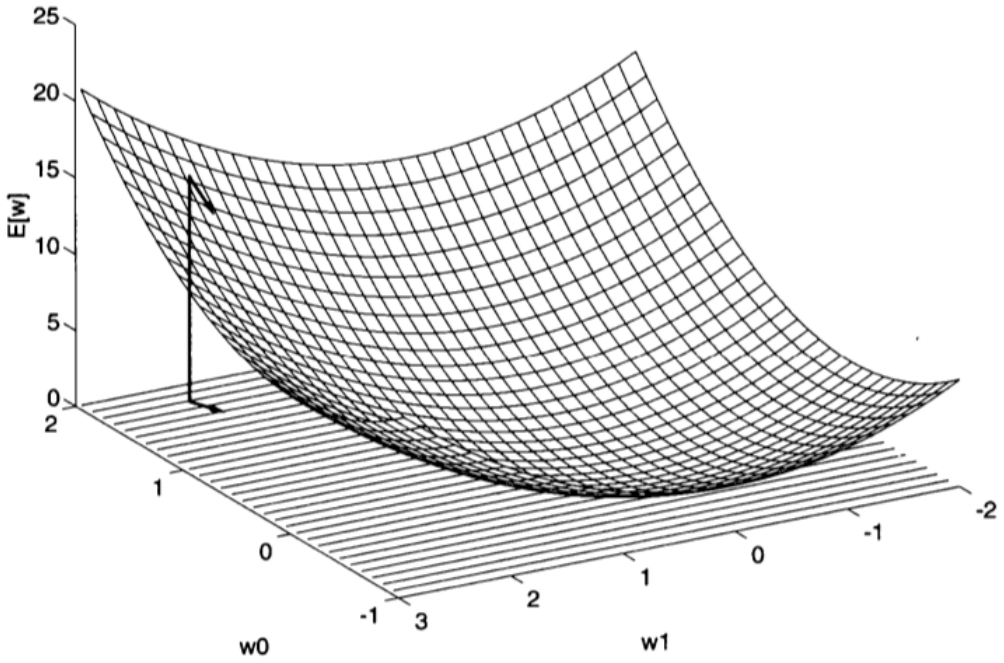


FIGURA 1.6: Error para las diferentes hipótesis, considerando una unidad lineal con 2 pesos w_0 y w_1. El espacio de hipótesis es el plano w_0, w_1 mientras que el eje vertical indica el error para dichos valores de pesos. La flecha marca la dirección de mayor descenso del error para el vector de pesos [Mit97].

y se simboliza $\nabla E(\vec{w})$.

$$\nabla E(\vec{w}) = \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, ..., \frac{\partial E}{\partial w_n}\right]$$

Si interpretamos $\nabla E(\vec{w})$ como un vector en el espacio de pesos, se corresponde con la dirección que produce el mayor ascenso en el error E, por lo que la negación del mismo nos da la dirección de mayor disminución del error. Por lo tanto, la regla de actualización de pesos está dada por:

$$\vec{w} \leftarrow \vec{w} + \Delta\vec{w}$$

donde

$$\Delta\vec{w} = -\eta\nabla E(\vec{w})$$

siendo η el *learning rate*.

Considerando cada uno de los componentes por separado, tenemos:

$$w_i \leftarrow w_i + \Delta w_i$$

donde

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i} \tag{1.2}$$

quedando claro que la dirección óptima de reducción del error se logra alterando cada componente w_i de $\vec{w}$ en proporción a $\frac{\partial E}{\partial w_i}$. Veamos el cálculo de este último valor:

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_{d \in D} \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_{d \in D} 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\
&= \sum_{d \in D} (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - \vec{w} \cdot \vec{x_d}) \\
\frac{\partial E}{\partial w_i} &= \sum_{d \in D} (t_d - o_d)(-x_{id})
\end{aligned} \tag{1.3}$$

donde x_{id} es la componente i del ejemplo de entrenamiento d. Substituyendo la ecuación 1.3 en la ecuación 1.2 obtenemos la regla para la actualización de pesos

$$\Delta w_i = \eta \sum_{d \in D} (t_d - o_d) x_{id} \tag{1.4}$$

Resumiendo, el algoritmo de descenso por gradiente para el entrenamiento de unidades lineales está compuesto por los siguientes pasos:

- Se selecciona un vector aleatorio de pesos inicial.
- Se aplica la unidad lineal a todos los ejemplos de entrenamiento.
- Se computa Δw_i para cada uno de las componentes del vector de pesos, de acuerdo a la ecuación 1.4.
- Se actualiza cada uno de los pesos w_i, sumándoles Δw_i.
- Se repite este procedimiento hasta la convergencia (o hasta la condición de terminación).

1.2.3. Descenso por gradiente estocástico

Las principales dificultades prácticas al aplicar descenso por gradiente son:

1. La convergencia a un mínimo local puede ser lenta en ciertas ocasiones.

2. Si hay múltiples mínimos locales en la superficie del error, no existen garantías de que se encontrará un mínimo global.

Una modificación muy utilizada del algoritmo de descenso por gradiente es el llamado *descenso por gradiente estocástico* o *SGD*, del inglés. Mientras que la regla de actualización dada en la ecuación 1.4 computa Δw_i considerando *todos* los ejemplos en el conjunto D, la idea central detrás de SGD es realizar la actualización de pesos considerando únicamente un ejemplo seleccionado al azar en el conjunto D, de aquí el nombre *estocástico*. Así, si se utiliza un valor suficientemente pequeño para η, *SGD* puede aproximar el algoritmo de *descenso por gradiente* de manera eficiente.

Por lo tanto, las diferencias fundamentales entre estos dos algoritmos son:

- En el algoritmo estándar, el error se calcula mediante una sumatoria sobre todos los ejemplos *antes* de realizar la actualización de pesos. Mientras que en *SGD* los pesos se actualizan luego de analizar cada ejemplo de entrenamiento.

- Esta sumatoria sobre todos los ejemplos tiene un costo computacional muchísimo mayor para cada actualización de pesos. Por otro lado, debido a que utiliza el verdadero gradiente, el algoritmo clásico puede ser utilizado con un η mayor.

- Ante la existencia de múltiples mínimos locales en la superficie del error, *SGD* tiene menor probabilidad de quedar atrapado en los mismos ya que actualiza los pesos utilizando el gradiente con respecto a un ejemplo particular en cada paso.

Una tercera variación de este algoritmo, que resulta ser la más utilizada en la práctica, es el *mini-batch SGD*. Se trata de un punto intermedio entre los dos algoritmos anteriores donde se utiliza un subconjunto pequeño, generalmente entre 10 y 1000 ejemplos de entrenamiento, seleccionados al azar para realizar la actualización de pesos en cada paso.

1.2.4. Redes Multicapa adecuadas para el Algoritmo Backpropagation

Como se explicó en la sección 1.2.1 un simple perceptrón sólo puede representar una superficie de decisión lineal, por lo que su efectividad está limitada

a datos linealmente separables. Por el contrario, el tipo de redes multicapa que se pueden obtener utilizando el algoritmo de Backpropagation son capaces de representar una gran variedad de superficies de decisión no lineales y complejas, como la que se observa en la Figura 1.7.

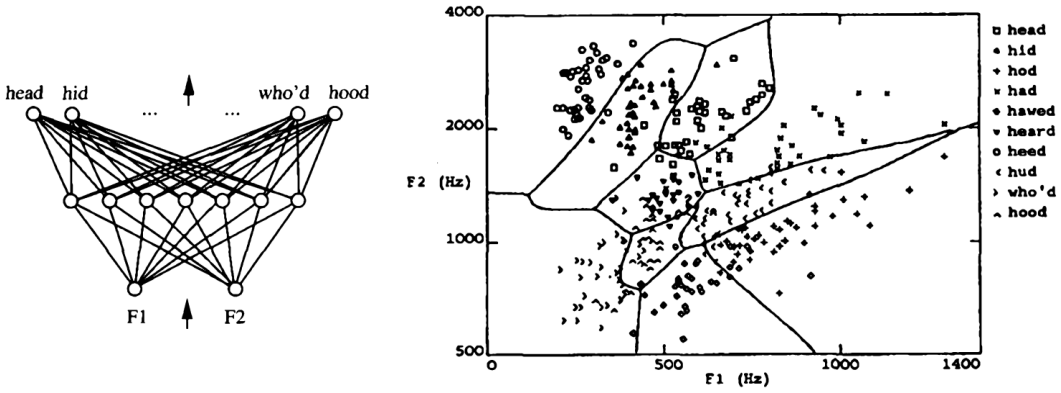


FIGURA 1.7: Regiones de decisión no lineales (derecha) para una red neuronal multicapa (izquierda). La red fue entrenada para reconocer entre la pronunciación de 10 palabras similares. La entrada de la red consiste de 2 valores, F1 y F2, obtenidos de un análisis espectral de dichas pronunciaciones. Los distintos puntos que componen las diferentes regiones se corresponden con datos de test que no fueron utilizados para el entrenamiento de la red [Mit97].

¿Qué tipo de unidades se deben usar para la construcción de dichas redes neuronales multicapa? Podríamos sentirnos tentados de elegir las unidades lineales presentadas en la sección 1.2.2 para las cuales ya tenemos una regla de actualización utilizando el descenso por gradiente. El problema radica en que la combinación de sucesivas capas de unidades lineales también producen una separación lineal.

Otra opción sería utilizar el perceptrón, pero dicha función tiene una respuesta discreta (basada en el umbral) y por lo tanto su derivada es cero en cualquier punto, lo que la hace no apta para utilizarla con el método de descenso por gradiente. Además, otro problema presente en los perceptrones es su inestabilidad, dado que un pequeño cambio en los pesos de un perceptrón de la red puede causar que la salida del mismo cambie rotundamente, de -1 a 1.

Necesitamos una unidad cuya salida sea una función no lineal de su entrada, y sea diferenciable en todo su dominio. Una solución que se encontró es utilizar la función o unidad sigmoide, la cual tiene forma de "S", es continua y diferenciable y aplica un umbral tal como lo hace el perceptrón. Podemos ver una representación gráfica de dicha unidad en la Figura 1.8.

Tal como el perceptrón, se computa una combinación lineal de la entrada y luego se aplica un umbral al resultado. Sin embargo, la diferencia reside en que

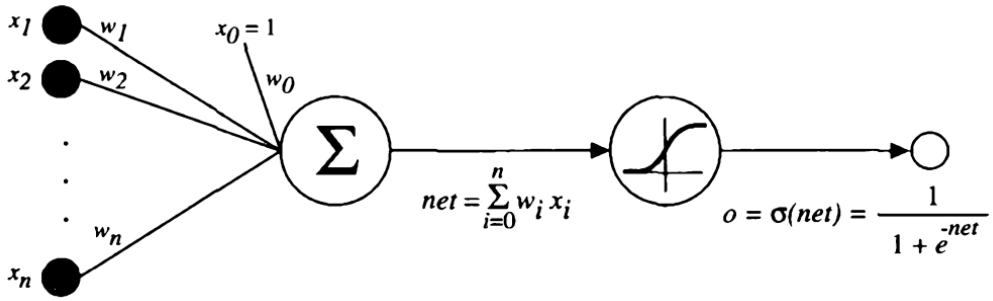


FIGURA 1.8: Unidad Sigmoide [Mit97].

la salida es continua. Más precisamente la salida o se computa:

$$o = \sigma(\vec{w} \cdot \vec{x})$$

donde

$$\sigma(y) = \frac{1}{1 + e^{-y}}$$

La función σ es la que recibe el nombre de función sigmoide, o también, función logística. Notar que su conjunto de imagen es el intervalo $(0, 1)$, y se trata de una función monótona creciente. La función sigmoide tiene la particularidad de que su derivada se expresa sencillamente en términos de su salida, en particular: $\frac{\partial \sigma(y)}{\partial y} = \sigma(y) \cdot (1 - \sigma(y))$.

Otras funciones diferenciables se suelen utilizar en lugar de σ. Por ejemplo, el término e^{-y} a veces es reemplazado por $e^{-k \cdot y}$ donde k es una constante positiva que determina la pendiente del umbral. Otras funciones muy usadas son tanh y $ReLU(x) = \text{máx}(x, 0)$. Siendo esta última, o variantes de la misma, el estándar a utilizar en la actualidad.

1.2.5. Algoritmo Backpropagation

El algoritmo de Backpropagation permite aprender los pesos adecuados para redes neuronales multicapa. Utiliza el método de descenso por gradiente para minimizar el error determinado por la función de costo entre la salida de la red y los valores objetivo.

El problema que enfrenta este algoritmo es encontrar los valores adecuados para los parámetros de la red entre un inmenso espacio de posibles hipótesis definido por todos los valores reales que puede tomar cada uno de los pesos asociados a las neuronas que componen la red. Esta búsqueda se puede visualizar de manera similar a la Figura 1.6 con ciertas diferencias como la posibilidad de tener múltiples neuronas de salida que dan lugar a una función de error de la

forma:

$$E(\vec{w}) = \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2$$

donde *outputs* es el conjunto de neuronas de salida de la red y t_{kd} y o_{kd} son el valor objetivo y el valor de salida producido por la red (dadas las funciones de activación en cada nodo), respectivamente, asociados a la *k-ésima* unidad de salida y ejemplo de entrenamiento d. La superficie del error en estos casos de redes neuronales con gran cantidad de pesos se encuentran en espacios de muy alta dimensionalidad, más específicamente, una dimensión por cada peso de la red.

Otra diferencia importante es que en el caso de redes multicapa, la superficie del error puede contener múltiples puntos de ensilladura. Lamentablemente, esto implica que el gradiente es igual a cero en dichos puntos, estancando el entrenamiento. Por este motivo existen algoritmos de descenso por gradiente más especializados como Adam [KB14] que intentan resolver este problema.

Para la explicación del algoritmo, consideraremos el caso más simple con una única neurona de entrada, una de salida y 2 capas intermedias con 2 neuronas cada una. Por otro lado, cada ejemplo de entrenamiento es un par de la forma $\langle x_{input}, y_{target} \rangle$, donde x_{input} es el vector de valores de entrada e y_{target} es el valor objetivo en la salida de la red. El optimizador utilizado será Descenso por Gradiente Estocástico.

1. **Parámetros de la red:** Todas las neuronas de capas vecinas están conectadas por medio de los pesos w_{ij}, que constituyen los parámetros de la red.

2. **Función de activación:** Cada uno de los nodos de la red tiene una entrada x, una función de activación $f(x)$ y una salida $y = f(x)$. Como explicamos anteriormente, $f(x)$ debe ser una función no lineal.

   La función que utilizaremos para la explicación es la función sigmoide: $f(x) = \frac{1}{1+e^{-x}}$.

3. **Función de costo:** El objetivo del algoritmo es aprender los pesos adecuados para la red de manera automática a partir de los datos, de forma que la salida producida y_{output} esté lo más cerca posible al valor objetivo y_{target} para todas las posibles entradas x_{input}.

   Para tener una medida cuantitativa de nuestro objetivo, usamos la función de costo E. Una elección muy común es usar la función dada en la ecuación 1.1.

4. **Propagación hacia adelante:** Primero, tomamos un ejemplo de entrenamiento $\langle x_{input}, y_{target} \rangle$ y lo introducimos en la red. Para mantener la consistencia entre los diferentes nodos, consideramos las neuronas de entrada como cualquier otra neurona de la red, con la diferencia de que la función de activación es la identidad, de manera que $y_1 = x_{input}$.

   Una representación esquemática de la red se observa en la Figura 1.9a

   Continuamos la propagación hacia la primer capa de la red. Tomando las salidas y de los nodos anteriores y usando los pesos para calcular la nueva entrada al nodo correspondiente en la primer capa: $x_j = \sum_{i \in in(j)} w_{ij} y_i + b_j$ donde $in(j)$ son las salidas de los nodos de la capa anterior. Luego, calculamos su salida y_j usando la función de activación (ver Figura 1.9b).

   Continuamos con este mismo procedimiento hasta obtener la salida final de la red.

5. **Derivada del error:** El algoritmo de backpropagation decide cuánto actualizar cada peso de la red comparando la salida obtenida con la salida deseada para un ejemplo particular. Para esto, necesitamos calcular cómo cambia el error con respecto a cada peso $\frac{\delta E}{\delta w_{ij}}$. Una vez obtenidas las derivadas, podemos actualizar los pesos mediante la regla: $w_{ij} = w_{ij} - \eta \frac{\delta E}{\delta w_{ij}}$, donde η es el *learning rate* ya explicado.

   Para el cálculo de $\frac{\delta E}{\delta w_{ij}}$ debemos mantener, para cada nodo dos derivadas adicionales (ver Figura 1.9c):

   - Cómo cambia el error con respecto a la entrada de dicho nodo: $\frac{\delta E}{\delta x}$.
   - Cómo cambia el error con respecto a la salida de dicho nodo: $\frac{\delta E}{\delta y}$.

6. **Back propagation — Propagación hacia atrás:** Habiendo calculado la salida para un ejemplo de entrenamiento, calculamos la derivada del error con respecto a dicha salida. Dada nuestra función de costo, y considerando este único ejemplo, tenemos: $E = \frac{1}{2}(y_{output} - y_{target})^2$, y por lo tanto, $\frac{\delta E}{\delta y_{output}} = y_{output} - y_{target}$.

   Ahora, calculamos $\frac{\delta E}{\delta x}$ mediante la regla de la cadena:

   $$\frac{\delta E}{\delta x} = \frac{\delta y}{\delta x} \frac{\delta E}{\delta y} = \frac{\delta}{\delta x} f(x) \frac{\delta E}{\delta y}$$

   considerando que $\frac{\delta}{\delta x} f(x) = f(x)(1 - f(x))$ siendo f la función sigmoide.

   Así, teniendo la derivada del error con respecto a la entrada de un determinado nodo, podemos ahora calcular la derivada del error con respecto

a los pesos involucrados en dicha entrada (ver Figura 1.9d):

$$\frac{\delta E}{\delta w_{ij}} = \frac{\delta x_j}{\delta w_{ij}} \frac{\delta E}{\delta x_j} = y_i \frac{\delta E}{\delta x_j}$$

Para cerrar el ciclo, usando la regla de la cadena nuevamente podemos obtener $\frac{\delta E}{\delta y}$ de la capa anterior (ver Figura 1.9d):

$$\frac{\delta E}{\delta y_i} = \sum_{j \in out(i)} \frac{\delta x_j}{\delta y_i} \frac{\delta E}{\delta x_j} = \sum_{j \in out(i)} w_{ij} \frac{\delta E}{\delta x_j}$$

Lo único que resta es repetir este procedimiento hasta haber calculado todas las derivadas del error.

1.2.6. Optimizador Adam

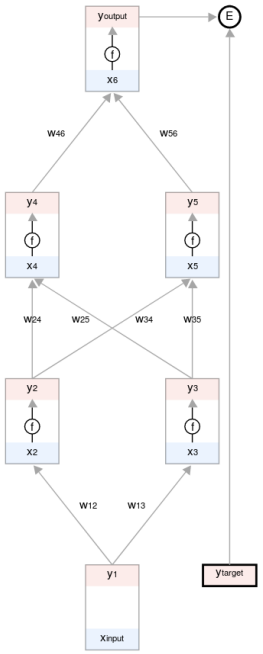
Todas las secciones anteriores fueron explicadas utilizando el algoritmo de Descenso por Gradiente debido a que éste es el algoritmo que dio origen a las redes neuronales artificiales. Sin embargo, en la práctica generalmente se utilizan otros algoritmos que se basan en la misma idea pero lo extienden con particularidades que los hacen más eficientes. Entre ellos se pueden mencionar: Momentum, Nesterov Accelerated Gradient, Adagrad, Adadelta, RMSprop, Adam, AdaMax y Nadam.

En la presente tesina utilizamos Adam, y por lo tanto, lo explicaremos con más detalle. Existe un excelente trabajo comparando todos estos algoritmos realizado por Sebastian Ruder: *An overview of gradient descent optimization algorithms* [Rud16].

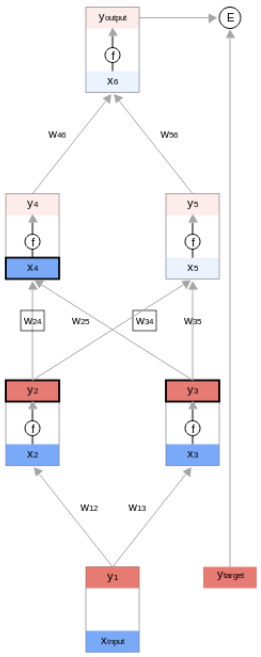
El algoritmo original de descenso por gradiente utiliza un único *learning rate* para todas las actualizaciones de pesos y éste no cambia durante el entrenamiento. En Adam, por el contrario, se mantiene un *learning rate* para cada peso (o parámetro) de la red, los cuales se van adaptando de manera independiente a medida que transcurre el entrenamiento.

Los autores expresan que Adam combina las ventajas de otras dos extensiones del descenso por gradiente estocástico, en particular:

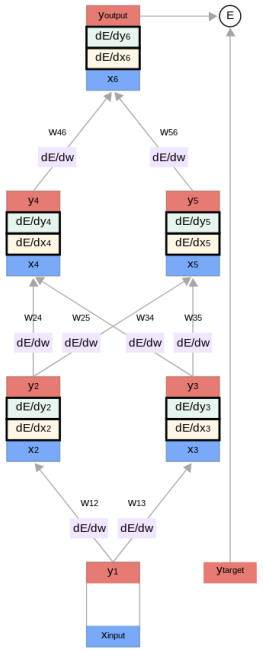
- **Adaptive Gradient Algorithm (AdaGrad)** el cual mantiene un *learning rate* por parámetro que mejora el rendimiento en problemas con gradientes dispersos.
- **Root Mean Square Propagation (RMSProp)** también utiliza un *learning rate* por peso, pero en este caso es adaptado según la media de los



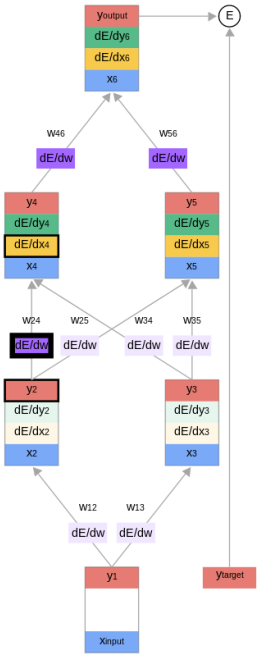
(A) Representación esquemática de la red.



