Universidad Nacional Autónoma de México

Facultad de Ingeniería

Predicción de "likes" en Twitter para memes aleatorios

T E S I S

PARA OPTAR POR EL GRADO DE:

Ingeniero en computación

PRESENTA:

Diego Alberto Salinas Navarro

TUTOR:

Dr. Ivan Vladimir Meza Ruiz

Ciudad Universitaria, CDMX, 2022

Índice general

Índice de figuras

Resumen

Los **memes**, antes y sobre todo ahora, representan la realidad y, por tanto, también la constituyen, la remodelan. Además, lo hacen con ese matiz de ironía que caracteriza al ser humano, capaz de revestir de humor su propia desgracia (32).

El **shitposting**, derivado del deseo del ser humano por representar la realidad bajo su propia perspectiva, contiene todo aquel contenido irónico, sarcástico, irrelevante e incoherente hecho mayormente con propósitos graciosos. Las redes sociales y la exorbitante popularidad del Internet han permitido al shitposting volverse un tema cada más relevante para los usuarios.

Si bien existe el campo del humor computacional, este se ha caracterizado por incursionar en proyectos que implican la resolución de tareas apoyándose del procesamiento de lenguaje natural (23). Y es que, a día de hoy no existe ningún antecedente que relacione de forma clara el shitposting y el aprendizaje profundo.

En este trabajo, se aborda esa problemática mediante la creación de modelos de visión computacional basados en redes neuronales siamesas y redes neuronales convolucionales. Se diseñaron distintas arquitecturas cuyo principal objetivo era la clasificación e interpretación de shitposting.

Para el entrenamiento y verificación de estos modelos, se recolectaron cierta cantidad de memes provenientes de **ShitpostBot5000**, con la idea de crear un conjunto de datos donde los memes sean diferenciados por su respectiva cantidad de likes en twitter. La implementación fue realizada en *Python*, utilizando la biblioteca de inteligencia arificial *PyTorch*.

Una vez diseñados e implementados, estos modelos pasaron por una etapa de verificación, donde su desempeño fue evaluado por la predicción de likes de los memes que recibían como entrada.

En cuanto a los resultados obtenidos, se logró tomar modelos de visión computacional y deformar su propósito original para que pudieran aprender a intepretar de forma muy básica el humor y contexto de los memes y el shitposting.

Capítulo 1

Introducción

1.1. Estado del arte

El humor es el acto de comentar o enjuiciar una situación desde una perspectiva propia con la intención de hacer reír o generar felicidad en el espectador. Podríamos decir que el objetivo principal del humor es el de resaltar el lado ridículo o cómico de las cosas, es difícil asegurar qué es y qué no es gracioso, ya que este concepto depende de la percepción de cada persona, mientras que a una persona algo puede parecerle gracioso, otra puede interpretarlo de una forma contraria.

El humor es un campo de investigación multidisciplinar. A lo largo del tiempo, múltiples profesionistas han estado trabajando en el humor en muchos campos de investigación tales como la psicología, filosofía y lingüística, sociología y literatura. En el contexto del humor en la inteligencia artificial, la investigación tiene como objetivo modelar el humor de una forma computacionalmente viable (16).

1.1.1. Shitposting

Es el acto de publicar contenido con calidad pobre o inútil cuyo objetivo es lograr la mayor reacción con el menor esfuerzo posible. El *shitposting* nació para crear polémica y generar discusión u odio respecto a un tema específico, generalmente temas políticos, un ejemplo reciente son las elecciones de 2016 en Estados Unidos, donde en foros de extrema derecha abundaron imágenes desacreditando a Hillary Clinton y alabando a Donald Trump (6).

En años recientes, se ha observado como el *shitposting* se ha desviado y ha pasado a tener objetivos más simples, como el hecho de crear humor irónico con memes absurdos, el *shitposting* es en sí la antítesis del humor elaborado, ya que se trata solo de imágenes o videos sin sentido aparente cuyo contexto es otorgado

por el espectador.