(B) Propagación hacia adelante.



(C) Cálculo de las derivadas.



(D) Propagación hacia atrás.

FIGURA 1.9: Visualización del Algoritmo Backpropagation [Goo18a].

valores recientes de los gradientes para dicho peso de la red (ej: cuán rápido está cambiando) por lo que resulta útil en problemas no estacionarios o aprendizaje automático en línea (online machine learning).

En vez de adaptar el *learning rate* basado en el primer momento (la media) como en RMSProp, Adam también utiliza el segundo momento (la varianza) de

los gradientes.

Específicamente, el algoritmo calcula una media móvil con decaimiento exponencial del gradiente (g_t) y del gradiente al cuadrado (g_t^2), y los parámetros β_1 y β_2 controlan su tasas de decaimiento:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

Los valores recomendados por los autores para β_1 y β_2 son 0,9 y 0,999, respectivamente.

El valor inicial de las medias móviles (vectores de ceros) y los valores β_1 y β_2 cercanos a 1 (recomendado) dan como resultado un sesgo de las estimaciones de momento hacia cero. Esto se soluciona calculando las siguientes estimaciones corregidas por sesgo:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Así, la regla de actualización de pesos queda:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t + \epsilon}} \hat{m}_t$$

con un valor recomendado para $\epsilon = 10^{-8}$.

1.3. Redes Neuronales Convolucionales

Veamos ahora un tipo más específico de Redes Neuronales: las Redes Neuronales Convolucionales. Éstas reciben su nombre por estar compuestas principalmente por capas convolucionales en vez de capas completamente conectadas.

Cabe aclarar que estas redes son usadas principalmente en el campo de reconocimiento de patrones en imágenes lo cual nos permite adaptar la arquitectura de red utilizada para explotar las características propias de este dominio y además reducir el número de parámetros que deben ser aprendidos.

Una de las principales limitaciones de las redes con capas completamente conectadas es su complejidad computacional para procesar imágenes. Por ejemplo, al trabajar con un dataset de imágenes RGB de 64 píxeles por 64 píxeles, el número de pesos para cada neurona de la primer capa completamente conectada es de $64 \times 64 \times 3 = 12288$. Por otro lado, utilizando una capa convolucional con un filtro o *kernel* de 5×5 nos da un total de $75 = 5 \times 5 \times 3$ pesos por neurona.

1.3.1. Capas convolucionales

Mientras que las capas densas o completamente conectadas aprenden patrones globales de sus datos entrada (ej: para el dataset MNIST, patrones que involucran todos los píxeles en conjunto), las capas convolucionales aprenden patrones locales (en el caso de imágenes: patrones en pequeñas ventanas de la imagen de entrada, como en la Figura 1.10).

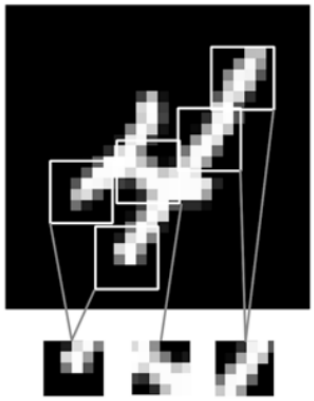


FIGURA 1.10: Las imágenes se descomponen en patrones locales como bordes, texturas, etc [Cho17].

Esta forma de procesar la entrada tiene propiedades interesantes:

- **Los patrones aprendidos no dependen de la ubicación del objeto en la imagen (translation invariant)**. Al aprender un determinado patrón en la esquina inferior derecha de la imagen, la RNC puede reconocerlo en cualquier parte, por ejemplo, en la esquina superior izquierda.
- **Se aprenden jerarquías espaciales de patrones (Figura 1.11)**. Una primer capa de convolución aprende patrones locales pequeños como bordes o texturas, la siguiente capa aprende patrones más complejos compuestos de los anteriores y así sucesivamente hasta llegar a la representación abstracta como ser "gato", "perro" o "dígito 0".

Las convoluciones operan sobre tensores 3D, llamados *feature maps*, que poseen dos ejes espaciales (*ancho* y *alto*) y un eje de *profundidad*, también denominado *canales*.

La operación de convolución extrae fragmentos o parches del *feature map* de entrada y aplica la misma transformación a todos los parches, produciendo un nuevo *feature map* de salida, el cuál también es un tensor 3D. Su profundidad o cantidad de canales es un parámetro de la capa de convolución, y en este caso ya no se corresponde con los canales RGB de una imagen sino que se los

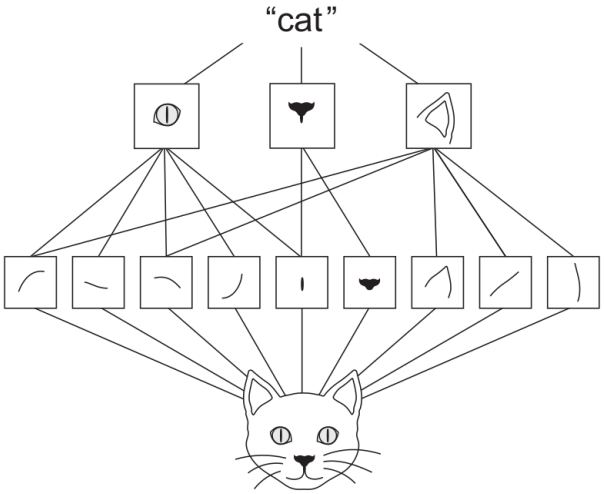


FIGURA 1.11: El mundo visual forma una jerarquía espacial de módulos visuales: los bordes locales se combinan en objetos como ojos u orejas, que a su vez se combinan en conceptos de alto nivel como “gato” [Cho17].

llama *filtros*. Estos filtros pueden codificar diferentes aspectos de la imagen de entrada, como por ejemplo, el concepto de “existe un rostro en la imagen”.

Para explicar la transformación mencionada, restrinjamos el tensor de entrada a una imagen de 1 canal, es decir, escala de grises y supongamos que tenemos un kernel 3×3 con los valores:

0	-1	0
-1	5	-1
0	-1	0

Por cada bloque de 3×3 píxeles de la imagen de entrada, multiplicamos cada píxel por la entrada correspondiente del kernel y luego sumamos los resultados. Esa suma se convierte en un nuevo píxel en la imagen de salida. Este procedimiento se puede observar en la Figura 1.12.

En el caso de una imagen de entrada con múltiples canales, supongamos 3, se realiza el mismo procedimiento y las 3 sumas resultantes son adicionadas nuevamente para generar el valor final.

Las capas de convolución tienen 2 parámetros principales:

- **Tamaño del filtro o kernel.** Los valores más usados son 3×3 o 5×5.
- **Profundidad del *feature map* de salida.** Corresponde a la cantidad de filtros producidos por la capa de convolución y como dijimos es un parámetro arbitrario.

Específicamente, el proceso de convolución consiste en *deslizar* el kernel, supongamos de un tamaño 3×3, sobre el tensor 3D de entrada y calculando,

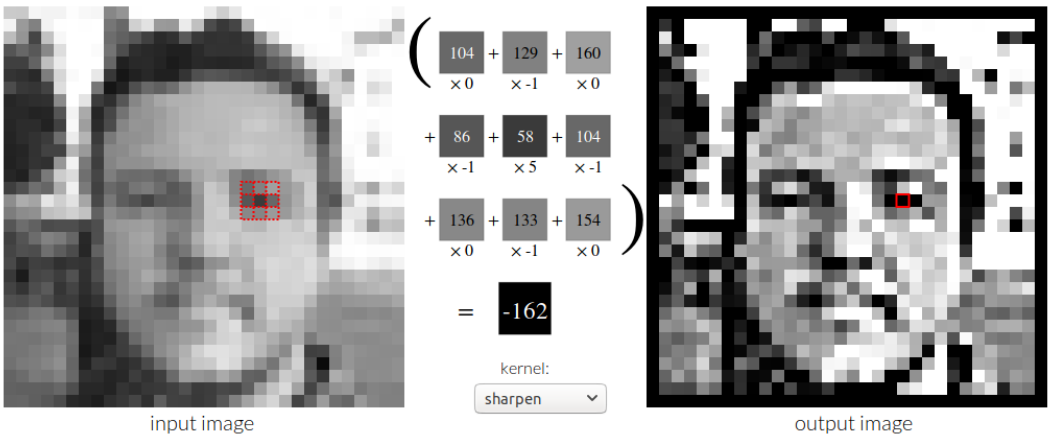


FIGURA 1.12: Convolución entre la imagen en escala de grises y el kernel antes descripto. En este caso, el kernel es un caso particular para aumentar la nitidez [Pow18].

en cada posible ubicación, la transformación recientemente explicada entre el kernel y el parche de la imagen, generando un nuevo vector 1D de dimensión $(cantidad\_de\_filtros,)$. Todo estos vectores se re-ensamblan para formar el *feature map* de salida. Las ubicaciones espaciales se mantienen entre el *feature map* de entrada y de salida, es decir, la porción inferior-derecha de la salida contiene información de la porción inferior-derecha de la entrada. Este proceso se detalla en la Figura 1.13.

El ancho y alto del *feature map* de salida puede diferir del de entrada por 2 razones principales:

- La propia operación de convolución genera, como dijimos, salidas de dimensión $(cantidad\_de\_filtros,)$ a partir de entradas de 3×3 o 5×5, según la configuración. Una forma de mantener la dimensionalidad del volumen de entrada es mediante el uso de *padding*.
- El uso de *strides* mayores a 1.

Padding

Considerando una entrada de 5×5, existen 9 formas distintas de centrar una ventana de 3×3 (Figura 1.14) por lo que la salida será de $3 \times 3 \times cantidad\_de\_filtros$.

Para mantener el ancho/alto en la salida al igual que en la entrada podemos utilizar *padding*. El *padding* o *zero-padding* consiste en agregar columnas y filas de ceros en los bordes de la imagen de entrada aumentando su dimensionalidad. De esta forma, si agregamos un *padding* de 1, las maneras posibles de centrar

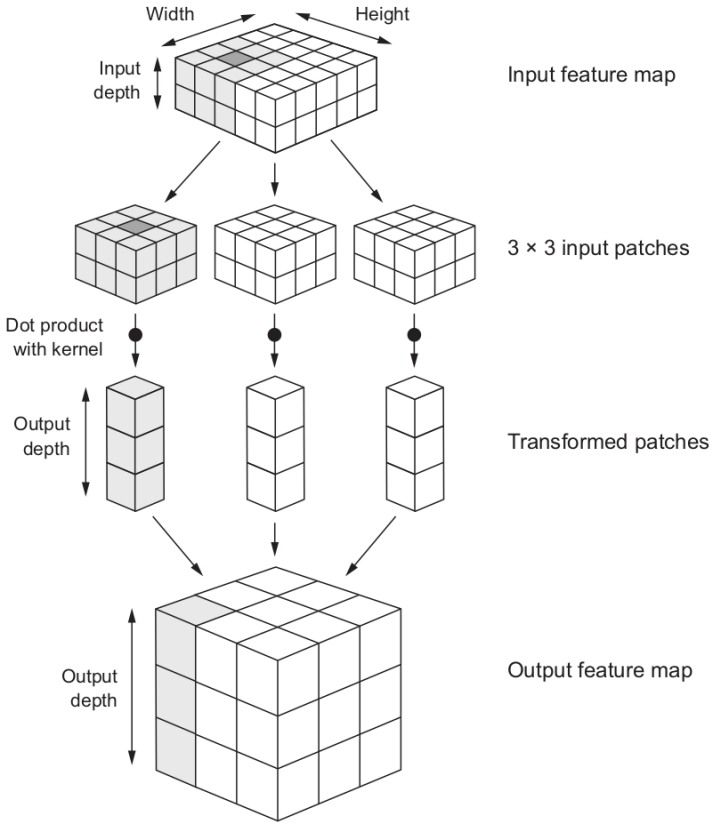


FIGURA 1.13: Cómo funciona el proceso de convolución [Cho17].

una ventana de 3×3 aumentan a 25, por lo que la capa de convolución no altera la dimensionalidad del volumen de entrada (Figura 1.15).

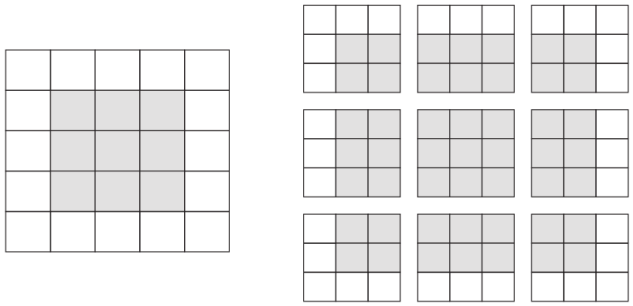


FIGURA 1.14: Posibilidades para centrar una ventana de 3×3 en un *feature map* de entrada de 5×5 [Cho17].

Strides

El otro factor que influye en la dimensión de salida es la noción de *strides*. Hasta ahora hemos asumido que todos los parches son contiguos pero la distancia entre ventanas vecinas es un parámetro de la convolución, llamado *stride* y cuyo valor por defecto es 1.

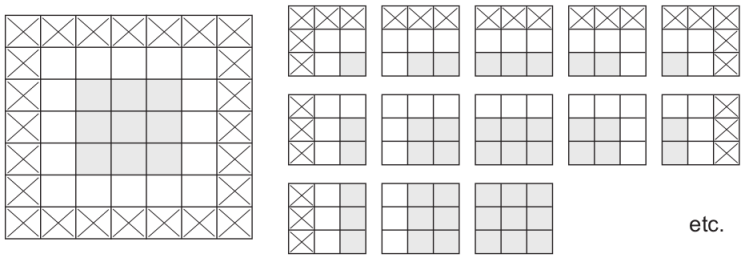


FIGURA 1.15: Agregando padding al *feature map* de entrada [Cho17].

En la Figura 1.16 se observa una convolución con tamaño de kernel 3×3 y $stride = 2$. Esto se traduce en un *feature map* cuyas dimensiones de *ancho* y *alto* son reducidas a la mitad (además de la reducción propia al proceso de convolución). Concretamente, la dimensión de salida es:

$$O = \frac{W - K + 2P}{S} + 1$$

donde O es el ancho/alto de salida, W es el ancho/alto de entrada, K es el tamaño del kernel, P el padding aplicado y S el valor de stride.

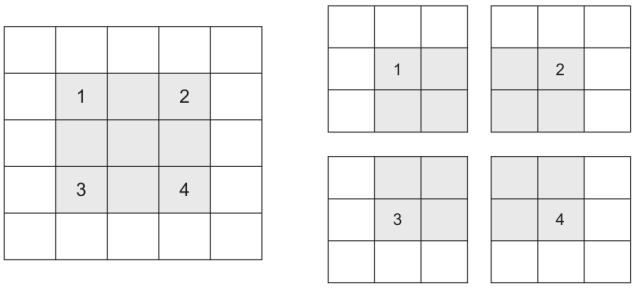


FIGURA 1.16: Convolución con $stride = 2$ [Cho17].

En la práctica, para reducir la dimensionalidad de los volúmenes de salida se suelen usar capas de *pooling* que veremos a continuación.

1.3.2. Capas de pooling

La capa de *pooling* más utilizada en la práctica es *Max Pooling*. La transformación realizada por esta capa es muy simple, su funcionamiento se puede observar en la Figura 1.17. Básicamente, al igual que la convolución consiste en tomar parches del volumen de entrada y devolver el máximo valor en cada canal. Generalmente se utiliza un tamaño de kernel de 2×2 y $stride = 2$ lo que reduce la dimensionalidad de los *feature maps* a la mitad.

¿Por qué reducir la dimensionalidad de los feature maps?

Una de las razones principales es reducir la cantidad pesos a procesar en las etapas finales de la red. Como veremos en la próxima sección, una práctica muy común es utilizar capas completamente conectadas luego de bloques de capas convolucionales. Si nuestra última capa convolucional devuelve 64 filtros de 22×22 y aplicamos una capa completamente conectada de 512 neuronas, nos da como resultado $22 \times 22 \times 64 \times 512 \approx 15{,}8$ millones de pesos sólo en esta conexión.

El otro motivo es generar jerarquías espaciales de filtros haciendo que las sucesivas capas convolucionales consideren ventanas cada vez más grandes con respecto a la entrada original de la red. Si por ejemplo aplicamos 3 capas convolucionales sucesivas sin ninguna capa de *Max Pooling* en el medio, cada una con tamaño de kernel 3×3 y $stride = 1$, las ventanas de 3×3 en la última capa sólo tendrán información proveniente de una ventana de 7×7 con respecto a la imagen original (imagine tener que clasificar dígitos considerando sólo ventanas de 7×7 píxeles). Necesitamos que los filtros generados por la última capa contengan información que abarque la totalidad de la imagen original.

Además de *Max Pooling*, este tipo de capas puede aplicar otras funciones, como *Average Pooling* o *L2-norm Pooling*. Sin embargo, en la práctica generalmente se utiliza *Max Pooling* ya que es la que ha demostrado mejores resultados, y también, aunque en menor medida, *Average Pooling*.

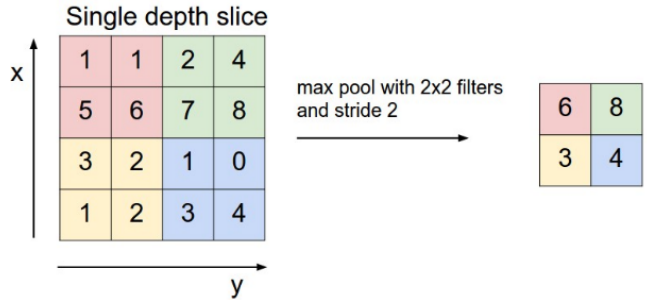


FIGURA 1.17: Capa de *Max Pooling* con filtros de 2×2 y $stride = 2$ [Kar18].

1.3.3. Arquitectura comúnmente utilizada

Una arquitectura muy utilizada al construir una Red Neuronal Convolucional (RNC) es apilar bloques de 2 o 3 capas convolucionales seguidas por una capa de pooling, y al final una serie de capas completamente conectadas, tal como se observa en la Figura 1.18

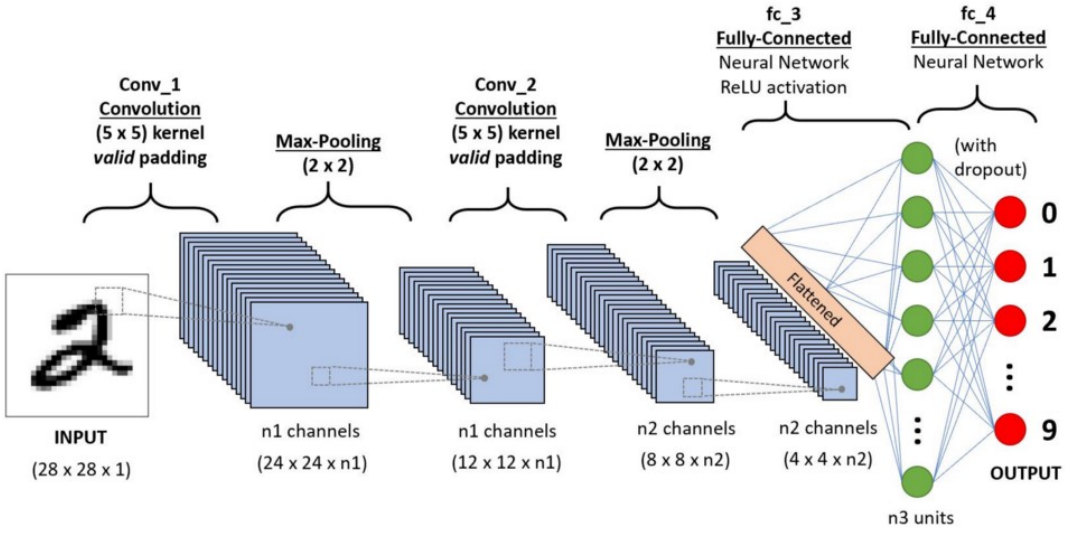


FIGURA 1.18: Ejemplo de arquitectura muy utilizada para una RNC [Sah18].

1.4. Redes Adversarias Generativas

Pasemos ahora a analizar otro tipo distinto de Redes Neuronales: las Redes Adversarias Generativas, más comúnmente conocidas por su nombre en inglés Generative Adversarial Networks (GAN).

Primero, veamos cómo funcionan los algoritmos generativos, y para ello resulta útil compararlos con algoritmos discriminativos.

Los algoritmos discriminativos clasifican datos de entrada, es decir, dadas las características de un ejemplo determinado, predicen una etiqueta o categoría a la cual pertenece dicho ejemplo. Por ejemplo, dadas las palabras de un email, un algoritmo discriminativo inferirá si es *spam* o *no_spam*. Éstas son las posibles categorías y el conjunto de palabras que componen el email son sus características. Matemáticamente, el algoritmo devolverá $p(y|x)$, donde y representa la etiqueta y x las características, o sea la probabilidad de que el email sea *spam* dado el conjunto de palabras que contiene.

Una forma de pensar los algoritmos generativos es que realizan el proceso inverso. En vez de predecir una etiqueta dadas las características del ejemplo de entrada, intentan inferir estas características dada una etiqueta particular. La pregunta sería: "Asumiendo que el email es spam, ¿que tan probable es un determinado conjunto de características?". Por lo tanto, intentan capturar la probabilidad $p(x|y)$ (y $p(y)$), es decir, modela como se distribuyen las características (datos de entrada) para cada tipo de mensajes (*spam* o *no_spam*).