Es justamente en este aspecto donde entra la subjetividad del humor que maneja el *shitposting*, mientras que una persona promedio puede ver este tipo de contenido como algo absurdo o estúpido, alguien que conozca el trasfondo puede comprender el humor que hay detrás.

1.1.2. ShitpostBot5000

ShitpostBot5000 es una página de Facebook y cuenta de Twitter donde un bot publica una imagen cada media hora. Selecciona al azar una plantilla (template) y luego la completa con imágenes de origen (source) formando un meme.

La idea de *ShitpostBot5000* es crear memes aleatorios cuyo contexto sea otorgado por los mismos usuarios de las redes sociales, quienes califican la calidad de estos dependiendo de las interacciones que produce. La magia detrás del bot es que pueda crear imágenes graciosas a partir de la fusión de dos imágenes que aparentemente no tengan nada en común.

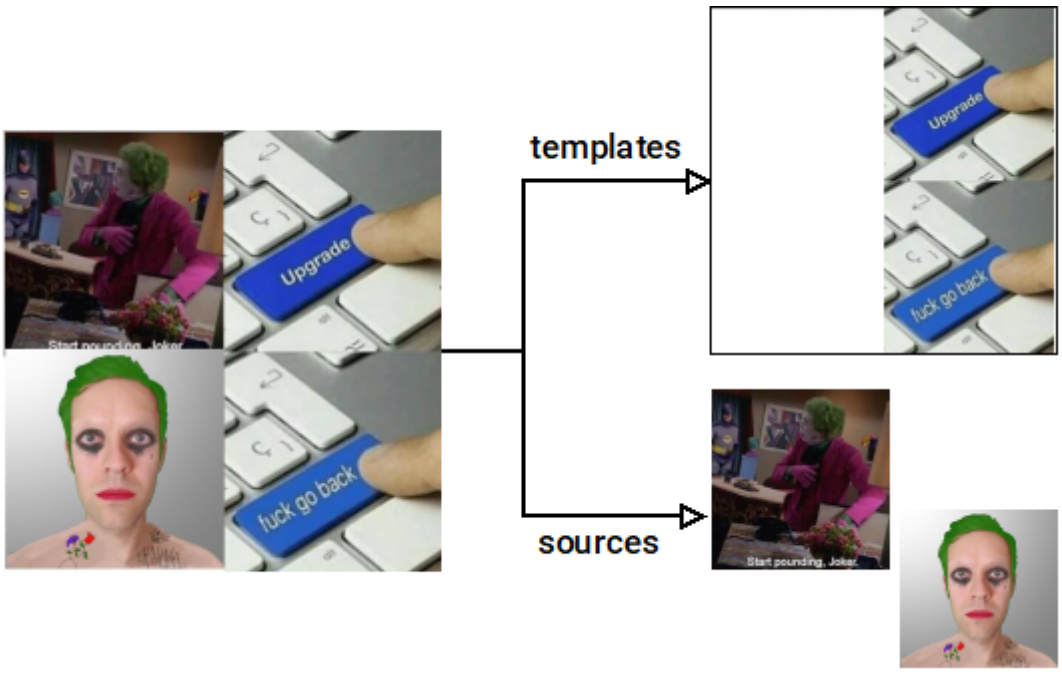

Figura 1.1: Estructura de un meme de *ShitpostBot5000*. (Elaboración propia)

1.2. Planteamiento de problema

Según Graeme Ritchie, uno de los principales exponentes en el área de humor computacional, no existe una teoría del humor que sea lo suficientemente precisa, detallada y formal para ser implementable. Por lo tanto, el humor computacional hasta ahora ha consistido en gran medida de pequeños prototipos de investigación basados en mecanismos diseñados específicamente para resolver pequeños problemas (23). En pleno 2021, el *shitposting* representa una de las corrientes más influyentes de Internet y uno de los temas de discusión más recurrentes en las redes sociales.

Y es que, establecer una relación sólida entre el *shitposting* y el aprendizaje profundo se ha convertido en un problema fascinante para un pequeño nicho de la comunidad científica, sobre todo a desarrolladores de bots, tales como *"The Bot Appreciation Society"* (1). A diferencia de tareas comunes de visión computacional como la clasificación de imágenes, este problema se amplía hasta el estudio de un contexto independiente al contenido de las imágenes, donde, además de realizar una extracción y análisis de características, se debe tomar en cuenta el contexto detrás de cada imagen (20), es decir, aquello que le da sentido al meme y lo hace gracioso.

Es por eso que, se debe transformar las tareas a las que están destinadas las redes neuronales convolucionales, buscando darles este nuevo enfoque. Tal vez, en un futuro, la comunidad será capaz de crear nuevas arquitecturas que estén completamente especializadas para el análisis del humor y así, poder realizar tareas relacionadas a este de forma más sencilla.

1.3. Objetivo y metas

Para construir un sistema computacional capaz de procesar memes aleatorios, así como realizar una interpretación y clasificación del humor relacionado a estos utilizando herramientas de aprendizaje profundo, es necesario cumplir las siguientes metas trazadas:

- Usar herramientas de extracción de datos de twitter para recolectar y analizar memes de *ShitpostBot5000*, buscando crear un conjunto de datos donde se puedan observar memes de diversos grados de popularidad, así como sus respectivas características.

- Emplear redes neuronales siamesas y redes neuronales convolucionales para procesar y representar imágenes. Además, realizar un afinamiento de las

arquitecturas de redes neuronales convolucionales más populares en el campo de la visión computacional para la realización tareas relacionadas a la detección e interpretación del humor en memes aleatorios o *shitposting*.

- Diseñar, implementar y evaluar una serie de modelos de aprendizaje profundo que tengan como principal función la interpretación y clasificación de *shitposting* basado en su éxito, es decir, el número de likes obtenidos.

1.4. Estructura de la tesis

En el presente documento, se seguirá la siguiente estructura:

- Marco teórico: Se explicarán de forma clara y concisa todos los conceptos fundamentales que justifican la parte experimental del trabajo.
- Metodología: Se presentarán y justificarán los procedimientos utilizados para la parte experimental, así como el cumplimiento de las metas establecidas anteriormente.
- Experimentación y análisis de resultados: Se propondrán diversas arquitecturas, se medirán sus desempeños y se hará un análisis exhaustivo de estos, buscando encontrar un patrón o enfoque adecuado que permita resolver de manera satisfactoria el problema planteado.
- Conclusiones: Se realizarán una serie de reflexiones basadas en los resultados obtenidos, además, se propondrán trabajos futuros buscando complementar una posible solución.

Capítulo 2

Marco teórico

A lo largo de los siglos, los seres humanos se han preguntado como es que funciona el cerebro, siendo hasta el día de hoy, en muchos aspectos, un misterio. Aún así, se define al cerebro humano como una red neuronal biológica interconectada donde las neuronas transmiten señales eléctricas, las dendritas reciben una señal de entrada, la procesan y posteriormente la envían por el axón a la neurona adyacente (Figura 2.1). No es sorpresa que los computólogos hayan inspirado varios de sus trabajos en el cerebro humano, un ejemplo claro es el primer modelo conceptual de una red neuronal artificial creada en 1943 por McCulloch y Pitts (28).