Esto es muy importante porque los algoritmos generativos abordan un problema más general y aprenden realmente como se estructuran y distribuyen

los datos de entrada mientras que los discriminativos simplemente aprenden a categorizar una variable objetivo en función de la entrada. Además, utilizando el Teorema de Bayes pueden ser utilizados como clasificadores al igual que los algoritmos discriminativos.

En [Goo+14], Goodfellow et al. presentaron la idea original de las Redes Adversarias Generativas. Estas redes son llamadas adversarias debido a que el problema se estructura de manera que existen dos entidades compitiendo entre si, donde cada una de ellas es un modelo de aprendizaje automatizado.

Supongamos que queremos construir un generador de imágenes de rostros. Comenzamos por alimentar uno de nuestros dos modelos, el *generador*, con números aleatorios. Este aplica diferentes transformaciones a dicha entrada y genera una imagen de salida. Repetimos este procedimiento una gran cantidad de veces.

Por el otro lado, tomamos una cantidad similar de imágenes reales provenientes de nuestro dataset y alimentamos estas últimas junto con las imágenes ficticias del paso anterior a nuestro otro modelo, el *discriminador*. El trabajo de éste es devolver un número para cada entrada que representa la probabilidad de que dicha imagen pertenezca al dataset original.

Por cada error en una imagen ficticia, el discriminador es penalizado y el generador es recompensado. El discriminador también es penalizado o recompensado según su clasificación sobre imágenes reales. Por este motivo se llaman redes adversarias. Al transcurrir el entrenamiento, la competencia lleva al perfeccionamiento mutuo de ambos modelos. Una representación gráfica de estos modelos se puede ver en la Figura 1.19.

El hecho que estos dos modelos sean redes neuronales nos permite utilizar el algoritmo de backpropagation y una función de costo/optimización adecuada para representar las penalizaciones/recompensas sobre cada uno de ellos.

Específicamente, la función a optimizar es:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{data}(x)}\big[\log D(x)\big] + \mathbb{E}_{z\sim p_z(z)}\big[\log(1 - D(G(z)))\big]$$

donde p_{data} es la distribución de probabilidad de los datos reales, p_z es la distribución de probabilidad para el muestreo de vectores aleatorios Z (generalmente una distribución normal o uniforme), $D(x)$ es la estimación retornada por el discriminador D de la probabilidad de que la imagen de entrada x sea un ejemplo real y, por último, G es una función diferenciable que mapea los vectores aleatorios de entrada z en imágenes del dominio de los datos reales. Ambos modelos G y D se construyen a partir de perceptrones multicapa.

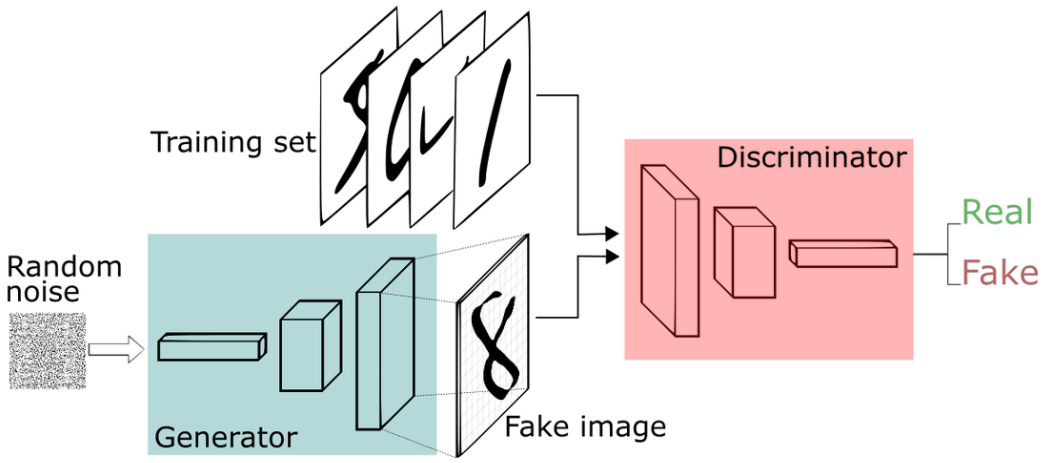


FIGURA 1.19: Esquema de una red adversaria generativa [Sil18].

El valor óptimo para esta función a optimizar se logra cuando la distribución de probabilidad del generador resulta idéntica a la de los datos reales, lo que significa que las imágenes generadas son indistinguibles de las originales. El algoritmo presentado en el paper [Goo+14] se reproduce en la Figura 1.20.

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

  for k steps **do**

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
  - Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for

  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

FIGURA 1.20: Algoritmo de entrenamiento de una red adversaria generativa [Goo+14].

El término $D(x)$ responde a la pregunta "¿Cuál es la probabilidad de que la entrada x provenga de los datos reales?". Si $x = G(z)$, entonces representa la predicción por parte del discriminador de la probabilidad de que un ejemplo

apócrifo sea considerado real. Si consideramos D y G por separado, G busca minimizar la función $V(D, G)$ mientras que D busca maximizarla.

En la Figura 1.21 se pueden observar resultados presentados en el paper original.

FIGURA 1.21: Visualización de ejemplos producidos por el generador del modelo. Las columnas resaltadas de la derecha corresponden a los datos reales más parecidos a los generados en la última columna de cada resultado, en un intento por demostrar que el modelo no está simplemente memorizando ejemplos. Los dataset son: (a) MNIST (b) Toronto Faces Dataset (c) CIFAR-10 (red neuronal completamente conectada) (d) CIFAR-10 (red neuronal convolucional para el discriminador y de-convolucional para el generador) [Goo+14].

Por último, cabe aclarar que la formulación original de GAN tiene algunos problemas como ser:

- Dificultad de entrenamiento: Hay casos donde los modelos no aprenden o convergen a un mínimo local.
- "Colapso de moda" — del inglés "Mode collapse": cuando el generador produce el mismo ejemplo para una gran cantidad de vectores aleatorios de entrada.

Estos problemas se fueron reduciendo en futuras implementaciones de GAN como ser el caso de DCGAN y Wasserstein GAN que analizaremos a continuación.

1.5. Redes Adversarias Generativas Convolucionales

Los autores Radford y col. [RMC15] proponen una serie de cambios y restricciones en las arquitecturas utilizadas para el generador/discriminador que permiten solucionar, o al menos aliviar, los problemas antes mencionados, entrenar modelos más profundos y generar imágenes de mayor resolución. Las mismas se enumeran a continuación:

1. Se utilizan redes completamente convolucionales con $stride > 1$ (*strided convolution*) en el discriminador y $stride = \frac{1}{n}$ con $n \in \mathbb{N}$ (*fractionally-strided convolution*) en el generador. Esto se realiza para eliminar las capas de pooling que producen una reducción de dimensionalidad determinista, permitiéndole al modelo aprender su propio sobremuestreo y submuestreo espacial, según sea el caso.

2. Se reduce el uso de capas completamente conectadas reemplazándolas por capas convolucionales, sólo la primer capa del generador y la última del discriminador son completamente conectadas. El generador toma como entrada vectores Z de ruido aleatorio, los cuales son reorganizados en tensores de 4 dimensiones (aplicando previamente una capa completamente conectada) para luego ser consumidos por las capas convolucionales (ver Figura 1.22). En el discriminador, por otro lado, la última capa de convolución se "aplana" y se provee como entrada a una única neurona sigmoide de salida.

3. Se utiliza la técnica de *Batch Normalization* la cual normaliza la entrada de cada neurona a media cero y varianza uno. Evitando así problemas en el entrenamiento causados por una mala inicialización de pesos y mejorando el flujo del gradiente en modelos profundos. En dicho trabajo, esta técnica es imprescindible para evitar que el generador colapse a un único punto generando siempre el mismo ejemplo. Batch Normalization no se aplica ni a la capa de entrada del discriminador, ya que resulta en un modelo inestable, ni a la capa de salida del generador, pues los datos

deben capturar libremente la media y desviación estándar de las imágenes reales.

4. Se utiliza ReLU como función de activación en el generador (excepto en la capa de salida, donde se utiliza Tanh). Observaron que el uso de una activación acotada permitía al modelo aprender más rápidamente a saturar y cubrir el espacio de colores de los datos de entrenamiento. En el discriminador se utiliza Leaky ReLU como activación, ofreciendo buenos resultados, especialmente en modelos de alta resolución.

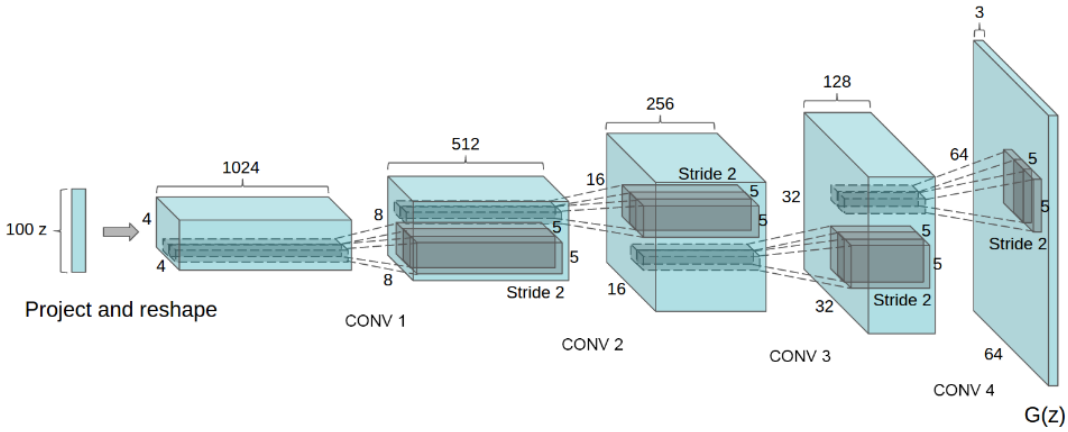


FIGURA 1.22: Generador de la arquitectura DCGAN. El discriminador es análogo [RMC15].

Entre los principales resultados del trabajo de Radford y col. [RMC15] podemos mencionar:

1. **Interpolaciones en el espacio latente:** esto permite obtener indicios en caso de memorización de muestras (en el caso de que se observen transiciones bruscas) y ver la forma en que el espacio se ordena jerárquicamente. Si al atravesar este espacio observamos cambios semánticos suaves en la generación de imágenes (como objetos que aparecen/desaparecen), podemos afirmar que el modelo ha aprendido representaciones relevantes e interesantes. Los resultados de estos experimentos se reproducen en la Figura 1.23.

2. **Visualizando los filtros del discriminador:** Se demuestra que al entrenar una DCGAN en un gran dataset de imágenes, ésta aprende una jerarquía de características interesantes. En la Figura 1.24 se puede ver cómo los filtros aprendidos por el discriminador se activan en partes típicas de la habitación, como camas o ventanas. Para comparar, también

se muestra el resultado sobre filtros inicializados de forma aleatoria en los cuales no se observa ninguna activación semánticamente interesante.

3. **Aritmética de vectores sobre el espacio latente:** En el trabajo [Mik+13] los autores demuestran que realizando operaciones aritméticas simples entre las representaciones aprendidas sobre las palabras del modelo *Skip-gram* se pueden obtener resultados interesantes, como el hecho de que el resultado de: $vector(\text{“}Madrid\text{”}) - vector(\text{“}Spain\text{”}) + vector(\text{“}France\text{”})$, es más cercano al $vector(\text{“}Paris\text{”})$ comparado con cualquier otro vector de palabra.

   Los autores del presente paper descubren que existe una estructura similar en las representaciones de vectores Z del espacio latente del generador. Realizaron operaciones aritméticas sobre vectores aleatorios cuyas imágenes generadas poseen cierta característica visual logrando resultados semejantes a los mencionados, los cuales se observan en la Figura 1.25.

4. **Utilizando un vector “de giro”**: Este último resultado está relacionado con el anterior. Se crea un vector “de giro” a partir de ejemplos de rostros mirando a la izquierda contra rostros mirando a la derecha. Sumando dicho vector a diferentes ejemplos se observa como cambia su orientación (ver Figura 1.26).

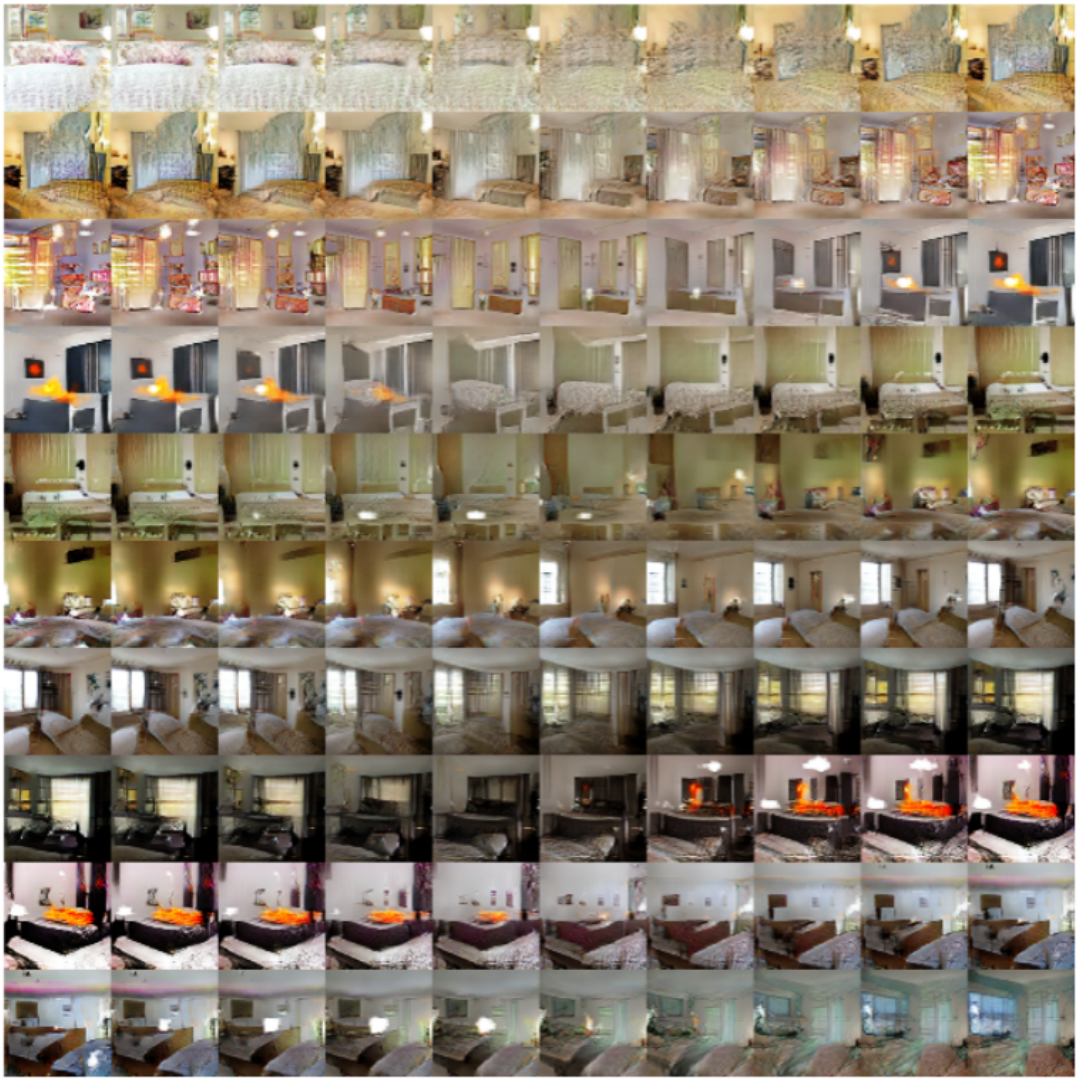


FIGURA 1.23: Interpolación entre 9 vectores aleatorios en Z muestran que el espacio aprendido tiene transiciones graduales donde todas las imágenes lucen como una habitación. En la última fila se observa lo que parece ser un televisor convertirse lentamente en una ventana [RMC15].

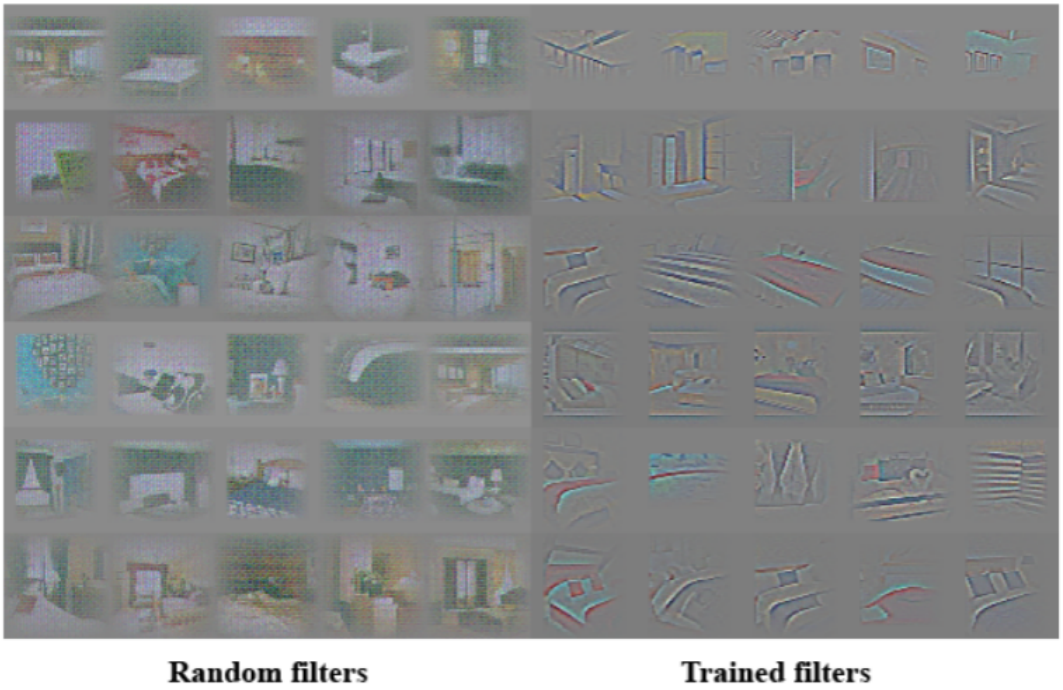


FIGURA 1.24: A la derecha se observa como los filtros aprendidos por el discriminador se activan en partes típicas de la habitación, como camas o ventanas. A la izquierda, se muestra el resultado sobre filtros inicializados de forma aleatoria en los cuales no se observa ninguna activación semánticamente interesante [RMC15].

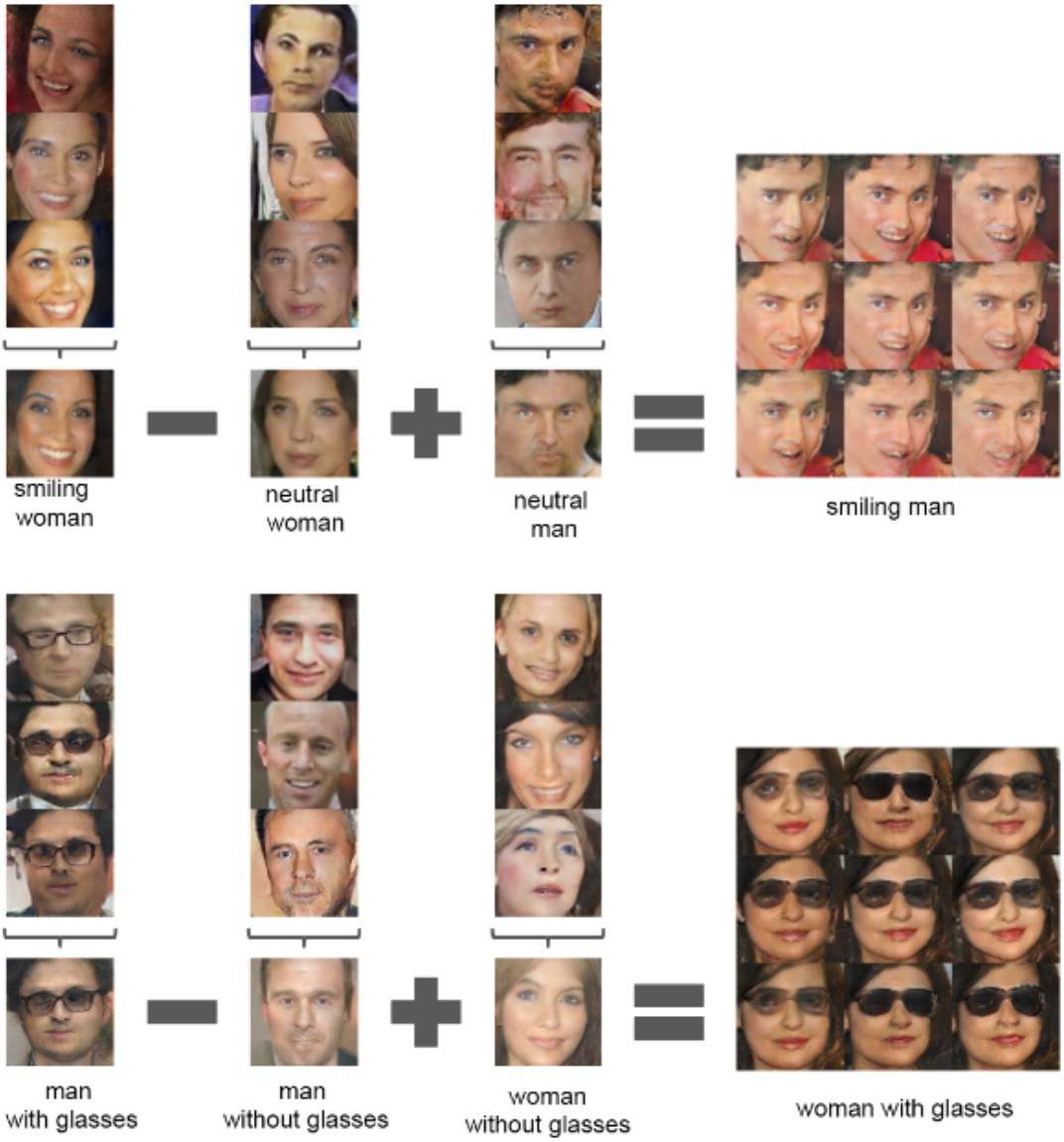


FIGURA 1.25: Aritmética de vectores que representan conceptos visuales [RMC15].