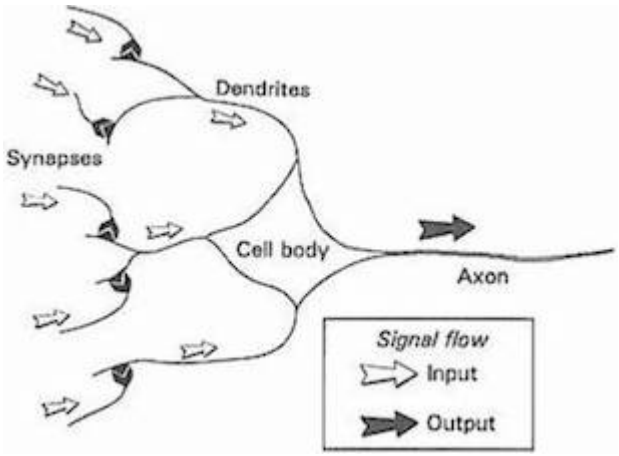

Figura 2.1: Esquema de una neurona. (Tomado de (28))

Las redes neuronales artificiales fueron concebidas como un modelo computacional basado en la estructura cerebral para resolver problemas que para un ser humano son triviales, mientras que para una máquina suelen ser bastante complejos. Desde principios de la década pasada, las redes neuronales se han convertido

en la técnica dominante del aprendizaje automático.

2.1. Redes neuronales y su funcionamiento

Las redes neuronales se componen por una serie de capas de nodos, incluyendo una capa de entrada, capas ocultas y una capa de salida. Cada nodo o neurona funciona de manera simple, cumpliendo tres funciones básicas: lee una entrada x_i, la procesa y genera una salida. Las conexiones entre los nodos tienen pesos definidos w_i, mientras que cada nodo cuenta con un umbral definido b (Figura 2.2).

La principal función de los pesos es asignar cierta importancia a las entradas, ya que estos definen que tanto contribuye una entrada con respecto las otras. Todas las entradas son multiplicadas por sus respectivos pesos y se suman:

$$z = f(X) = b + \sum_{i}^{n} w_i x_i \tag{2.1}$$

La ecuación anterior puede ser definida de forma matricial (Ecuación 2.4), donde los parámetros y pesos de la red neuronal se encuentren en una matriz W (Ecuación 2.2) traspuesta y los datos de entrada en otra matriz X (Ecuación 2.3).

$$W = \begin{bmatrix} b \\ w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} \tag{2.2}$$

$$X = \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \tag{2.3}$$

$$z = f(x) = W^T X \tag{2.4}$$

$$y = \sigma(z) \tag{2.5}$$

Una vez realizada la suma ponderada, esta pasa por una función de activación y sufre una transformación (Ecuación 2.5). El objetivo de las funciones de activación es poder modelar funciones no lineales, ya que la mayoría de las tareas no

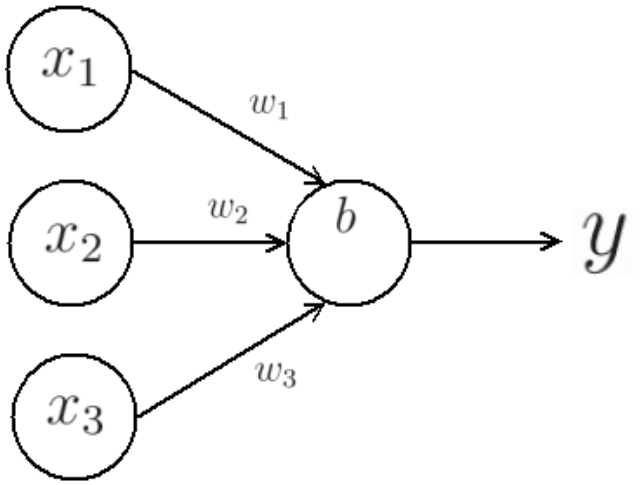

Figura 2.2: Diagrama de un nodo o neurona artificial. Se observan tres datos de entrada $[x_1, x_2, x_3]$, sus respectivos pesos $[w_1, w_2, w_3]$, el umbral b y la salida y. (Elaboración propia)

pueden ser resueltas con modelo lineales (33). Aunque hay infinidad de funciones de activación, solamente se profundizará en dos: la sigmoide y la ReLU.

- La sigmoide es la función de activación más antigua y popular, actúa como una especie de "aplanadora" comprimiendo el rango de la salida de 0 a 1. Si la entrada es un valor negativo grande, la sigmoide tiende a 0, mientras que en el caso donde se tiene una entrada positiva grande, la sigmoide tiende a 1.

- La ReLU se puede definir como el "máximo", en otras palabras, permite el paso de todos los valores positivos y asigna un 0 a todos los valores negativos.

Una vez calculado el resultado de la función de activación, este pasa como salida de la neurona y es enviada a la siguiente capa. Es así como los nodos comparten información entre ellos, es decir, la salida de un nodo se vuelve la entrada de otro, esto permite que la entrada vaya pasando a través de las capas de la red hasta la salida (3) (Figura 2.3). Este proceso es conocido como la propagación hacia delante (*forward propagation*).

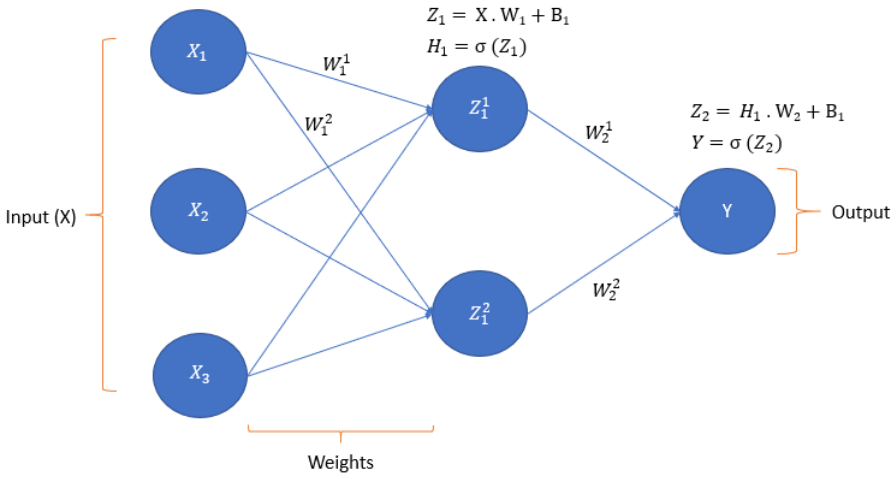

Figura 2.3: Propagación hacia delante en una red neuronal. (Elaboración propia)

2.2. Entrenamiento de redes neuronales

Entendiendo de una forma básica cómo es que las neuronas funcionan y se comunican entre sí, toca profundizar en cómo es que son entrenadas las redes neuronales y, en consecuencia, aprenden.

El principal objetivo del entrenamiento es determinar el mejor conjunto de pesos para maximizar la precisión de la red neuronal, dicho esto, se debe comprender que los pesos son fuertemente interdependientes y que la modificación de uno puede alterar el comportamiento de todas las neuronas de la red.

Es prácticamente imposible obtener un conjunto de pesos ideales modificando uno a la vez, por eso es preferible modificar todos los pesos de forma simultánea. El enfoque más simple se trata de modificar pesos de forma aleatoria, registrar los resultados y guardar la combinación que de los mejores resultados. Prácticamente estamos tratando de optimizar los pesos por fuerza bruta (2).

Aunque el enfoque aleatorio puede resultar bastante atractivo para entrenar las redes neuronales, no es viable en redes con una gran cantidad de nodos, por lo que a lo largo de los años se exploraron otras soluciones más elegantes buscando una mayor eficiencia computacional.

2.2.1. Funciones de pérdida

Las funciones de pérdida son uno de los pilares en el entrenamiento de redes neuronales, ya que estas se encargan de establecer un valor de penalización por no lograr el resultado esperado. Si la diferencia entre el valor predicho y el valor esperado es grande, entonces la función de pérdida da un número más alto como salida y en cambio, si la diferencia es pequeña, da como resultado números más pequeños.