FIGURA 1.26: Se crea un vector "de giro" a partir de ejemplos de rostros mirando a la izquierda vs a la derecha. Sumando dicho vector a diferentes ejemplos se observa como cambia su orientación [RMC15].

1.6. Wasserstein GAN

Si bien los trabajos anteriores muestran resultados muy interesantes utilizando GAN, al igual que trabajos similares realizados por Denton y col. [Den+15], Salimans y col. [Sal+16] o Wu y col. [Wu+16], todos éstos se basan en la búsqueda heurística de arquitecturas estables.

A pesar de su éxito, hay poca o ninguna teoría que explique el comportamiento inestable del entrenamiento GAN. Lo que es peor, los enfoques para atacar este problema se basan en heurísticas que son extremadamente sensibles a los cambios. Esto hace que sea muy difícil experimentar con nuevas variantes o utilizarlas en nuevos dominios, lo que limita drásticamente su aplicabilidad.

Antes de enumerar los hallazgos en el trabajo *Towards Principled Methods for Training Generative Adversarial Networks* de Arjovsky y col. [AB17] veamos algunos conceptos que nos ayudarán a entenderlos.

1.6.1. Divergencia de Kullback-Leibler y Divergencia de Jensen–Shannon

Los enfoques tradicionales en modelos generativos se basaban en minimizar alguna de estas dos funciones de costo (Divergencia de K-L y J-S) entre la distribución de probabilidad desconocida de los datos reales, llamémosla p y la distribución de probabilidad del generador, llamémosla q. Las fórmulas correspondientes son las siguientes:

$$D_{KL}(P \parallel Q) = \sum_{x=1}^{N} P(x) \log \frac{P(x)}{Q(x)}$$

$$D_{JS}(P \parallel Q) = \frac{1}{2} D_{KL}(P \parallel \frac{P+Q}{2}) + \frac{1}{2} D_{KL}(Q \parallel \frac{P+Q}{2})$$

En la Figura 1.27 se observa la distribución p junto con 3 posibilidades distintas para la distribución q. Todas se asumen distribuciones normales y la diferencia entre las distribuciones q_i $(i = 1, ..., 3)$ es que poseen distintas medias. Graficando ambas divergencias entre p y q en función del valor de la media de q la cual varía entre 0 y 35 obtenemos los resultados de la Figura 1.28. Cuando p y q son idénticas, la divergencia es 0. A medida que la media de q aumenta, también crece la divergencia. Llega un punto donde el gradiente empieza a decrecer acercándose cada vez más a cero, momento en el cual el generador no recibe ninguna información del descenso por gradiente para mejorar su rendimiento.

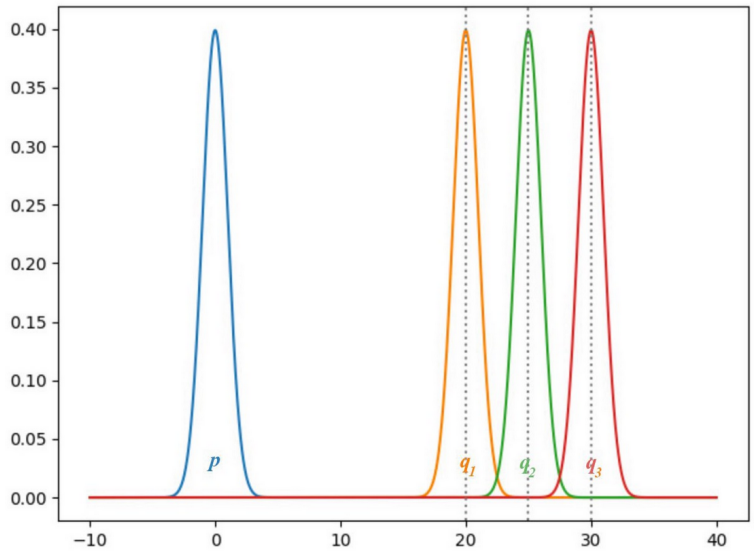


FIGURA 1.27: Distintas posibilidades, con diferentes medias, para la distribución de probabilidad q_i del generador [Hui18].

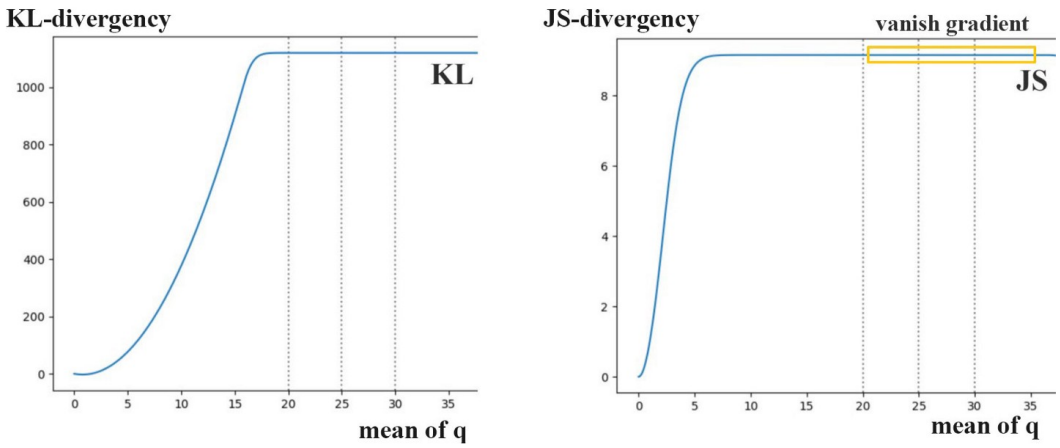


FIGURA 1.28: Divergencia de Kullback-Leibler y Divergencia de Jensen–Shannon en función de la media de la distribución q del generador [Hui18].

Teniendo esto en cuenta, veamos las conclusiones del trabajo antes mencionado:

1. Un discriminador óptimo produce información relevante para el entrenamiento del generador. Sin embargo, si el generador aún no está haciendo un buen trabajo, el gradiente disminuye y el generador no puede mejorar. Esta es la situación recientemente explicada. En la Ecuación 1.5 vemos la fórmula del gradiente del generador en la arquitectura GAN original.

$$\nabla_\theta \mathbb{E}_{z\sim p_z(z)}\big[\log(1 - D(G_\theta(z)))\big] \to 0 \tag{1.5}$$

2. Para evitar el problema de desaparición del gradiente (gradient vanishing problem), se propuso una función de costo alternativa para el entrenamiento del generador (Ecuación 1.6).

$$\nabla_\theta \mathbb{E}_{z\sim p_z(z)}\big[-\log D(G_\theta(z))\big] \tag{1.6}$$

   Si bien esta nueva función de costo no sufre el problema de desaparición del gradiente, en la práctica, la gran varianza del gradiente producido provoca pasos de actualización considerablemente inestables y por consiguiente, un modelo también inestable.

3. En este primer trabajo, Arjovsky propone agregar ruido a las imágenes generadas para estabilizar el modelo como vemos en la Figura 1.29.

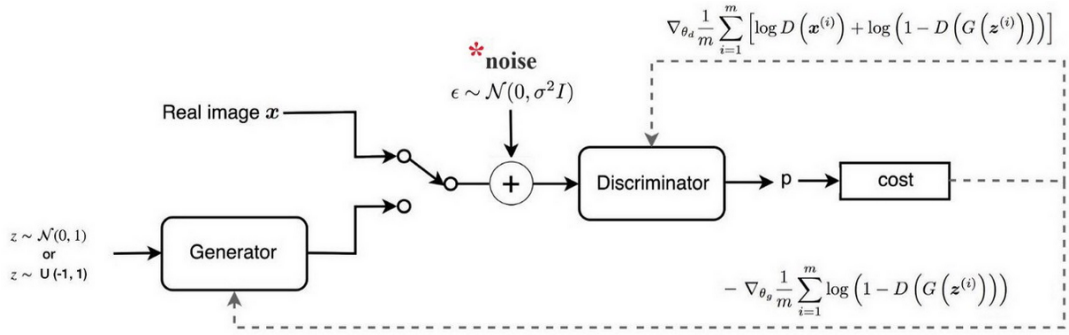


FIGURA 1.29: Arquitectura GAN propuesta por Arjovsky y col: Se agrega ruido a las imágenes que recibe el discriminador como una forma de estabilizar el modelo [Hui18].

1.6.2. Distancia Wasserstein / Earth-Mover (EM) distance

En vez de agregar ruido, en una nueva propuesta de Arjovsky y col. [ACB17] proponen una novedosa función de costo utilizando la distancia Wasserstein, explicada a continuación.

Supongamos que queremos mover las cajas de la izquierda en la Figura 1.30 a las posiciones demarcadas con línea de punto a su derecha. El costo de mover cada caja es la distancia "en bloques" entre una posición y la otra, en el ejemplo su valor es 6 para mover la caja número 1.

En la Figura 1.31 podemos ver dos posibilidades distintas de movimiento para reacomodar las cajas de la forma deseada. Las tablas de la derecha indican cómo se realizan los movimientos. Si sumamos estos desplazamientos teniendo

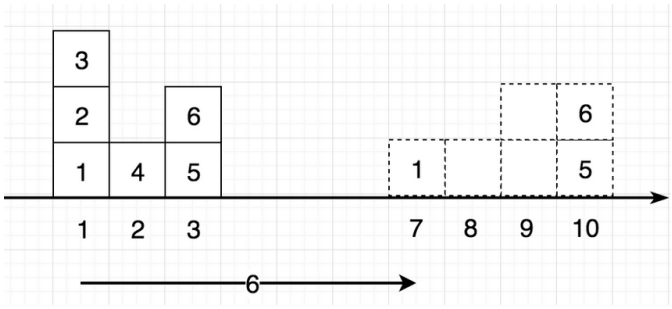


FIGURA 1.30: Distancia Wasserstein: costo de movimiento [Hui18].

en cuenta la posición de origen y la de destino, ambos planes tienen un costo de 42.

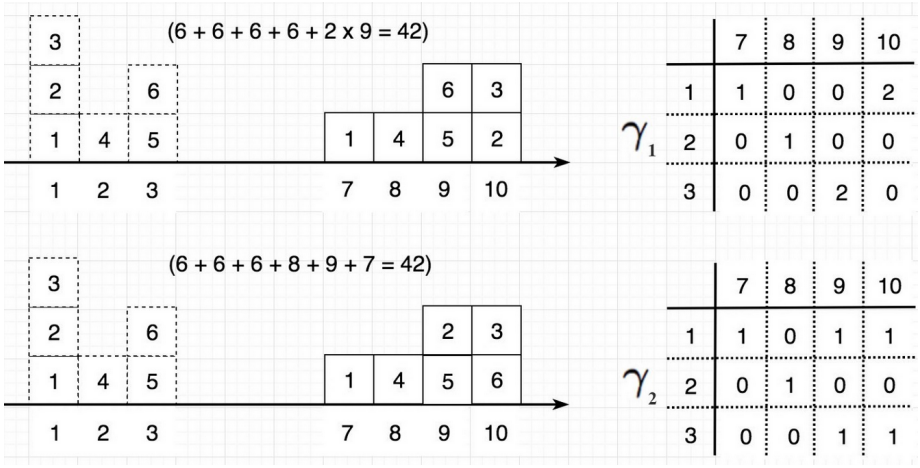


FIGURA 1.31: Distancia Wasserstein: dos posibilidades de movimiento distintas con igual costo [Hui18].

Sin embargo, no todas las formas de transportar las cajas tienen el mismo costo. Si consideramos el caso de la Figura 1.32, la distancia Wasserstein es el costo del plan más barato, en este caso 2.

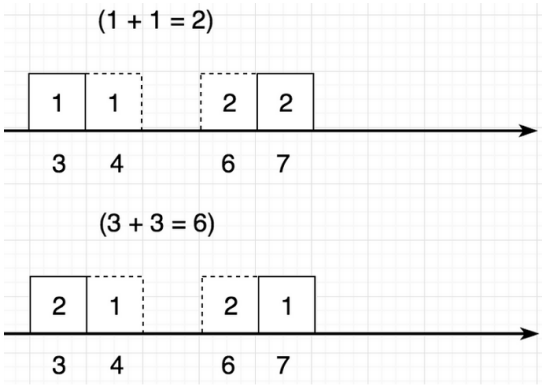


FIGURA 1.32: Distancia Wasserstein: dos posibilidades de movimiento distintas con distinto costo [Hui18].

Formalmente, la distancia Wasserstein es el costo mínimo de transportar masa al convertir la distribución de probabilidad q a la distribución de probabilidad p.

Esta nueva función de costo tiene la propiedad de poseer un gradiente más suave y le permite al generador aprender incluso cuando éste no está haciendo un buen trabajo. En la Figura 1.33 se encuentran graficadas dos distribuciones, la real en azul y la de un posible generador, falsa, en verde. Vemos además el valor producido por el discriminador (o crítico) ***D(X)*** para cada valor de X, tanto en el caso de la arquitectura GAN (en rojo) como el nuevo crítico (que reemplaza al discriminador) de la propuesta WGAN (en celeste). Para GAN se pueden observar las áreas de explosión y desaparición del gradiente, mientras que en el caso de WGAN, el gradiente es más suave en todo el dominio y permite que el generador mejore sin importar el rendimiento de éste.

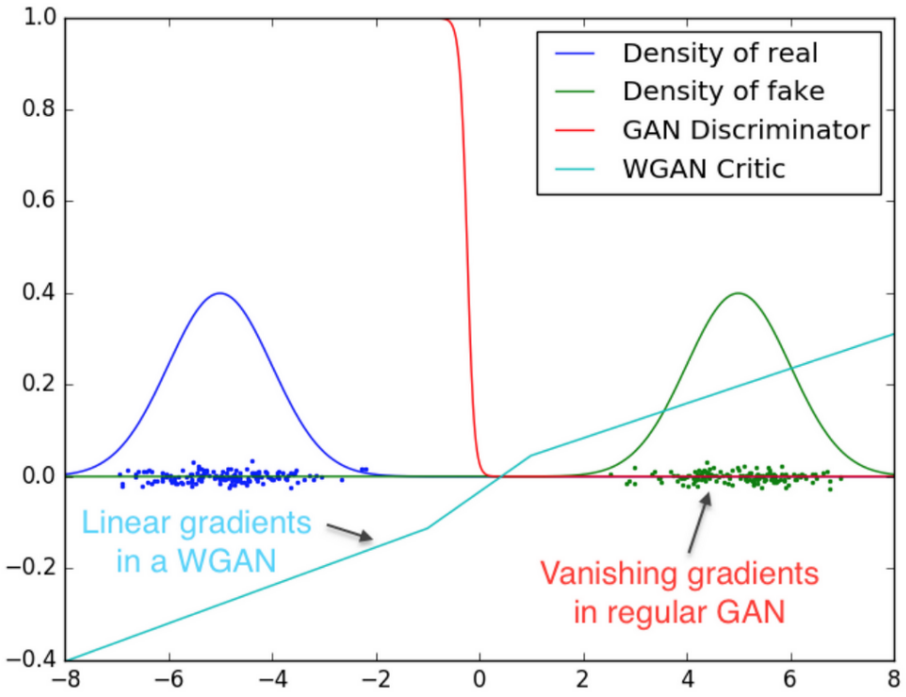


FIGURA 1.33: Comparación entre los valores producidos por el discriminador de GAN y el crítico de WGAN sobre una distribución real y una ficticia [ACB17].

El cómputo exacto de la distancia Wasserstein es intratable pero utilizando la dualidad Kantorovich-Rubinstein se puede simplificar el cálculo [Vil09] usando la equivalencia:

$$W(\mathbb{P}_r, \mathbb{P}_\theta) = \sup_{\|f\|_L \leq 1} \mathbb{E}_{x \sim \mathbb{P}_r}[f(x)] - \mathbb{E}_{x \sim \mathbb{P}_\theta}[f(x)]$$

donde f es una función lipschitziana, es decir: $|f(x_1) - f(x_2)| \leq |x_1 - x_2|$.

De esta forma, para calcular la distancia Wasserstein tenemos que encontrar la función lipschitziana f. Podemos construir una red profunda para aprenderla. De hecho, esta red es muy similar al discriminador de GAN, sin la última activación sigmoidea por lo que su salida es un escalar que representa un "puntaje" en vez de una probabilidad. Este puntaje se puede interpretar como "cuán reales son las imágenes de entrada". Además, al discriminador se lo pasa a llamar crítico para reflejar este nuevo rol. En la Figura 1.34 podemos ver un esquema de ambas arquitecturas, donde la diferencia principal es la función de costo utilizada.

Para cumplir con la restricción de la función f, WGAN aplica un procedimiento muy simple para restringir el valor de los pesos que se basa en el recorte de los mismos, llamado *weight clipping*. Es decir, los pesos del crítico deben estar dentro de un cierto rango controlado por el hiper-parámetro c del modelo. Estos son los pasos 6 y 7 del algoritmo que se presenta en la Figura 1.35.

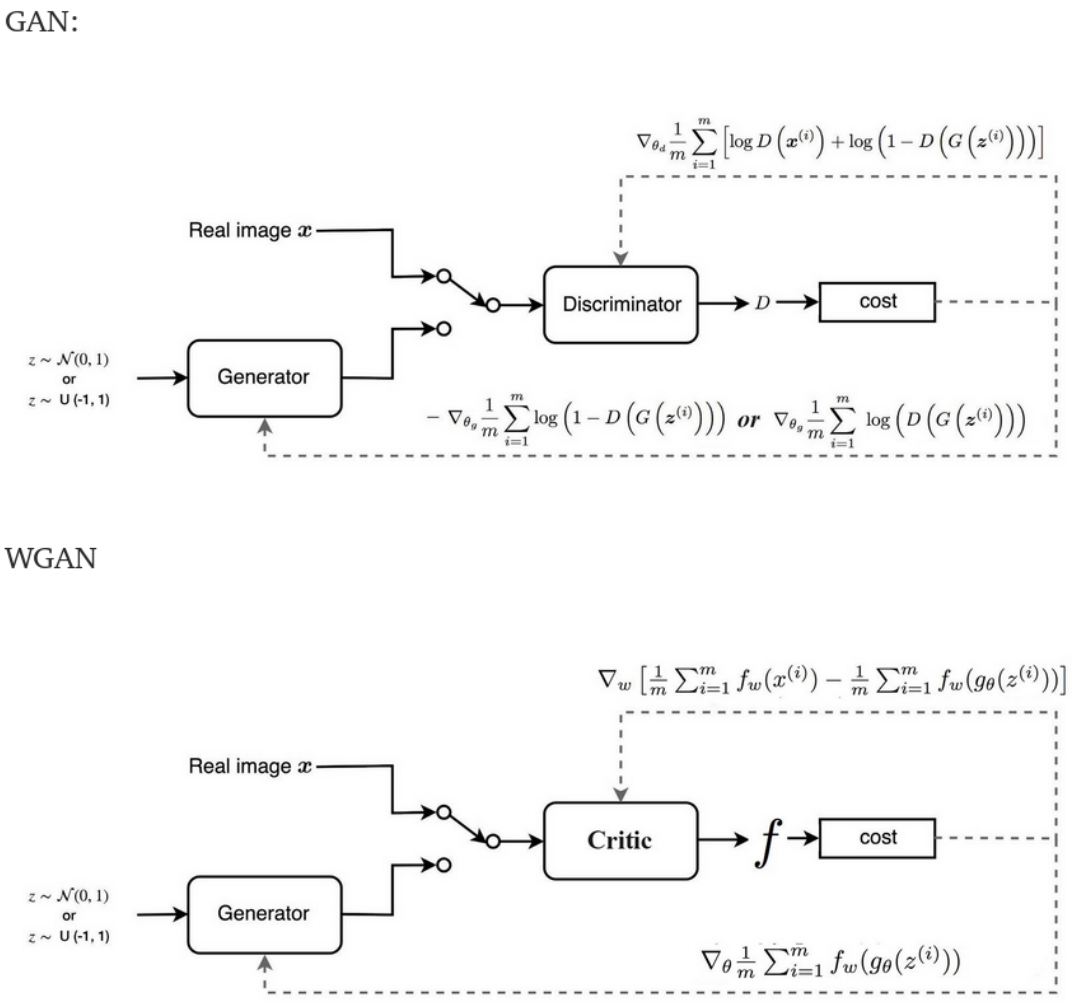


FIGURA 1.34: Comparación entre las arquitecturas GAN y WGAN [Hui18].

Algorithm 1 WGAN, our proposed algorithm. All experiments in the paper used the default values $\alpha = 0.00005$, $c = 0.01$, $m = 64$, $n_{\text{critic}} = 5$.

Require: : α, the learning rate. c, the clipping parameter. m, the batch size. n_{critic}, the number of iterations of the critic per generator iteration.
Require: : w_0, initial critic parameters. θ_0, initial generator's parameters.

while θ has not converged **do**
  for $t = 0, ..., n_{\text{critic}}$ **do**
    Sample $\{x^{(i)}\}_{i=1}^{m} \sim \mathbb{P}_r$ a batch from the real data.
    Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
    $g_w \leftarrow \nabla_w \left[\frac{1}{m}\sum_{i=1}^{m} f_w(x^{(i)}) - \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))\right]$
    $w \leftarrow w + \alpha \cdot \text{RMSProp}(w, g_w)$
    $w \leftarrow \text{clip}(w, -c, c)$
  end for
  Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
  $g_\theta \leftarrow -\nabla_\theta \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))$
  $\theta \leftarrow \theta - \alpha \cdot \text{RMSProp}(\theta, g_\theta)$
end while

FIGURA 1.35: Algoritmo WGAN [ACB17].

Dado que el algoritmo de WGAN entrena al crítico f relativamente bien antes de cada paso de actualización de pesos del generador, la función de costo es una estimación de la distancia Wasserstein.

En la Figura 1.36 se encuentra la evolución de la función de costo junto con ejemplos del generador en diferentes iteraciones. A la izquierda, la formulación GAN original utilizando la divergencia de Jensen–Shannon donde no se observa una reducción del costo a medida que la calidad de las imágenes generadas aumenta, por lo que debemos guardar las imágenes durante el entrenamiento y evaluarlas visualmente. A la derecha, por otro lado, se observa como disminuye la distancia Wasserstein a medida que aumenta la calidad del generador, lo cual es más conveniente. En ambos casos se utilizaron arquitecturas DCGAN para discriminador/crítico y generador.

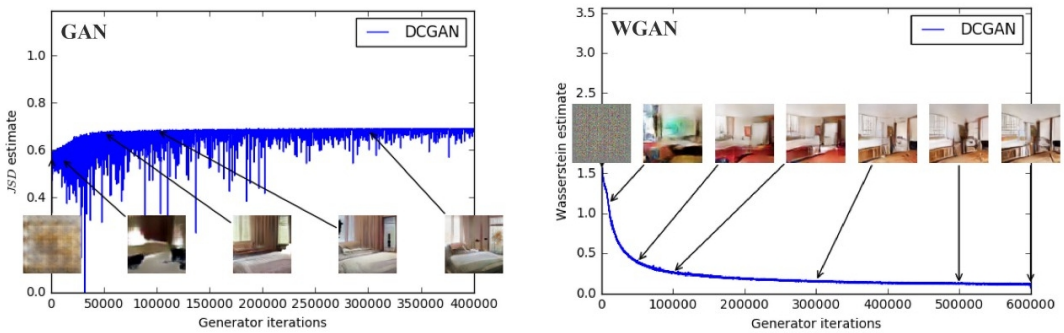


FIGURA 1.36: Correlación entre la función de costo y la calidad de las imágenes producidas por el generador para GAN (izquierda) y WGAN (derecha) [ACB17].

Por lo tanto, entre las principales ventajas de WGAN sobre GAN, podemos mencionar:

- Mayor estabilidad del proceso de optimización.
- Solución del problema de colapso de moda (mode colapse), el cual los autores afirman que no se observó en ninguno de los experimentos realizados.
- Una función de costo comprensible que se correlaciona con la convergencia del generador y su calidad de muestreo.

1.7. WGAN-GP (Gradient Penalty)

El método utilizado para cumplir con la restricción de Lipschitz necesaria para f es criticado por los propios autores del trabajo, los cuales alientan otros investigadores a mejorar este procedimiento justificando su utilización por su simpleza y buen rendimiento en los experimentos realizados.

El principal problema es que el modelo es muy sensible al hiper-parámetro c y cuando éste no es ajustado correctamente, el modelo produce imágenes de baja calidad y no converge.

En la Figura 1.37 se observa la norma de los gradientes del crítico en sus diferentes capas. Estos gradientes pasan de un escenario de desaparición del gradiente a uno de explosión del gradiente cuando el parámetro c se mueve entre 0,01 y 0,1. A la derecha se observa la distribución de los pesos utilizando *weight clipping* en la cuál los pesos son "empujados" a los extremos c y $-c$. Debajo de ésta se encuentra la distribución obtenida utilizando *Gradient Penalty* que se presentará a continuación.

Otro de los problemas identificados es que *weight clipping* funciona como un método de regularización, limitando la capacidad del modelo e impidiendo su posibilidad de aprender representaciones complejas.

En la Figura 1.38 se observan las curvas de nivel obtenidas para el crítico de WGAN entrenado hasta la optimalidad en distintos datasets de juguete, utilizando *weight clipping* (fila superior) y *Gradient Penalty* (fila inferior). Como "generador" se utiliza la distribución real de los datos sumada a ruido Gaussiano. En el primer caso, el crítico presenta mayores dificultades para capturar los detalles finos de cada distribución. Esto mejora en el segundo caso.

Una función diferenciable f es 1-Lipschtiz si y sólo si tiene gradientes con norma menor o igual a 1 en todo su dominio. Basándose en esto, WGAN-GP penaliza al modelo si la norma del gradiente se aleja de su valor de norma

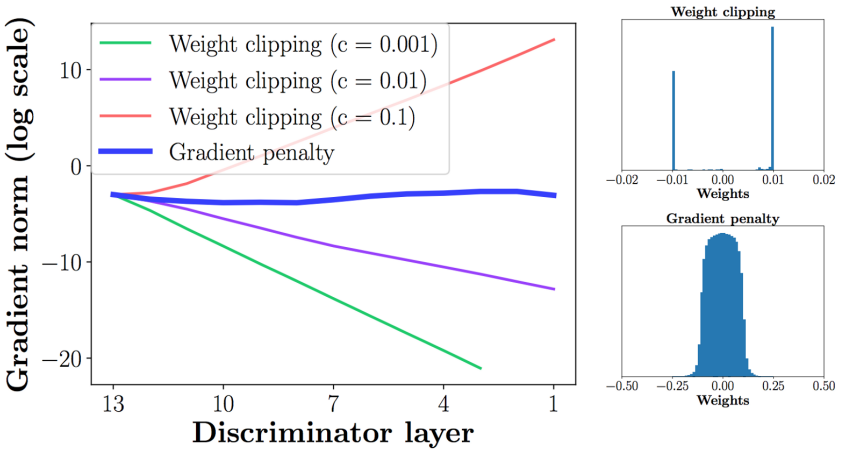


FIGURA 1.37: Norma de los gradientes del crítico en sus diferentes capas utilizando distintos valores para el parámetro c y comparado con WGAN-GP. [Gul+17b].

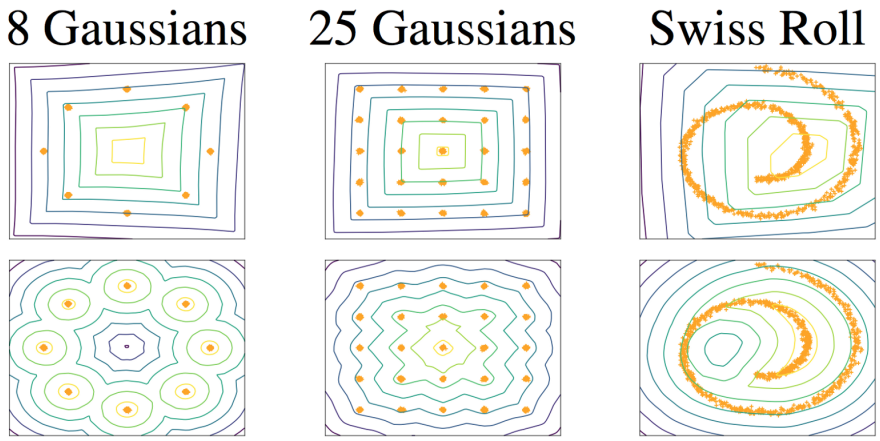


FIGURA 1.38: Curvas de nivel obtenidas para el crítico de WGAN utilizando *weight clipping* (fila superior) y *Gradient Penalty* (fila inferior) [Gul+17b].

objetivo 1, utilizando la siguiente función de costo:

$$\mathbb{E}_{\tilde{x}\sim\mathbb{P}_g}\Big[D(\tilde{x})\Big] - \mathbb{E}_{x\sim\mathbb{P}_r}\Big[D(x)\Big] + \lambda\mathbb{E}_{\hat{x}\sim\mathbb{P}_{\hat{x}}}\Big[(\|\nabla_{\hat{x}}D(\hat{x})\| - 1)^2\Big]$$

donde $\hat{x} = \epsilon x + (1-\epsilon)\tilde{x}$ con $\epsilon \sim U[0,1]$.

El algoritmo completo se puede ver en la Figura 1.39.

WGAN-GP mejora la estabilidad del entrenamiento. Como se muestra a continuación (Figura 1.40), cuando el diseño de la arquitectura del discriminador y/o generador es sub-óptimo WGAN-GP produce buenos resultados mientras que el resto de las variantes (DCGAN, LSGAN, WGAN (clipping)) fallan con algunas de ellas.

Algorithm 1 WGAN with gradient penalty. We use default values of $\lambda = 10$, $n_{\text{critic}} = 5$, $\alpha = 0.0001$, $\beta_1 = 0$, $\beta_2 = 0.9$.

Require: The gradient penalty coefficient λ, the number of critic iterations per generator iteration n_{critic}, the batch size m, Adam hyperparameters α, β_1, β_2.
Require: initial critic parameters w_0, initial generator parameters θ_0.

while θ has not converged **do**
  for $t = 1, ..., n_{\text{critic}}$ **do**
    for $i = 1, ..., m$ **do**
      Sample real data $\boldsymbol{x} \sim \mathbb{P}_r$, latent variable $\boldsymbol{z} \sim p(\boldsymbol{z})$, a random number $\epsilon \sim U[0, 1]$.
      $\tilde{\boldsymbol{x}} \leftarrow G_\theta(\boldsymbol{z})$
      $\hat{\boldsymbol{x}} \leftarrow \epsilon \boldsymbol{x} + (1 - \epsilon)\tilde{\boldsymbol{x}}$
      $L^{(i)} \leftarrow D_w(\tilde{\boldsymbol{x}}) - D_w(\boldsymbol{x}) + \lambda(\|\nabla_{\hat{\boldsymbol{x}}} D_w(\hat{\boldsymbol{x}})\|_2 - 1)^2$
    end for
    $w \leftarrow \text{Adam}(\nabla_w \frac{1}{m} \sum_{i=1}^{m} L^{(i)}, w, \alpha, \beta_1, \beta_2)$
  end for
  Sample a batch of latent variables $\{\boldsymbol{z}^{(i)}\}_{i=1}^{m} \sim p(\boldsymbol{z})$.
  $\theta \leftarrow \text{Adam}(\nabla_\theta \frac{1}{m} \sum_{i=1}^{m} -D_w(G_\theta(\boldsymbol{z})), \theta, \alpha, \beta_1, \beta_2)$
end while

FIGURA 1.39: Algoritmo WGAN-GP [Gul+17b].

FIGURA 1.40: Diferentes arquitecturas GAN entrenadas mediante diversos métodos. WGAN-GP es el único que logra un entrenamiento efectivo utilizando cualquiera de las arquitecturas [Gul+17b].

Capítulo 2

Trabajos Relacionados

Existen varios trabajos con motivaciones similares a la presente tesina. Por ejemplo, en [CB16] los autores proponen reconstruir el vector Z^* que da origen a una determinada imagen X realizando un descenso por el gradiente. Básicamente utilizan el gradiente del generador G con respecto a un vector Z inicializado al azar sobre la distribución del espacio Z, siempre que el grafo computacional de G esté disponible. En este caso muestran resultados obtenidos sobre el dataset MNIST [LC10] donde la imagen generada por el Z recuperado mantiene el estilo e identidad del dígito original. Estos resultados se pueden ver en la Figura 2.1.

En una segunda versión de este trabajo, la cual se puede ver en [CB18], los autores aplican este modelo al dataset CelebA [Liu+15] y a un dataset de zapatos [YG14]. Los resultados se observan en las Figuras 2.2a y 2.2b.

Un detalle a mencionar sobre estos trabajos es que requieren la disponibilidad del grafo computacional del generador G para el cálculo del gradiente durante la codificación de cada imagen. En nuestra propuesta sólo necesitamos una muestra grande de pares $(G(Z), Z)$ para el entrenamiento de la inversa y luego las codificaciones se realizan con esta última sin necesidad de utilizar el generador.

En el trabajo [Per+16], se consideran datasets de rostros. Los autores entrenan un par de encoders E_z y E_y de manera de obtener, a partir de una imagen X, un vector Z y un vector de atributos condicionales Y del espacio latente. Luego, utilizando una red adversaria generativa condicional [MO14] pueden reconstruir la imagen original realizando modificaciones complejas a partir de alteraciones en el vector de atributos condicionales Y. En este caso, si bien las reconstrucciones, sin modificación del vector de atributos Y, son coherentes, tienen una diferencia apreciable con la identidad original (Figura 2.3).

En Adversarially Learned Inference (ALI) [Dum+16] los autores proponen aprender un par encoder-decoder utilizando el proceso adversario de la Figura 2.4. El tercer componente del modelo es el discriminador, el cual recibe pares

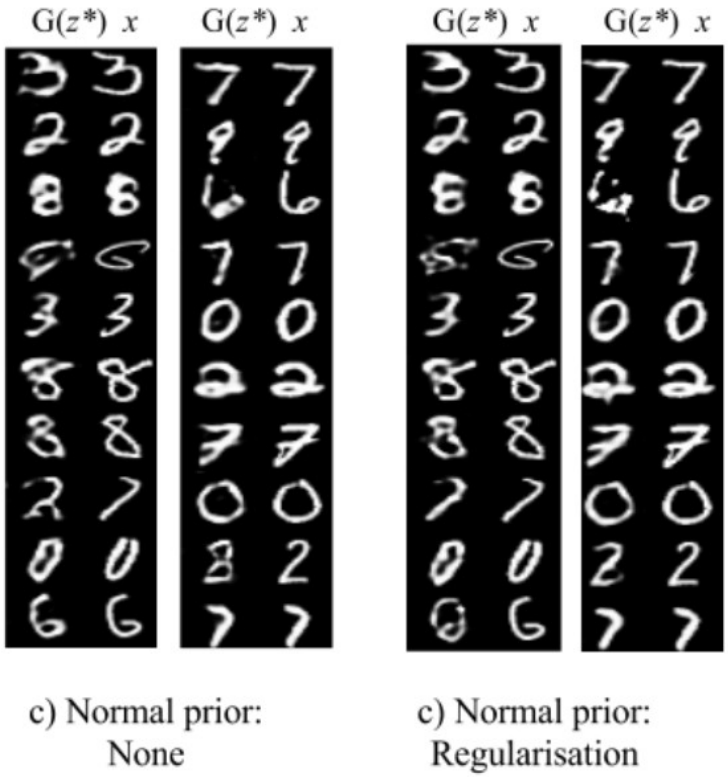


Figura 2.1: Resultados en [CB16]

de la forma (x, z) donde x es un ejemplo del dataset de interés y z es un vector del espacio latente.

La distribución conjunta del encoder es $q(x, z) = q(x)q(z|x)$, donde $q(x)$ es la distribución empírica de los datos de nuestro dataset. Paralelamente, la distribución conjunta del decoder es $p(x, z) = p(z)p(x|z)$, donde $p(z)$ es la distribución desde donde se toman las muestras del espacio latente y generalmente se trata de una distribución simple, como la normal estándar $p(z) = \mathcal{N}(0, 1)$.

El objetivo de ALI es emparejar ambas distribuciones conjuntas, lo que nos asegura la similitud de las distribuciones condicionales: $q(z|x) \approx p(z|x)$. Para esto el par encoder-decoder es entrenado para engañar al discriminador. Los resultados de este trabajo están en la Figura 2.5.

En [Lar+15] combinan un Variational Autoencoder con un modelo GAN donde el decoder del VAE se corresponde con el generador GAN, como se aprecia en la Figura 2.6. Los resultados obtenidos por este modelo se encuentran en la última fila de la Figura 2.7, la primer fila se corresponde con la entrada y las demás son comparaciones con otros modelos.

Por último, el trabajo [Mai+18] trata de una aplicación práctica para esta técnica de reconstrucción, que tiene como objetivo engañar sistemas de reconocimiento facial. La forma del ataque se describe en la Figura 2.8. No entraremos

(a) Shoe data samples, x, from a test set

(b) Reconstructed data samples, $G(z^*)$ using a GAN at resolution 128×128

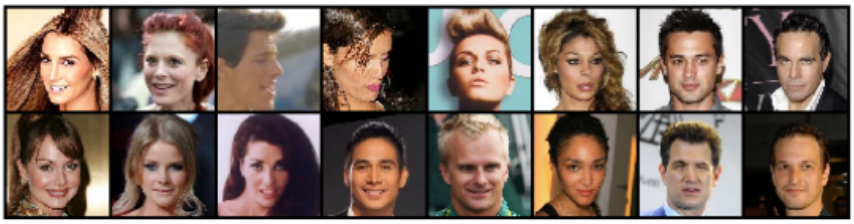
(c) Reconstructed data samples, $G(z^*)$ using a GAN at resolution 64×64

(d) Reconstructed data samples, $G(z^*)$ using a WGAN at resolution 64×64

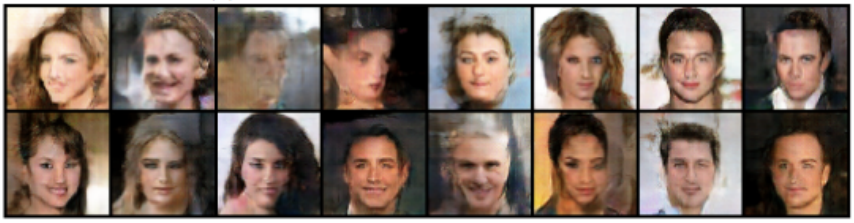
(A) Reconstrucción del dataset [YG14] de zapatos [CB18].

en detalles sobre el modelo pero se trata de un bloque de de-convolución con modificaciones. Los resultados de reconstrucción se observan en la Figura 2.9.

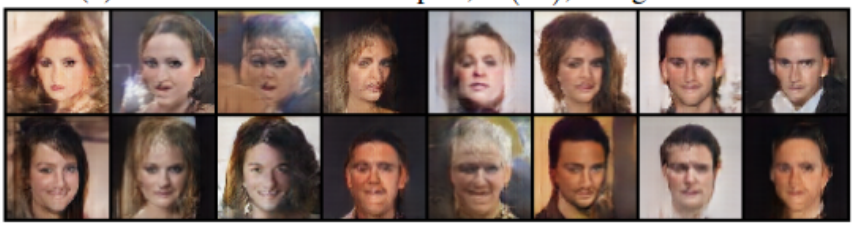
En estos trabajos hemos repasado distintos modelos propuestos con un objetivo en común: la preservación de la identidad de la imagen original. Es en este aspecto (de preservación de la identidad) donde todavía es necesario comprender y mejorar el desempeño de los pares encoder-generador, y a lo que nos abocaremos en la presente tesina.



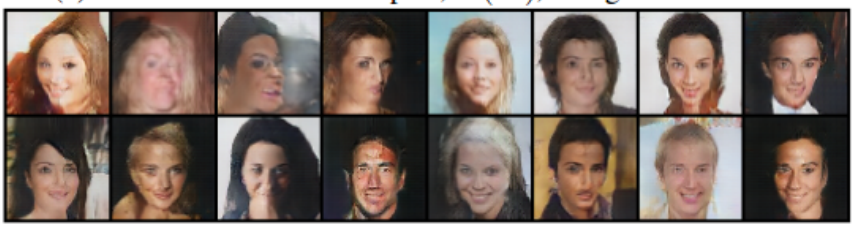
(a) CelebA faces, x, from a test set



(b) Reconstructed data samples, $G(z^*)$, using a WGAN



(c) Reconstructed data samples, $G(z^*)$, using a GAN+noise



(d) Reconstructed data samples, $G(z^*)$, using a GAN

(B) Reconstrucción del dataset [Liu+15] CelebA [CB18].

FIGURA 2.2: Resultados en [CB18]. La primer fila de imágenes contiene ejemplos del dataset de test y el resto de las filas corresponden a las reconstrucciones con distintas arquitecturas GAN.

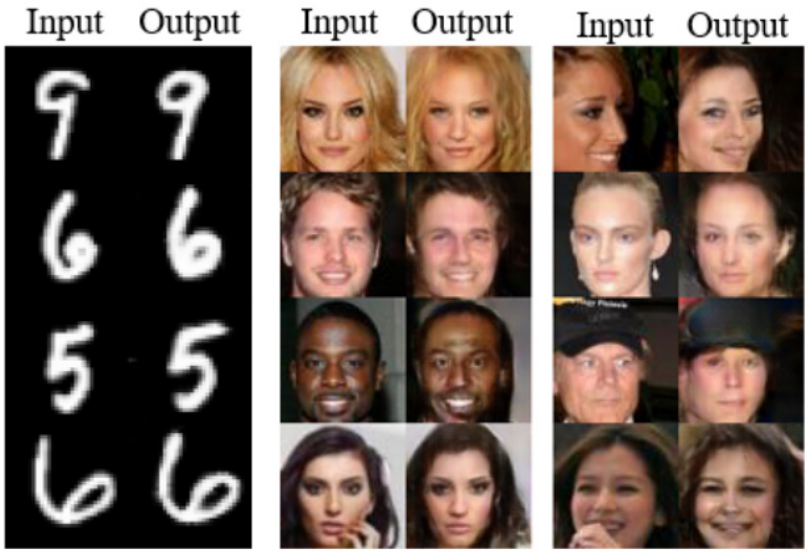


FIGURA 2.3: Resultados en [Per+16]

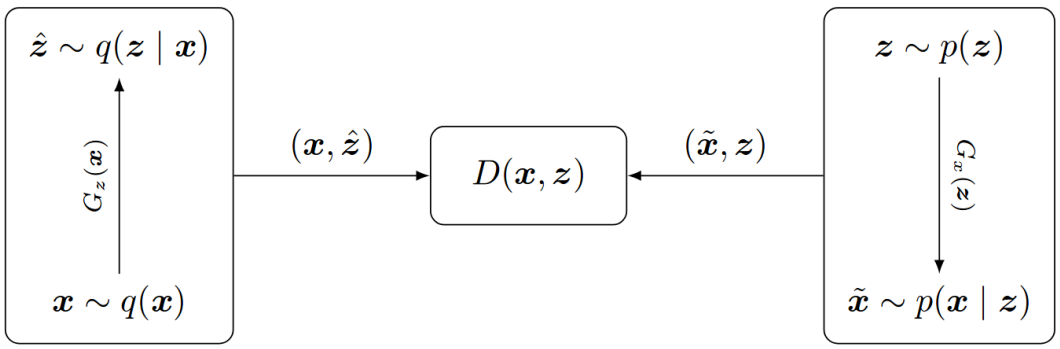


FIGURA 2.4: Modelo ALI [Dum+16].