La importancia de las funciones de pérdida es que estas son el principal indicativo de qué tan bien está realizando una tarea la red neuronal. El error obtenido se utiliza posteriormente para modificar los pesos en las capas ocultas, buscando que la próxima vez el error disminuya y que los valores de salida estén más cerca de los valores esperados (Figura 2.4).

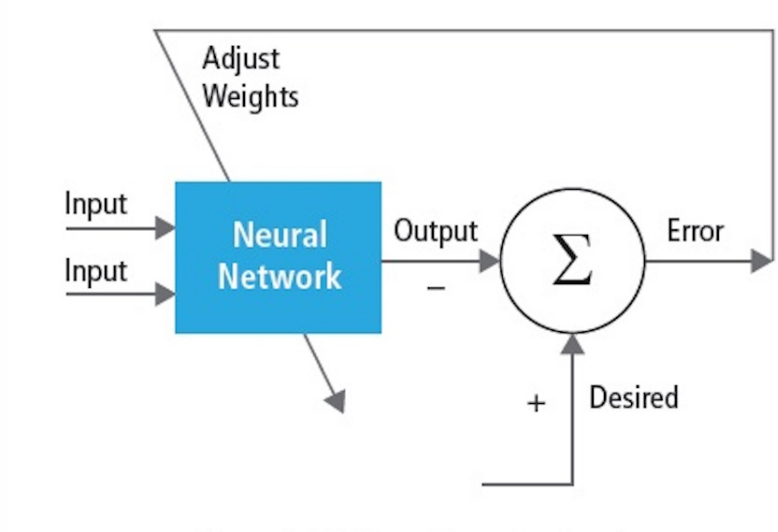

Figura 2.4: Diagrama de entrenamiento de una red neuronal. (Tomado de (26))

La función de pérdida varía dependiendo de que tipo de tarea será realizada, aunque normalmente se destacan tres tipos:

- Las funciones de pérdida de regresión son aquellas que se basan en la regresión lineal, estas siempre tratan de establecer una relación lineal entre una variable dependiente y y una variable independiente x. Son utilizadas para predecir valores continuos como precios de bienes raíces o ventas en una empresa.
- Las funciones de pérdida de clasificación binaria miden el desempeño de los modelos de clasificación, normalmente devuelve una probabilidad entre 0 y

1 de que un objeto pertenezca a la clase. Estas funciones se utilizan cuando se tienen situaciones positivo/negativo, es decir, "un objeto pertenece a esta clase o no".

- Las funciones de pérdida de clasificación multiclase son utilizadas cuando se tienen 3 clases o más, tienen un funcionamiento similar a las funciones de clasificación binaria, solo que estas se encuentran enfocadas a tareas de clasificación más complejas. Son ampliamente utilizadas en clasificación de imágenes y texto.

2.2.2. Descenso de gradiente (*Gradient Descent*)

Hoy en día, muchos problemas involucran la optimización de funciones multivariables y la mayoría de estos encuentran solución en un algoritmo eficiente conocido como descenso de gradiente.

El algoritmo empieza tomando parámetros aleatorios, después averigua en qué dirección la función de pérdida se inclina más hacia abajo y continúa levemente en esa dirección, es decir, se determina qué tanto deben ser modificados los parámetros de modo que la función de pérdida disminuya en mayor cantidad. El proceso se repite hasta que la función de pérdida llega hasta su punto más bajo (2) (Figura 2.5).