FIGURA 2.5: Resultados en [Dum+16] sobre el dataset CelebA. Las columnas impares son las imágenes originales de validación y las columnas pares su reconstrucción. Resultados sobre los datasets SVHN, Tiny ImageNet y CIFAR10 se pueden encontrar en el trabajo original.

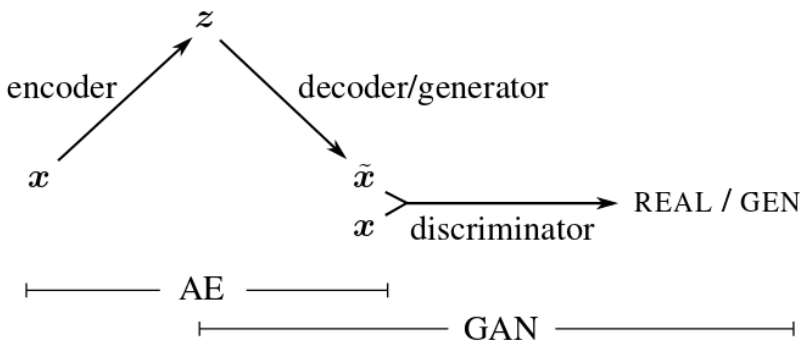


FIGURA 2.6: Modelo propuesto en [Lar+15]. El decoder del VAE se corresponde con el generador del modelo GAN.

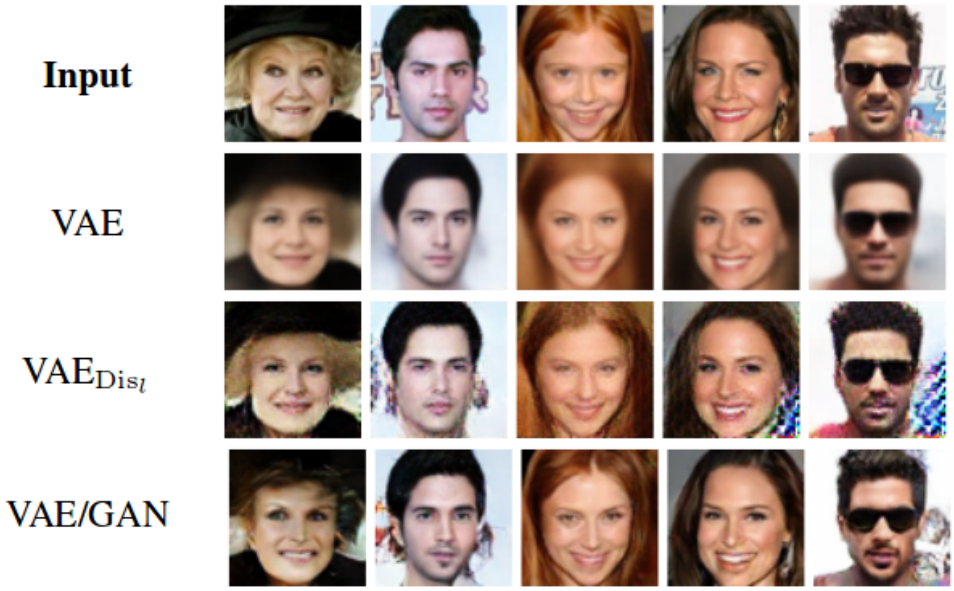


FIGURA 2.7: Resultados del trabajo [Lar+15].

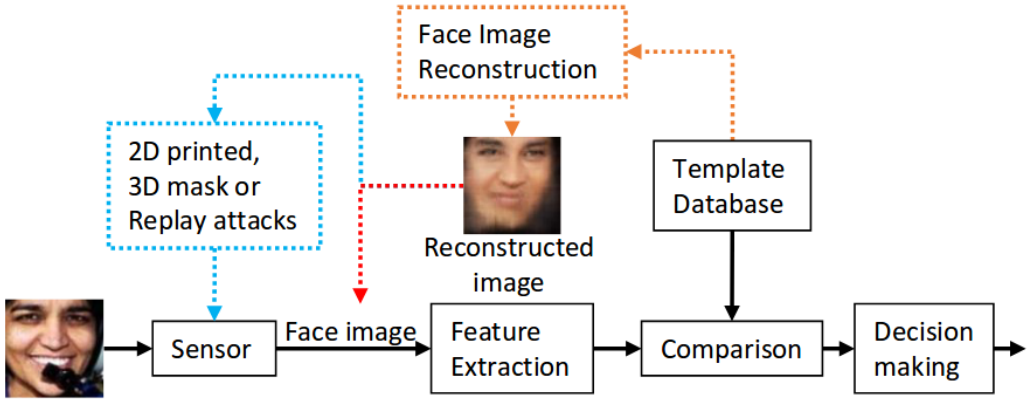


FIGURA 2.8: Ataque a sistemas de reconocimiento facial en [Mai+18]. La reconstrucción facial *(naranja)* se inyecta directamente al extractor de características *(rojo)* o bien mediante una reconstrucción 2D o 3D de la imagen *(azul)*.

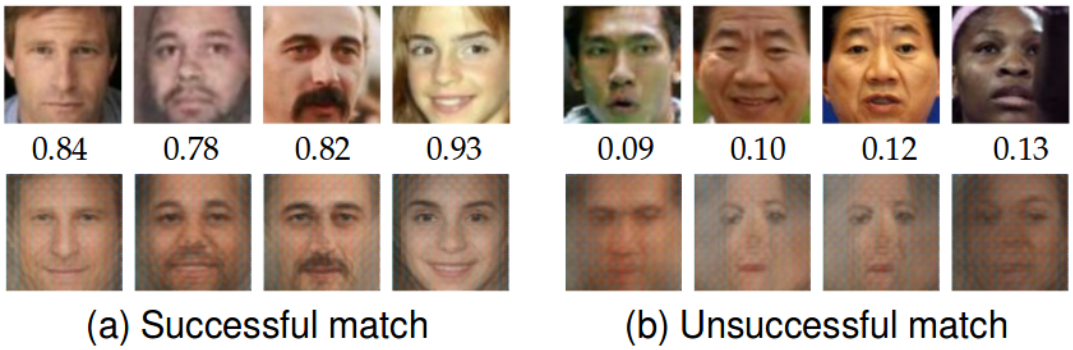


FIGURA 2.9: Resultados de reconstrucciones de rostros en [Mai+18] sobre el dataset Labeled Faces in the Wild [Hua+07]. Las imágenes superiores corresponden a las originales mientras que las inferiores a la reconstrucciones. Además, el número central es la similitud coseno entre las mismas. En el grupo *(a)* se encuentran las reconstrucciones exitosas y en el *(b)* las fallidas.

Capítulo 3

Modelo Propuesto y Experimentos

Las siguientes Subsecciones (3.1, 3.2 y 3.3) son complementarias. Para reproducir los resultados, prestar especial atención a la Subsección 3.2 que es donde se realiza la explicación de la capa reemplazada en la función inversa, épocas de entrenamiento y el optimizador utilizado con sus parámetros.

3.1. Modelo Propuesto

Para lograr nuestro objetivo de codificar imágenes de rostros nuevamente al espacio latente de vectores Z propusimos una Inversa/Encoder (I) con arquitectura similar al discriminador del modelo pero con la diferencia que removimos la última capa lineal con 1 sola neurona de salida y la reemplazamos por capas fully-connected donde la salida tiene igual cantidad de neuronas que el vector Z.

Los datos de entrenamiento para el generador (G) y discriminador (D) se proveen de la forma tradicional en un entrenamiento GAN. G recibe un vector de 128 componentes aleatorias generadas con una distribución normal estándar mientras que D recibe imágenes reales del dataset e imágenes sintéticas de la salida de G.

Una vez terminado este entrenamiento GAN, continuamos con el entrenamiento de la inversa utilizando el generador ya entrenado. En este caso los datos de entrenamiento se proveen de la forma $(G(Z), Z)$. De esta forma, tenemos infinitos datos para el entrenamiento.

La salida de esta inversa: $Z' = I(G(Z))$, es un nuevo vector de igual dimensionalidad que Z. El objetivo es que Z' capture las características principales del rostro a generar y que, por lo tanto, $G(Z')$ y $G(Z)$ sean la misma imagen.

Con respecto a la función de costo utilizada para entrenar la inversa I tenemos un costo total ($total\_cost$) compuesto por 2 sumandos: uno corresponde al error absoluto medio entre Z y Z', al cual llamaremos z_cost, mientras que

el otro se compone por el error absoluto medio entre los píxeles de la imagen $G(Z)$ y los de la imagen $G(Z')$ al cual le daremos el nombre x_cost.

La función de costo tiene como objetivo que $Z' = Z$, siendo Z la etiqueta o valor objetivo correspondiente a la imagen de entrada $G(Z)$ por lo que podemos decir que el entrenamiento de la inversa se trata de un aprendizaje supervisado sobre un dataset sintético mientras que el entrenamiento de la porción GAN tradicional es un aprendizaje no supervisado sobre el dataset real.

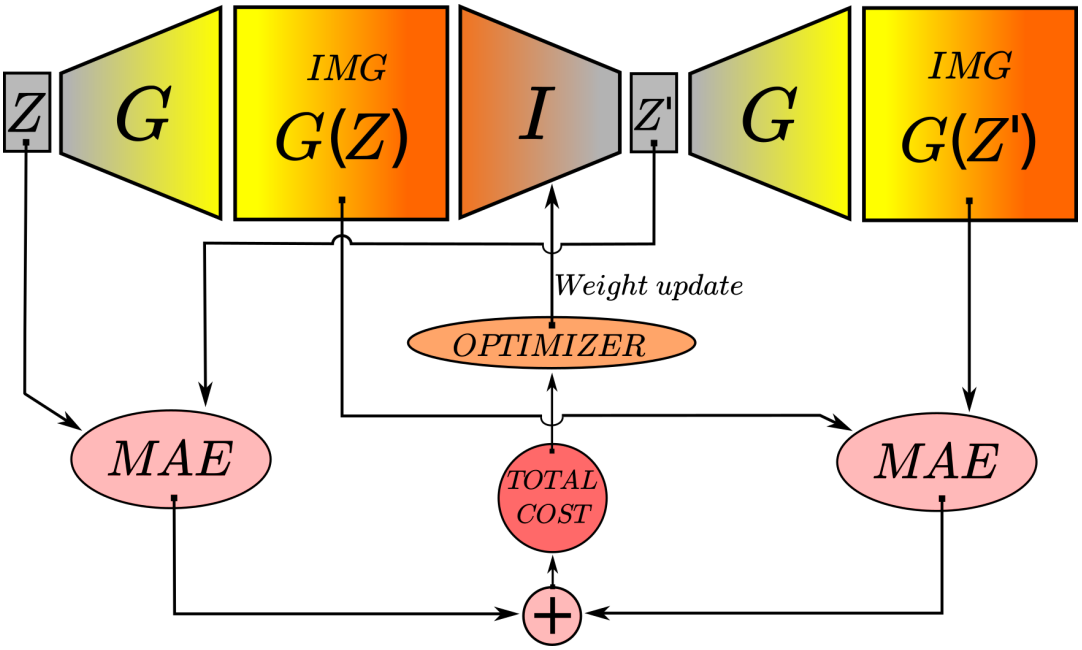


FIGURA 3.1: Arquitectura propuesta para lograr nuestro objetivo de codificar imágenes de rostros al espacio latente de vectores. Ponemos énfasis en el entrenamiento de la Inversa ya que el entrenamiento del Generador se realiza previamente de manera tradicional para un modelo GAN.

3.2. Detalles de Implementación

Se partió del modelo y código presentado en [Gul+17b] que a su vez está basado en el nuevo algoritmo Wasserstein GAN [ACB17], ambos explicados en el capítulo 1.

Este código base ofrece 2 arquitecturas: DCGAN [RMC15] y ResNet [He+16]. Tanto para el Generador (G) como para el Discriminador (D) utilizamos DCGAN por una cuestión de costo de entrenamiento pero debemos tener en cuenta que ResNet ofrece una mejora, aunque leve, de la calidad de imágenes generadas [Gul+17a].

El framework utilizado para nuestros experimentos fue TensorFlow [Goo18b], creado por Google. Éste nos ofrece implementaciones reutilizables de operaciones/estructuras utilizadas comúnmente en los algoritmos de redes neuronales, y en particular en nuestro trabajo, como multiplicación y adición de matrices, convoluciones, deconvoluciones, batch normalization, layer normalization y capas pre-armadas que utilizan las mismas. También tenemos funciones de activación como ser: lineal, relu, leaky-relu, tanh, sigmoid y más. Cuenta además con implementaciones de las funciones de costo (MSE, MAE, Log Loss, Cross Entropy, Hinge, Huber) y optimizadores (Stochastic Gradient Descent, Adam, Adagrad, Adadelta, RMSprop) más conocidos. Aunque sin duda la característica más importante que ofrece es la implementación en GPU de lo anteriormente nombrado, lo que nos permite utilizar nuestras placas de video para disminuir radicalmente los tiempos necesarios para el entrenamiento.

En experimentos preliminares utilizamos una única capa de 128 neuronas como capa terminal de la inversa I. Luego realizamos una prueba con mayor cantidad de capas/neuronas observando una pequeña mejora en el rendimiento de la misma. Por lo cual, teniendo en cuenta que el entrenamiento en GPU de este tipo de capas es muy económico, decidimos utilizar la arquitectura más compleja.

En particular, para reproducir nuestros resultados basta tomar la arquitectura del Discriminador en [Gul+17b] y reemplazar la última capa de una única neurona por tres capas fully-connected con activación leaky-relu de dimensiones 4096, 2048 y 1024 respectivamente, más la capa final ya mencionada de 128 neuronas con función de activación lineal.

Como optimizador usamos Adam [KB14] con $\eta = 1e^{-4}$, $\beta_1 = 0$ y $\beta_2 = 0{,}9$.

La longitud del entrenamiento en la primer etapa de Generador/Discriminador fue de 200.000 iteraciones dando como resultado unas 35 horas en el hardware utilizado. Por otro lado, el entrenamiento de la inversa es más largo (en cantidad de iteraciones) al caso generador/discriminador puesto que el tiempo necesario para completar una iteración es menor. En este caso se entrenó durante 350.000 iteraciones, lo que se traduce en 16 horas de entrenamiento.

3.3. Datasets

Para los distintos experimentos se utilizaron 3 datasets.

Uno de ellos es el reconocido dataset CelebA [Liu+15] utilizado como *benchmark* en innumerable cantidad de trabajos del área. Contiene 10.177 identidades dando un total de 200.000 imágenes de las mismas.

El dataset Humanae se generó a partir de la descarga y posterior procesamiento de todas las imágenes del trabajo "Photography by Angélica Dass" [Das16]. Éste cuenta con 3.042 imágenes de retratos homogéneos en estilo, posición y fondo igual al tono de piel.

Por último, el dataset de Leonardo DiCaprio se compone de 147.464 ejemplos y fue construido por nosotros mismos utilizando el algoritmo que presentaremos a continuación. Un detalle a tener en cuenta para este último dataset es que si bien se compone de una única persona, tiene una altísima variabilidad de posición, expresión, iluminación y maquillaje. Además, en este dataset no hemos normalizado la posición de ojos y boca entre todas las imágenes que lo componen, característica que sí está presente en los otros dos datasets. Esto fue buscado adrede para luego observar si se logra mantener estas propiedades al obtener las imágenes regeneradas.

En la Figura 3.2 se observa un subconjunto de las imágenes de los distintos datasets con los cuales fueron entrenadas las diferentes redes neuronales.

Estos datasets fueron preprocesados con un algoritmo de detección, centrado y recorte de rostros [Zan17] para lograr la mayor homogeneidad posible en los datos. Dicho algoritmo fue desarrollado por quién escribe, bajo la dirección del Dr. Lucas Uzal, durante mi práctica profesional en el grupo de Aprendizaje Automatizado y Aplicaciones [CIF18]. Se dará una breve descripción del algoritmo en los siguientes párrafos.

Se utilizó un algoritmo de detección HOG (Histogram of Oriented Gradients) para detectar rostros, junto con un algoritmo de localización de puntos de referencia en dichos rostros. El recorte se centra utilizando el punto entre ojos, con un tamaño proporcional a la distancia entre este punto y la parte inferior de la nariz. Para suavizar la zona y el tamaño de recorte entre frames consecutivos, se utilizan splines ajustadas sobre distintos valores de las detecciones exitosas.

Además, el algoritmo de recorte está optimizado para detectar secuencias de rostros en videos: en caso de haber una secuencia de frames $f_i, f_{i+1}, ..., f_j$ donde se detecta el rostro en f_i y f_j pero no en f_k para $k = i+1, ..., j-1$, se interpolan los rostros no detectados utilizando las splines para tratar de recuperar la mayor cantidad posible de información.

Por último, es posible establecer una identidad de interés para separar las caras de la misma, como se hizo en el caso de Leonardo DiCaprio para el cual generamos nuestro propio dataset extrayendo los rostros de diferentes películas.

(A) Dataset DiCaprio.

(B) Dataset CelebA.

(C) Dataset Humanae.

FIGURA 3.2: Imágenes reales de los 3 datasets. Todos fueron preprocesados con el algoritmo descripto. Tener en cuenta que CelebA viene ya procesado por algoritmo de alineación de caras que deliberadamente no aplicamos en DiCaprio.

Capítulo 4

Resultados

4.1. Imágenes Sintéticas

En la Figura 4.1 se muestran las imágenes sintéticas creadas por el generador de cada uno de los modelos entrenados. De aquí en adelante, cada vez que nombramos al *"Modelo X"* nos referimos al modelo propuesto ya entrenado sobre el dataset *X*.

Debido a que la cantidad de iteraciones de entrenamiento es fija tenemos una variación en la cantidad de épocas debido a las diferencias de tamaño de los datasets.

En el caso de DiCaprio tenemos 147.464 ejemplos de entrenamiento, lo que significa 87 épocas de entrenamiento dado un tamaño de batch de 64 y 200.000 iteraciones. Para CelebA disponemos de 177.350 ejemplos, es decir, 72 épocas. Mientras que para Humanae sólo hay 3.042 imágenes, dando como resultado 4.208 épocas.

Los 3 modelos logran generar rostros de su dominio con una calidad aceptable. Sin embargo, como se puede observar en la Figura 4.1c es notable la menor calidad del generador de Humanae debido al pequeño tamaño del dataset, comparados con los rostros de muy buena calidad que se pueden observar en las Figuras 4.1a y 4.1b, de DiCaprio y CelebA respectivamente.

Los resultados obtenidos son los esperados para este experimento con la salvedad de haber deseado una mayor calidad para el modelo Humanae.

(A) Dataset DiCaprio: Imágenes sintéticas.

(B) Dataset CelebA: Imágenes sintéticas.

(C) Dataset Humanae: Imágenes sintéticas.

FIGURA 4.1: Imágenes sintéticas de los 3 datasets.

4.2. Entrenamiento de la Inversa/Encoder

4.2.1. Evolución de la función de costo

El resultado esperado es una función de costo decreciente al transcurrir las iteraciones de entrenamiento.

En la Figura 4.2 podemos observar la evolución de las funciones de costo. Como se explicó, la función de costo total está compuesta por los sumandos *z_ cost* y *x_ cost*. Tenemos dos escalas para el *eje y*, la izquierda corresponde a *z_ cost* y a *total_ cost*, mientras que la derecha a *x_ cost* en color gris, esta convención de color también se aplica en las curvas, siendo más claras las que corresponden a este último costo.

El costo *x_ cost* entra en un régimen de progreso lento hacia su valor final en la iteración número 75.000, a partir de la cuál sólo sigue disminuyendo *z_ cost*.

4.2.2. Evaluación de la Inversa/Encoder

El error absoluto medio entre 2 vectores de 128 componentes tomados al azar de una distribución normal estándar es de aproximadamente 144. Mientras que, con la inversa entrenada, este error calculado sobre los vectores originales que se proveen como entrada al generador y los vectores resultantes del proceso generación y encoding resulta de entre 60 y 90, según el dataset.

Estos valores no parecerían ser óptimos en términos cuantitativos, sin embargo, al analizar las imágenes obtenidas que se pueden ver en la Figura 4.3, los resultados son muy prometedores.

El modelo logra capturar las características de la imagen como: orientación, sexo, iluminación, sonrisa, ojos, color de pelo, color de piel, vello facial, maquillaje, etc. a través de la inversa y reproducirlas al reconstruirlas con el generador, incluso en imágenes de test que nunca vio, tal como esperábamos.