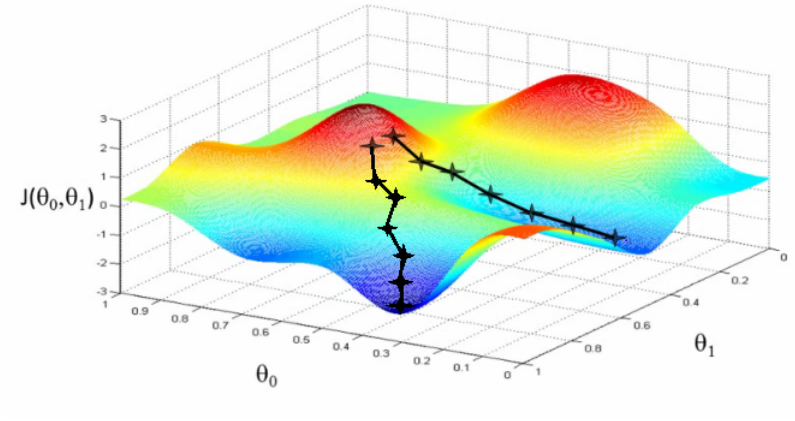

Figura 2.5: Descenso de gradiente de una función de pérdida no convexa con dos parámetros θ_1 y θ_2. (Tomado de (33))

Para ver en que dirección está disminuyendo más la pérdida, es necesario valerse del gradiente. El gradiente es un vector que contiene las derivadas parciales

de la función respecto a cada variable, es decir, contiene las pendientes de las líneas tangentes de la función de pérdida respecto a cada eje.

Ahora, sea cual sea la función de perdida L, se obtiene su gradiente calculando las derivadas parciales respecto a todos los parámetros. El gradiente queda de la siguiente manera:

$$\nabla L(\theta) = \left(\frac{\delta L}{\delta w_1}, \frac{\delta L}{\delta w_2}, \cdots, \frac{\delta L}{\delta w_n} \right) \tag{2.6}$$

Donde:

$$\theta = [w_1, w_2, \cdots, w_n] \tag{2.7}$$

Es necesario saber cómo es que se va a mover y en qué dirección, se debe recordar que el objetivo es llegar a un conjunto de parámetros donde la función de pérdida se encuentre en su mínimo.

Se comienza partiendo de un punto inicial que es elegido de forma aleatoria, se calcula la derivada y se observa la inclinación de la pendiente de la recta tangente. Esta pendiente será mayor en el punto inicial, pero a medida que se cambian los parámetros, esta se reducirá gradualmente hasta llegar al punto mínimo de la curva.

Surge otra duda, ¿Qué tanto debe moverse?, la respuesta a esta pregunta es la introducción de un hiper parámetro llamado tasa de aprendizaje. La tasa de aprendizaje η define qué tan grandes serán los pasos que demos hasta llegar al mínimo (10), este dato es asignado de manera empírica y cambia dependiendo del comportamiento de la función de pérdida. Si elegimos un valor muy pequeño, tardaremos demasiado tiempo en alcanzar el mínimo, mientras que, en caso contrario, si nuestra taza de aprendizaje es demasiado grande, daremos saltos enormes y estaremos vagando sin encontrar el mínimo (Figura 2.6).

El principal problema al que surge cuando se realiza el descenso de gradiente es que todas las funciones de pérdida son complejas, podemos pensar en ellas como funciones con un gran número de valles y crestas. Esto significa que existen varios mínimos locales y existe la posibilidad de quedar atascado en uno de estos (2).

2.2.2.1. Descenso de gradiente por lotes (*Batch Gradient Descent*)

El descenso de gradiente por lotes suma el error para cada punto dentro del conjunto de entrenamiento, actualizando los parámetros después de evaluar todos los datos de entrenamiento (10). Como se necesita realizar todo este cómputo para actualizar solamente una vez los parámetros y completar una época de entrenamiento, este enfoque de descenso de gradiente suele ser muy lento y caro en memoria.