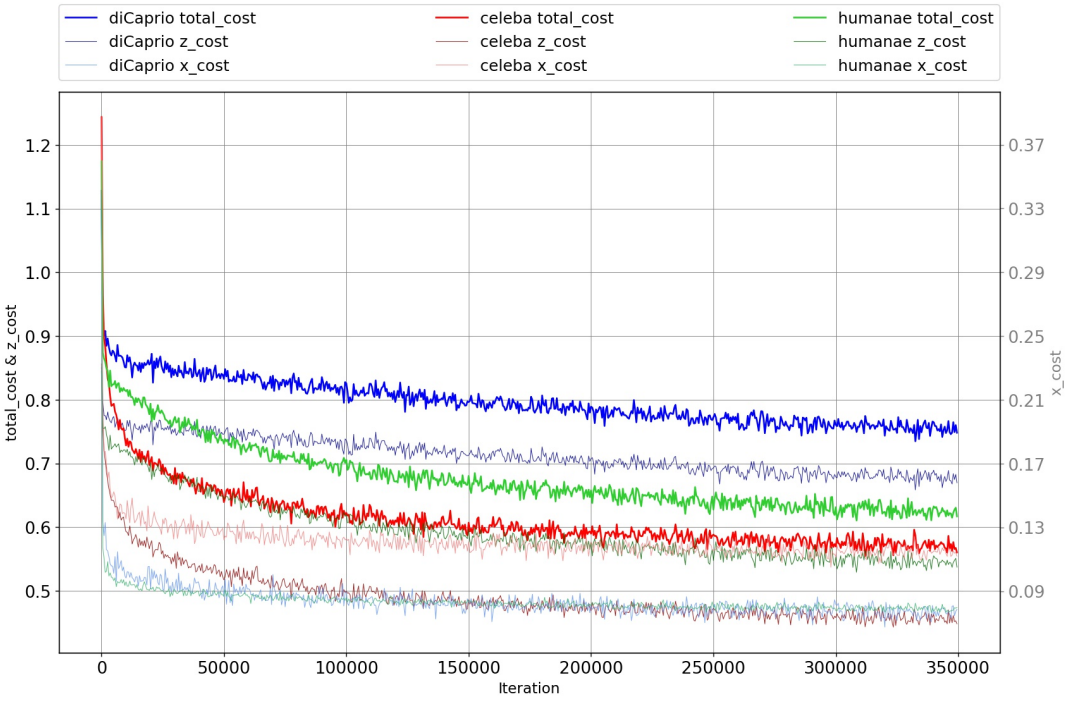


FIGURA 4.2: Evolución de costos en el entrenamiento de la inversa.

- *x_ cost*: está dado por el error absoluto medio entre los píxeles de la imagen de entrada y la imagen reconstruida al pasar por la inversa y generador.
- *z_ cost*: está dado por el error absoluto medio entre el vector Z que da origen a la imagen de entrada y el vector Z' devuelto por la inversa para dicha imagen.
- *total_ cost*: costo total dado por la suma de los 2 anteriores.

(A) Dataset DiCaprio: Imágenes reconstruidas.

(B) Dataset CelebA: Imágenes reconstruidas.

(C) Dataset Humanae: Imágenes reconstruidas.

FIGURA 4.3: Reconstrucción de imágenes reales tanto de test como de train.

4.3. Exploración del espacio latente

4.3.1. Norma de los vectores al codificar imágenes

Otro experimento realizado con el objetivo de comprender mejor los 2 espacios de interés (espacio euclídeo n-dimensional y espacio de imágenes de cada dataset) se encuentra resumido en la Figura 4.4. Calculamos la norma euclídea de los vectores Z' obtenidos al aplicar los diferentes encoders entrenados, a 1000 imágenes de cada uno de los datasets y comparamos con la norma de 1000 vectores generados al azar con la distribución ya nombrada.

Entrenamos el modelo hasta la convergencia y aún así, los vectores devueltos por la inversa al aplicarla sobre un conjunto de imágenes no tienen una distribución para su norma L2 coincidente, en la media y desvío estándar, con la distribución de la norma de los vectores de entrenamiento (recordar que la inversa es entrenada con ejemplos de la forma $(G(z), z)$ donde z tiene distribución normal estándar).

Este fenómeno no sería el resultado que esperábamos teniendo en cuenta que la calidad de reconstrucción de las imágenes es buena.

No pudimos obtener una explicación convincente del por qué las normas decaen para CelebA y tienden a aumentar para DiCaprio.

Otro fenómeno interesante que se puede observar en estos resultados es la mayor varianza para la norma de los encodings de las imágenes de los dataset de DiCaprio y Huamanae, comparada con la del dataset CelebA.

4.3.2. Desplazamiento dentro del espacio latente

A continuación describimos otra experiencia cuyo objetivo es el mismo que se mencionó anteriormente. Dado un Z_0 generado de forma aleatoria con distribución normal estándar, generamos una imagen sintética $S_1 = G(Z_0)$. Al pasar dicha imagen como entrada a la red Inversa obtenemos un nuevo Z', que al atravesar el Generador devuelve una imagen $S_2 = G(Z')$.
Definimos entonces el vector diferencia $D = Z' - Z_0$. Utilizando este vector hacemos la exploración del espacio latente de la siguiente forma:

1. Por un lado, simplemente sumamos o restamos el vector D al vector Z' y obtenemos un $Z_n = Z' + n \times D$, con $n \in [-6, -1] \cup [1, 6]$

2. Por el otro, obtenemos Z'' de forma similar pero en vez de utilizar el vector diferencia D como delta, se utiliza un vector D_{rnd}, con norma-2 igual a la norma-2 de D, definido $D_{rnd} = \frac{Z_{rnd}}{\|Z_{rnd}\|} \times \|D\|$, donde Z_{rnd} es otro vector aleatorio con distribución normal estándar.

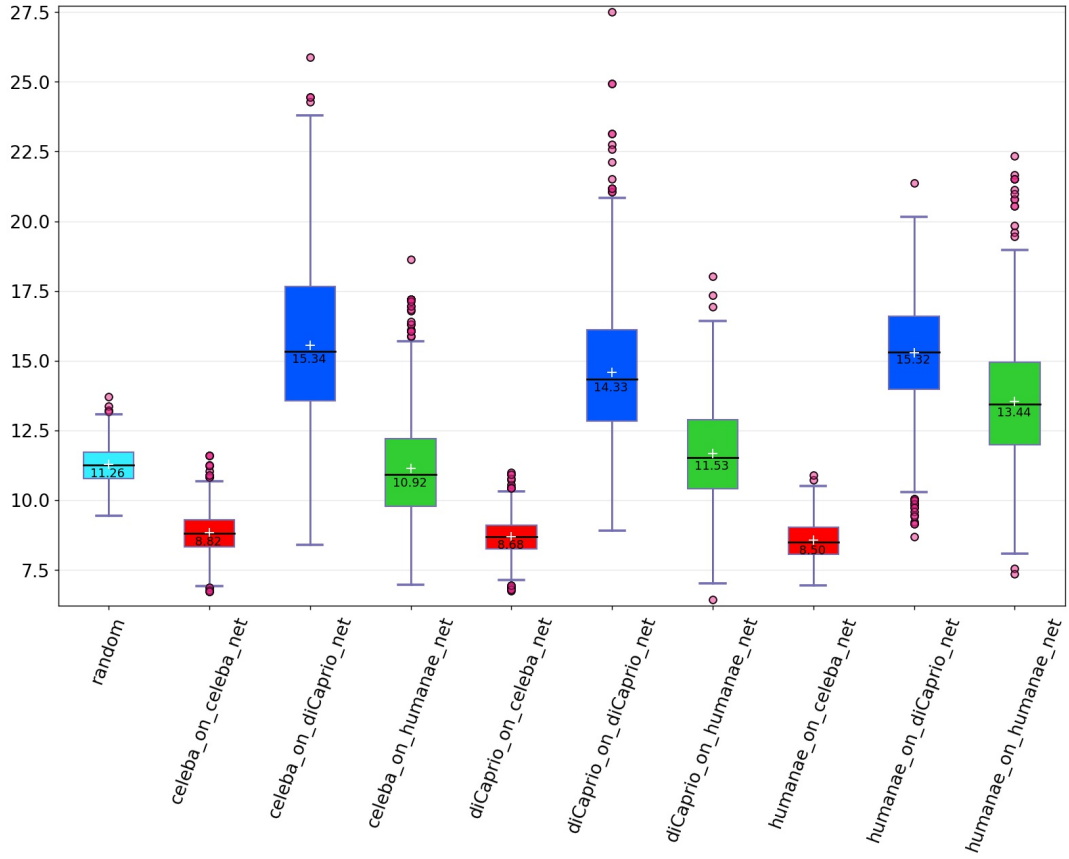


FIGURA 4.4: Boxplots de la norma de los vectores Z' dados por la inversión de imágenes de los distintos datasets utilizando los 3 modelos y generados de forma aleatoria con distribución normal estándar. En el *eje x* se encuentran los diferentes experimentos nombrados de la forma $D\_on\_M\_net$ donde D es el dataset que se dio como entrada al modelo M. En el *eje y* tenemos el valor de la norma euclídea. Los distintos boxplots están coloreados según el modelo M que se está evaluando. La mediana está señalada por la línea negra horizontal en cada uno de los boxplot y la media se encuentra señalada con el símbolo "+" en color blanco. Los outliers, por otro lado, se representan con los círculos rosados.

Utilizando estos vectores volvemos a generar imágenes sintéticas mediante G. Los resultados se encuentran en la Figura 4.5. En la zona central se observan las imágenes S_1 y S_2. A la derecha de S_2 las generadas sumando D y a la izquierda de S_1 los resultados de la resta.

Observar que por cada Z_0 inicial tenemos 2 filas similares, la primera se genera mediante el proceso descripto en el punto 1, mientras que la segunda se genera con el método 2. Se puede observar cómo las imágenes que se obtienen a través de la dirección del vector D son más similares a las originales que las que se obtienen en una dirección aleatoria.

(A) Dataset DiCaprio: Exploración del espacio latente.

(B) Dataset CelebA: Exploración del espacio latente.

(C) Dataset Humanae: Exploración del espacio latente.

FIGURA 4.5: Muestra de un desplazamiento sumando deltas en la dirección del vector diferencia D y en dirección aleatoria.

4.3.3. Generación usando imágenes y modelos cruzados

En la Figura 4.6 podemos ver los resultados de utilizar los distintos modelos con imágenes que no corresponden al dominio del dataset de entrenamiento, sino a los otros 2 disponibles en este trabajo.

Los identificadores a la izquierda de la figura corresponden al origen de los datos de entrada mientras que los ubicados en la parte superior corresponden al modelo utilizado para la inversión/generación.

En los resultados podemos observar:

- El modelo de CelebA es el que ofrece la mejor calidad de reconstrucción al utilizar imágenes de otros datasets debido a ser el que mayor variabilidad de identidades posee.
- Como es de esperar, los modelos no logran reproducir posiciones de rostros nunca vistas en los datos de entrenamiento del dominio propio. Esto es particularmente observable al utilizar Humanae o en el séptimo ejemplo de DiCaprio a CelebA (el cual posee una inclinación de cabeza nunca vista en los datos de CelebA debido a que el dataset es alineado) y es una limitación conocida de los modelos basados en aprendizaje automático.
- El modelo Humanae entiende la iluminación como el tono de piel, debido a que los datos del dataset original cumplen esta propiedad donde el fondo del rostro es de dicho tono.
- El modelo de DiCaprio recupera razonablemente bien los efectos de iluminación de los otros dominios pero sólo reconstruye imágenes convincentes en algunos casos, con mayor probabilidad cuando el origen respeta la iluminación y posición de rostro vistas por el modelo.

4.3.4. Interpolación y desplazamiento en el espacio latente

Siguiendo la idea de Radford [RMC15] realizamos una interpolación y desplazamiento en el espacio latente, con el valor agregado de poder seleccionar las 2 imágenes sobre las cuales trabajar, codificarlas y obtener sus vectores de representación dentro del espacio latente y no depender de imágenes aleatorias. Comentaremos más sobre esto y una posible aplicación práctica más adelante.

La Figura 4.7 muestra este procedimiento inspirado por los resultados de [Whi16]. Las 2 imágenes reales se observan a ambos lados de la figura, luego se codifican utilizando la red Inversa y se realiza una suma pesada de los 2 vectores resultantes. Las imágenes inmediatamente adyacentes a las reales son la regeneración del vector obtenido, mientras que las imágenes centrales son el resultado de la suma de los vectores de forma ponderada.

Observar también que hay 2 filas por cada par de imágenes, la primer fila es el resultado de la suma de los vectores, mientras que la segunda fila es normalizando dicho vector resultante de manera que su norma euclídea sea igual

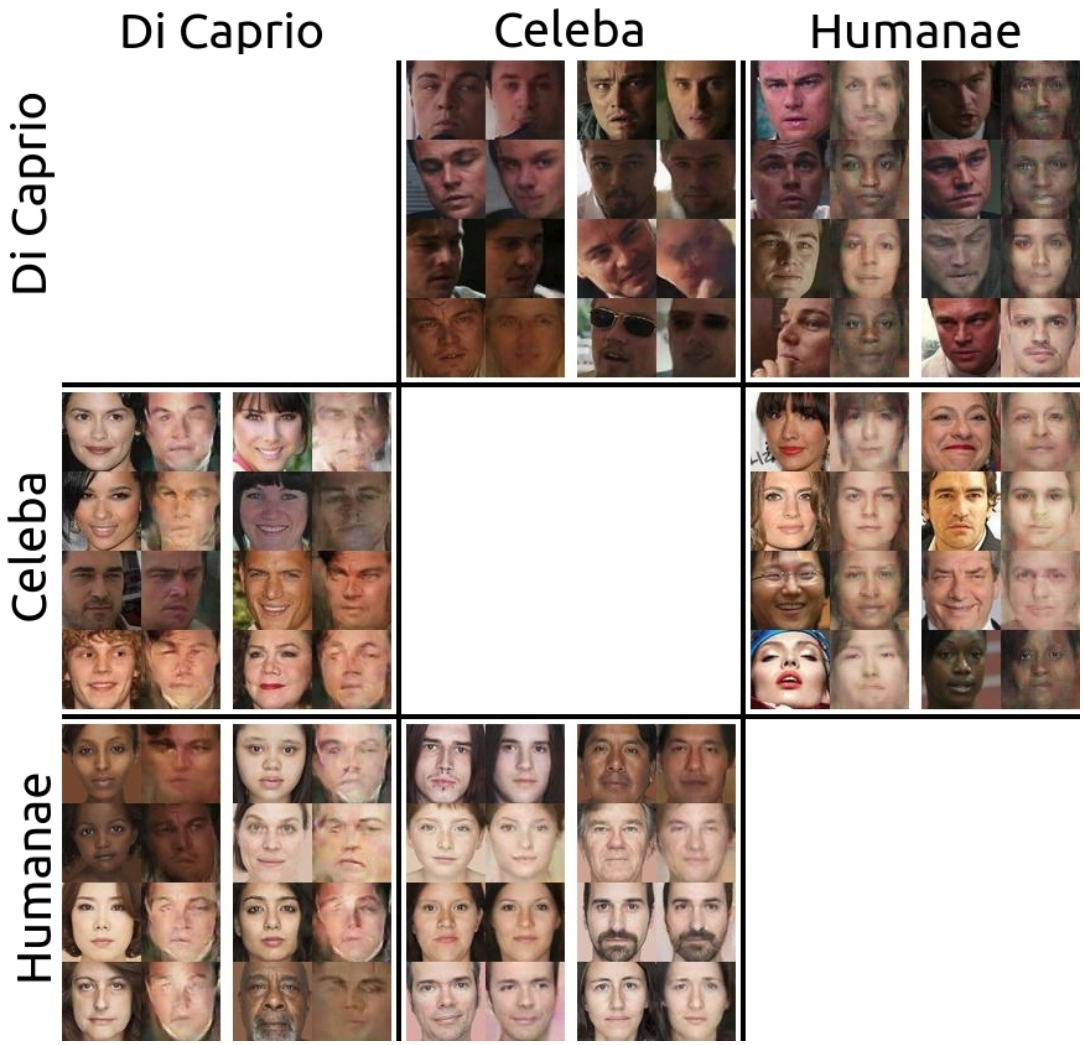


FIGURA 4.6: Imágenes de los distintos datasets, codificadas y regeneradas por las demás redes.

al valor de la mediana de la norma calculada sobre mil imágenes del dataset apropiado. Se observa una mejora de calidad mediante esta normalización.

(A) Dataset DiCaprio: interpolación entre 2 imágenes reales.

(B) Dataset CelebA: interpolación entre 2 imágenes reales.

(C) Dataset Humanae: interpolación entre 2 imágenes reales.

FIGURA 4.7: Interpolación entre 2 imágenes reales. La segunda fila de cada par muestra el mismo procedimiento agregando una normalización a los vectores resultantes dependiendo del dataset.

4.4. Dinámica de generación

Motivados por el hecho de que el encoder devuelve vectores diferentes a pesar de que las imágenes reconstruidas son notablemente similares, surge la pregunta de qué sucede al iterar el procedimiento de codificación/reconstrucción, constituyendo así un sistema dinámico.

Este experimento es de carácter exploratorio. La dinámica es el resultado de la interacción entre el generador, elemento al cual queremos caracterizar, y su inversa que es el elemento que nosotros estamos introduciendo.

Para poder observar la evolución del sistema, construimos un video con la secuencia del conjunto de imágenes generadas en cada iteración (accesible en [Zan19c]). Una captura de 9 imágenes en diferentes instantes de esta evolución se puede analizar en la Figura 4.8, con el número de iteración en la esquina superior izquierda.

El resultado ideal hubiera sido que los modelos converjan a la identidad original de la iteración cero. Sin embargo, los resultados observados sugieren que el sistema se comporta de forma caótica con ciertos rostros particulares que constituyen los atractores del mismo. Esto es un resultado esperable desde el punto de vista teórico. Ya que se espera que una dinámica en un espacio de alta dimensionalidad presente más comúnmente un comportamiento caótico.

En la Figura 4.9 podemos ver como la distancia de Manhattan entre cada vector y su siguiente converge a cero al avanzar las iteraciones, reforzando así la hipótesis de la presencia de atractores en el sistema.

Por otro lado, en la Figura 4.10 graficamos la distancia coseno entre los vectores en las iteraciones 0 y n, con $n = 0, ..., 5000$. Se puede apreciar en la Figura 4.10a como rápidamente (10 iteraciones) los vectores se alejan del inicial dando una distancia coseno con valor cercano a 1. Además, cuando éstos confluyen a los atractores se observan tres fenómenos principales: vectores constantes que no varían, vectores con un ciclo de variación determinado y vectores con un comportamiento no periódico presumiblemente caótico, como se ve en la Figura 4.10b.

Interpretando estos resultados desde el punto de vista del espacio latente, partimos de un conjunto de imágenes distintas que se mapearían idealmente a una hiper-esfera en nuestro manifold de alta dimensionalidad. Sin embargo, al avanzar el proceso dinámico, todo este volumen se restringe a unos pocos grados de libertad presentando en muchos casos imágenes estáticas y otras con comportamiento periódico o pseudo-periódico.

En todas estas figuras el *eje x* identifica el número de iteración mientras que el *eje y* muestra el valor de la distancia en cuestión.

Estos mismos resultados se replican con los otros dos datasets de este trabajo, con la diferencia que el colapso a los atractores sucede mucho antes para CelebA y muchísimo después para Humanae. Los resultados en video se muestran en: [Zan19d] y [Zan19e].

Dinámica de generación. Paso 0 (imágenes reales) | Dinámica de generación. Paso 1 | Dinámica de generación. Paso 50

Dinámica de generación. Paso 75 | Dinámica de generación. Paso 100 | Dinámica de generación. Paso 200

Dinámica de generación. Paso 350 | Dinámica de generación. Paso 4900 | Dinámica de generación. Paso 4999

FIGURA 4.8: Diferentes instantes en la dinámica de generación.

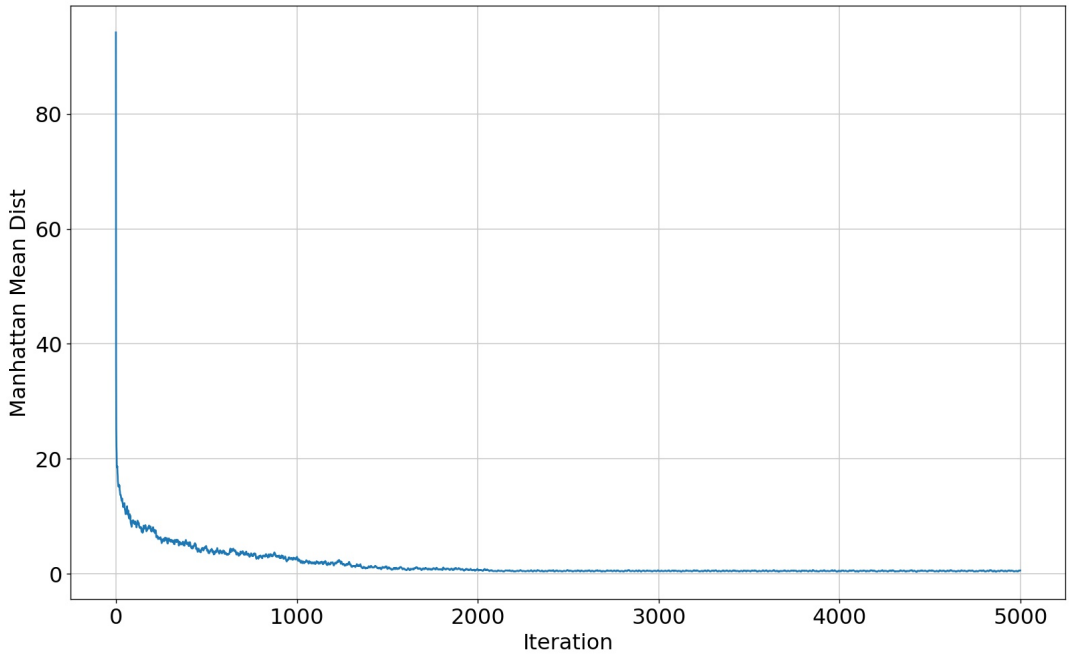
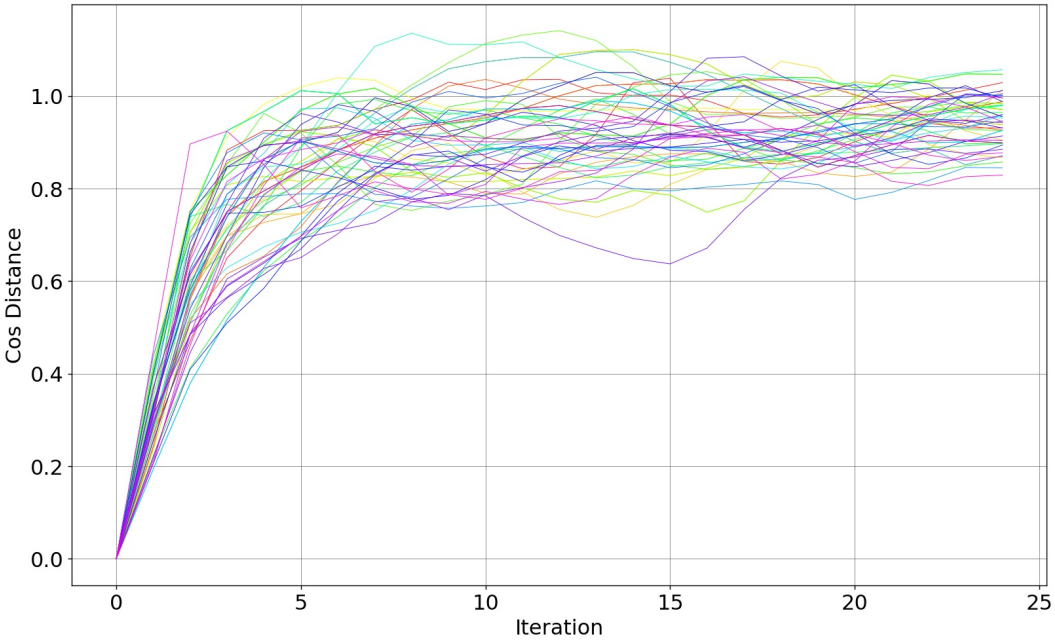
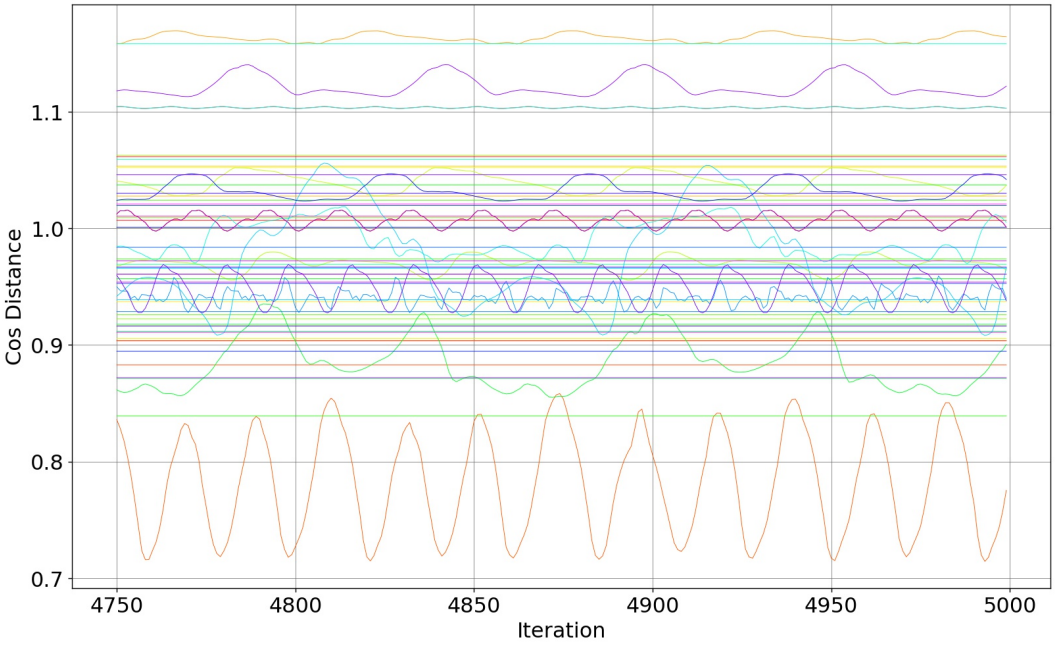


FIGURA 4.9: Distancia de Manhattan entre cada vector Z y su siguiente en la dinámica de generación.



(A) Iteraciones 0 a 25.



(B) Iteraciones 4750 a 5000.

FIGURA 4.10: Distancia coseno entre el vector Z en la iteración $\boldsymbol{n}$ y el primer vector Z de la dinámica.

4.5. Codificación y reconstrucción de videos propios con identidades familiares

Viendo que el modelo logra capturar y reproducir las diferentes características de la imagen procedimos a utilizar imágenes de videos capturados en casa y procesados con el mismo algoritmo de recorte como entrada a los diferentes modelos.

En el caso del dataset de DiCaprio esperábamos suplantar su identidad imitando expresiones y posiciones de la imagen de entrada.

En el caso de CelebA, la idea también era mantener las características de la imagen pero en este caso al tratarse de un dataset con miles de identidades, ver si lograba reproducir la identidad original y si devolvía siempre la misma o existía variación dependiendo de las diferentes condiciones.

Para Humanae, debido a la composición del dataset, no teníamos pretensiones de que reproduzca atributos nunca vistos pero igualmente quisimos analizar sus resultados.

Los resultados se pueden examinar en el siguiente video: [Zan19a]. En el caso del dataset de DiCaprio, si bien no todas las imágenes resultantes muestran una identidad perfectamente definida del actor, logra mantener las características principales de expresión/posición. Por otro lado, para CelebA, el modelo logra devolver una identidad similar a la original, variando levemente la misma ante las distintas entradas pero manteniendo siempre rasgos semejantes. Por último, en los resultados del dataset Humanae no se logra reproducir ni expresiones ni posición debido a las limitaciones del dataset tal como se esperaba.

También realizamos una selección de imágenes disponibles en la Figura 4.11. Existe una columna por cada dataset, y dos filas con dos identidades de entrada diferentes.

Por último, repetimos estos experimentos con un video obtenido de YouTube donde hay actores dando un discurso, la reconstrucción se puede ver en [Zan19b].

FIGURA 4.11: Codificación y generación con personas reales en los distintos datasets.

4.6. Análisis cuantitativo de los resultados del proceso de reconstrucción

Para dar un cierre a los resultados del proceso de inversión/reconstrucción se realizó un experimento para obtener medidas cuantitativas sobre la calidad de reconstrucción de nuestras imágenes al pasar por la inversa.

Estos resultados se encuentran en la Figura 4.12, cada uno de los 3 datasets sobre los que se realizaron las mediciones tiene su color. A la izquierda, la similitud coseno entre pares de caras aleatorias (reales) tomadas del conjunto de test de cada dataset. A la derecha, entre cada cara y su reconstrucción.

El único preprocesamiento realizado a las imágenes antes de calcular la similitud coseno fue restar la media del dataset en cuestión.

Como se aprecia en la figura hay caras más y menos similares dentro de los pares tomados aleatoriamente donde la gran mayoría no supera una similitud coseno de 0,35 mientras que las reconstrucciones tienen, en promedio, una similitud coseno de 0,9 con la imagen original.

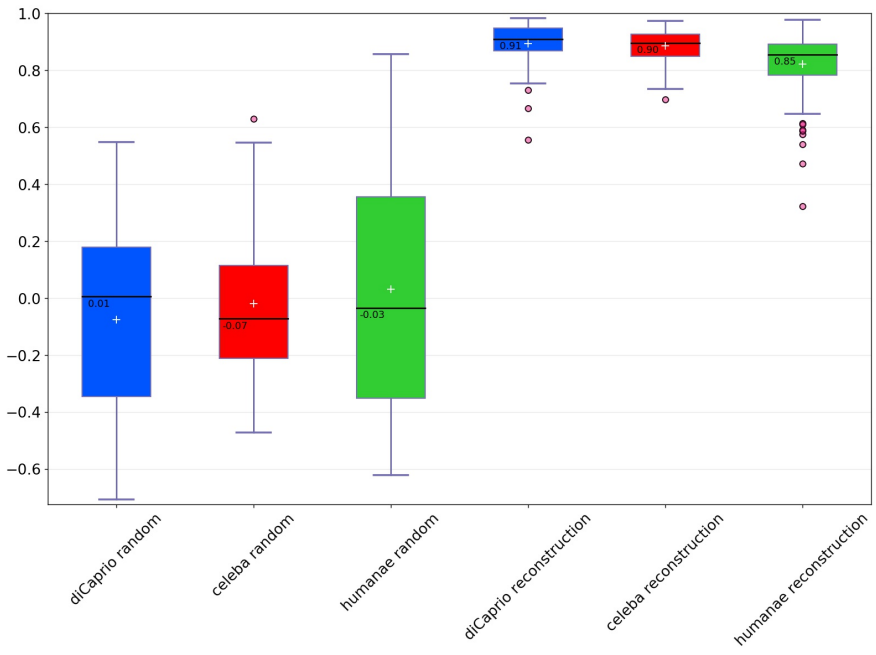


FIGURA 4.12: Análisis cuantitativo de los resultados del proceso de reconstrucción.

4.7. Posibles aplicaciones prácticas

A pesar de que esta tesina quede mejor enmarcada como un trabajo de investigación, hemos identificado algunas aplicaciones prácticas que podrían ser de utilidad:

- Generación y animación de rostros sintéticos de una determinada persona.
- Representación visual de chatbots.
- En términos de seguridad informática, suplantación de identidad en sistemas de validación por reconocimiento facial en video.
- Generación de identikits manipulables, mediante la combinación de diferentes personas que posean las características deseadas.

Capítulo 5

Conclusiones y Trabajos Futuros

5.1. Conclusiones

Un modelo GAN se compone por un Generador y un Discriminador compitiendo entre sí, donde el Generador permite transformar un vector aleatorio en una imagen del dominio utilizado para el entrenamiento del mismo. Por otro lado, el Discriminador intenta discernir entre imágenes reales del dominio e imágenes ficticias provenientes del generador. La competencia de estas dos redes neuronales lleva al perfeccionamiento de ambas hasta alcanzar el punto de equilibrio.

Basándonos en el trabajo de WGAN-GP, que se trata de un modelo GAN con ciertas mejoras con respecto a la propuesta original, presentamos un nuevo elemento dentro de este ecosistema al que definimos como Inversa o Encoder. Este último permite la codificación de imágenes de rostros del dominio particular en un vector Z perteneciente al espacio latente.

Como pudimos comprobar mediante la regeneración de dichas codificaciones nuevamente al espacio de imágenes del dominio, las mismas logran capturar las características semánticas principales de la imagen como: orientación, sexo, iluminación, sonrisa, ojos, color de pelo, color de piel, vello facial, maquillaje, etc. en los 3 datasets utilizados.

Pudimos comprobar la correctitud de la hipótesis planteada en las motivaciones. La misma sugería que la calidad de reconstrucción en el caso del dataset más restringido, compuesto únicamente por la identidad de Leonardo DiCaprio, es mayor a un dataset de diversas identidades como CelebA o Humanae. Sin embargo, esto no quiere decir que hayamos obtenido resultados no satisfactorios en estos últimos dos casos.

Por otro lado, cabe destacar que la capacidad de regeneración de una identidad nunca vista por el modelo (exceptuando el caso particular de la identidad de DiCaprio) es superior en el modelo entrenado con el dataset CelebA. Este

resultado era esperable debido a la mayor cantidad de ejemplos de entrenamiento con respecto a Humanae y a la composición de una única identidad para el dataset de DiCaprio.

Además, para el caso del dataset Humanae, los resultados son los esperados para su tamaño y composición. Obtuvimos la menor calidad de reconstrucción de imágenes propias del dominio y casi nula para el caso de identidades nunca vistas. Sin embargo, con sus 3000 ejemplos nos permite comprobar que no es necesario una enorme cantidad de datos de entrenamiento para el entrenamiento de la inversa propuesta.

Finalmente, en términos de generación y animación de rostros sintéticos con un comportamiento determinado; y de suplantación de identidad en sistemas de validación por reconocimiento facial podemos decir que logramos buenos resultados. Estos son apreciables en las secciones 4.5 y 4.6.

En el primer caso, se muestra cómo las imágenes sintéticas mantienen características de las originales como ya hemos mencionado y, en el segundo caso, presentamos un análisis utilizando una medida cuantitativa que sugiere muy buenos resultados de reconstrucción.

5.2. Trabajos Futuros

Como trabajos a futuro podemos proponer analizar cómo funciona el modelo propuesto para datasets de otros dominios.

Además, evaluar la utilización de una distancia coseno durante el entrenamiento del modelo en vez de utilizar el error absoluto medio. Esta medida fue utilizada en este trabajo en experimentos posteriores al entrenamiento dando buenos resultados. Sumado a esto, se podría modificar la función de costo actual utilizando una máscara al calcular la distancia de píxeles entre dos imágenes de manera que considere sólo la porción del rostro y no el fondo, el cual no aporta información relevante para nuestro objetivo.

Por último, si bien utilizamos WGAN-GP como modelo base, los procedimientos y modelo propuestos podrían aplicarse a otros modelos GAN actuales más cercanos al estado del arte y así obtener imágenes de mayor calidad y resolución.

Bibliografía

[AB17] Martin Arjovsky y Léon Bottou. «Towards Principled Methods for Training Generative Adversarial Networks». En: *arXiv e-prints*, arXiv:1701.04862 (ene. de 2017), arXiv:1701.04862. arXiv: 1701.04862 [stat.ML].

[ACB17] M. Arjovsky, S. Chintala y L. Bottou. «Wasserstein GAN». En: *ArXiv e-prints* (ene. de 2017). arXiv: 1701.07875 [stat.ML].

[CB16] Antonia Creswell y Anil Anthony Bharath. *Inverting The Generator Of A Generative Adversarial Network*. 2016. arXiv: 1611.05644 [cs.CV].

[CB18] Antonia Creswell y Anil A Bharath. *Inverting The Generator Of A Generative Adversarial Network (II)*. 2018. arXiv: 1802.05701 [cs.CV].

[Cho17] Francois Chollet. *Deep Learning with Python*. 1st. Greenwich, CT, USA: Manning Publications Co., 2017.

[CIF18] CIFASIS. *Aprendizaje Automatizado y Aplicaciones*. 2018. URL: http://www.cifasis-conicet.gov.ar/grupos/1 (visitado 01-11-2018).

[Das16] Angélica Dass. *Photography by Angélica Dass*. 2016. URL: http://humanae.tumblr.com/ (visitado 01-05-2018).

[Den+15] Emily Denton y col. *Deep Generative Image Models using a Laplacian Pyramid of Adversarial Networks*. 2015. arXiv: 1506.05751 [cs.CV].

[Dum+16] V. Dumoulin y col. «Adversarially Learned Inference». En: *ArXiv e-prints* (jun. de 2016). arXiv: 1606.00704 [stat.ML].

[Goo+14] Ian J. Goodfellow y col. «Generative Adversarial Networks». En: *arXiv e-prints*, arXiv:1406.2661 (jun. de 2014), arXiv:1406.2661. arXiv: 1406.2661 [stat.ML].

[Goo18a] Google. *Backpropagation algorithm*. 2018. URL: https://google-developers.appspot.com/machine-learning/crash-course/backprop-scroll/ (visitado 16-01-2019).

[Goo18b] Google. *TensorFlow*. 2018. URL: https://www.tensorflow.org/ (visitado 15-10-2018).

[Gul+17a] Ishaan Gulrajani y col. *Code for Improved Training of Wasserstein GANs*. https://github.com/igul222/improved_wgan_training. 2017.

[Gul+17b] Ishaan Gulrajani y col. *Improved Training of Wasserstein GANs*. 2017. arXiv: 1704.00028 [cs.LG].

[He+16] Kaiming He y col. «Deep Residual Learning for Image Recognition». En: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (jun. de 2016). DOI: 10.1109/cvpr.2016.90. URL: http://dx.doi.org/10.1109/CVPR.2016.90.

[Hua+07] Gary B. Huang y col. *Labeled Faces in the Wild: A Database for Studying Face Recognition in Unconstrained Environments*. Inf. téc. 07-49. University of Massachusetts, Amherst, oct. de 2007.

[Hui18] Jonathan Hui. *GAN - Wasserstein GAN & WGAN-GP*. 2018. URL: https://medium.com/@jonathan_hui/gan-wasserstein-gan-wgan-gp-6a1a2aa1b490 (visitado 30-06-2019).

[Kar18] Andrej Karpathy. *CS231n: Convolutional Neural Networks for Visual Recognition*. 2018. URL: http://cs231n.github.io/convolutional-networks/ (visitado 30-06-2019).

[KB14] Diederik P. Kingma y Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. arXiv: 1412.6980 [cs.LG].

[Lar+15] Anders Boesen Lindbo Larsen y col. *Autoencoding beyond pixels using a learned similarity metric*. 2015. arXiv: 1512.09300 [cs.LG].

[LC10] Yann LeCun y Corinna Cortes. «MNIST handwritten digit database». En: (2010). URL: http://yann.lecun.com/exdb/mnist/.

[Lee13] Dong-Hyun Lee. «Pseudo-Label : The Simple and Efficient Semi-Supervised Learning Method for Deep Neural Networks». En: *ICML 2013 Workshop : Challenges in Representation Learning (WREPL)* (jul. de 2013).

[Liu+15] Ziwei Liu y col. «Deep Learning Face Attributes in the Wild». En: *Proceedings of International Conference on Computer Vision (ICCV)*. Dic. de 2015.

[Mai+18] Guangcan Mai y col. «On the Reconstruction of Face Images from Deep Face Templates». En: *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2018), págs. 1-1. ISSN: 2160-9292. DOI: 10.1109/tpami.2018.2827389. URL: http://dx.doi.org/10.1109/TPAMI.2018.2827389.

[Mik+13] Tomas Mikolov y col. *Distributed Representations of Words and Phrases and their Compositionality*. 2013. arXiv: 1310.4546 [cs.CL].

[Mit97] Tom M. Mitchell. *Machine Learning*. WCB/McGraw-Hill, 1997.

[MO14] Mehdi Mirza y Simon Osindero. *Conditional Generative Adversarial Nets*. 2014. arXiv: 1411.1784 [cs.LG].

[New69] Allen Newell. «Perceptrons. An Introduction to Computational Geometry. Marvin Minsky and Seymour Papert. M.I.T. Press, Cambridge, Mass., 1969. vi + 258 pp., illus. Cloth, 12; *paper*,4.95». En: *Science* 165.3895 (1969), págs. 780-782. ISSN: 0036-8075. DOI: 10.1126/science.165.3895.780. eprint: http://science.sciencemag.org/content/165/3895/780.full.pdf. URL: http://science.sciencemag.org/content/165/3895/780.

[Per+16] Guim Perarnau y col. *Invertible Conditional GANs for image editing*. 2016. arXiv: 1611.06355 [cs.CV].

[Pow18] Victor Powell. *Image Kernels Explanied Visually*. 2018. URL: http://setosa.io/ev/image-kernels/ (visitado 30-06-2019).

[RMC15] Alec Radford, Luke Metz y Soumith Chintala. *Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*. 2015. arXiv: 1511.06434 [cs.LG].

[Rud16] Sebastian Ruder. *An overview of gradient descent optimization algorithms*. 2016. arXiv: 1609.04747 [cs.LG].

[Sah18] Sumit Saha. *A Comprehensive Guide to Convolutional Neural Networks*. 2018. URL: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53 (visitado 02-07-2019).

[Sal+16] Tim Salimans y col. *Improved Techniques for Training GANs*. 2016. arXiv: 1606.03498 [cs.LG].

[Sil18] Thalles Silva. *Generative Adversarial Network framework*. https://www.freecodecamp.org/news/an-intuitive-introduction-to-generative-adversarial-networks-gans-7a2264a81394/. 2018. (Visitado 01-05-2018).

[Vil09] Cédric Villani. «Optimal Transport : Old and New». En: Series Title: Grundlehren der mathematischen Wissenschaften. Springer, 2009.

[Whi16] Tom White. *Sampling Generative Networks*. 2016. arXiv: 1609.04468 [cs.NE].

[Wu+16] Jiajun Wu y col. *Learning a Probabilistic Latent Space of Object Shapes via 3D Generative-Adversarial Modeling*. 2016. arXiv: 1610.07584 [cs.CV].

[YG14] Aron Yu y Kristen Grauman. «Fine-Grained Visual Comparisons with Local Learning». En: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Jun. de 2014.

[Zan17] Álvaro Zanetti. *Detección, seguimiento e identificación de rostros*. https://github.com/zanettialvaro/pasantiaCIFASIS. 2017.

[Zan19a] Álvaro Zanetti. *Codificación Y Reconstrucción De Rostros Con Redes Adversarias Generativas (video 1 de 5)*. 2019. URL: https://youtu.be/R0d64bfwRXg (visitado 29-06-2019).

[Zan19b] Álvaro Zanetti. *Codificación Y Reconstrucción De Rostros Con Redes Adversarias Generativas (video 2 de 5)*. 2019. URL: https://youtu.be/KVdQper6TPQ (visitado 29-06-2019).

[Zan19c] Álvaro Zanetti. *Codificación Y Reconstrucción De Rostros Con Redes Adversarias Generativas (video 3 de 5)*. 2019. URL: https://youtu.be/jHhvZYZYON4 (visitado 29-06-2019).

[Zan19d] Álvaro Zanetti. *Codificación Y Reconstrucción De Rostros Con Redes Adversarias Generativas (video 4 de 5)*. 2019. URL: https://youtu.be/6-AyggBThuY (visitado 29-06-2019).

[Zan19e] Álvaro Zanetti. *Codificación Y Reconstrucción De Rostros Con Redes Adversarias Generativas (video 5 de 5)*. 2019. URL: https://youtu.be/N5Uu6gZm3OI (visitado 29-06-2019).