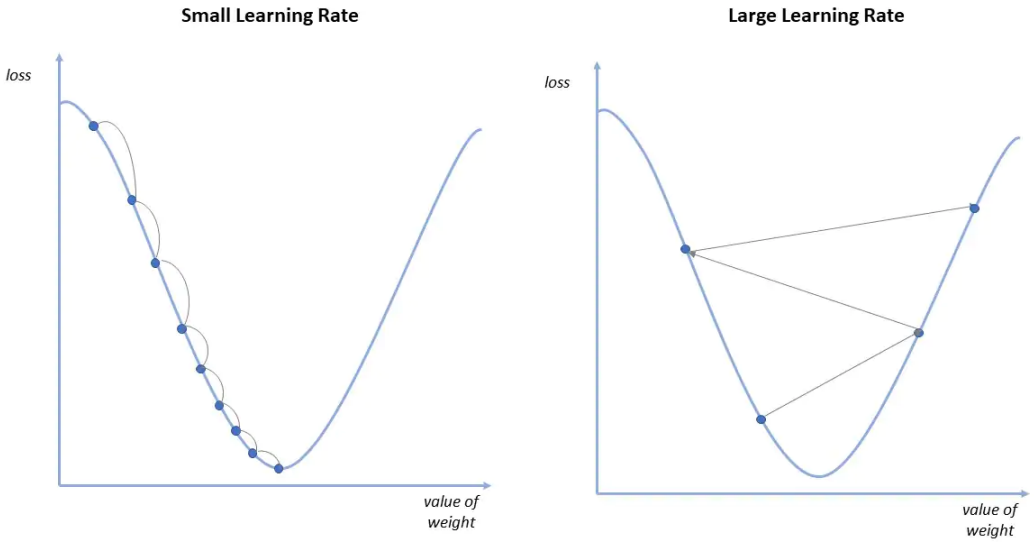

Figura 2.6: Efecto de la tasa de aprendizaje η en el descenso del gradiente. (Tomado de (11))

Aunque suele producir un gradiente más o menos estable y llega a una convergencia, esta no siempre es la ideal, ya que puede llegar a encontrar un mínimo local. Además, tampoco permite actualizar parámetros si agregamos nuevos ejemplos sobre la marcha.

$$\theta = \theta - \eta \cdot \nabla_\theta L(\theta) \tag{2.8}$$

2.2.2.2. Descenso de gradiente estocástico (*Stochastic Gradient Descent*)

En este caso, las muestras del conjunto de datos de entrenamiento se toman individualmente, se calcula el gradiente con respecto a ese punto y se actualizan los parámetros. Es decir, ya no es necesario cargar el conjunto de datos completo en memoria, ya que ahora en cada época de entrenamiento solo se carga una muestra (25).

En un principio parece una mala idea, ya que es común observar muestras "atípicas" en todos los conjuntos de entrenamiento que puedan entorpecer los resultados y, en consecuencia, dar una mala aproximación de gradiente. Resulta que, si se realiza este proceso con varias muestras del conjunto de forma aleatoria,

las fluctuaciones del gradiente pueden llegar a converger y llevar a una solución.

Este enfoque puede ayudar a "escapar" de los mínimos locales y saltar a otros que sean mejores, es decir, si se deja un mínimo local es posible saltar a uno que sea global.

$$\theta = \theta - \eta \cdot \nabla_\theta L(\theta; x^{(i)}; y^{(i)}) \tag{2.9}$$

2.2.2.3. Descenso de gradiente por lotes pequeños (*Mini-Batch Gradient Descent*)

Explorando las anteriores variaciones del descenso de gradiente, hay una forma de tomar lo mejor de ambos. El descenso de gradiente por lotes pequeños combina conceptos del descenso de gradiente por lotes y el descenso de gradiente estocástico (10).

Toma partes del conjunto de entrenamiento con k muestras y actualiza parámetros para cada una de estas partes, gracias a esto, reduce la varianza de las actualizaciones de los parámetros y se pueden optimizar las operaciones matriciales, incluso de manera paralelizable.

$$\theta = \theta - \eta \cdot \nabla_\theta L(\theta; x^{(i;i+k)}; y^{(i;i+k)}) \tag{2.10}$$

2.2.3. Propagación hacia atrás (*Back - Propagation*)

Independientemente de la variable del descenso de gradiente elegida, se debe garantizar el cálculo del gradiente de la función de pérdida respecto a cada uno de los parámetros de la red.

El enfoque más obvio para obtener las derivadas parciales es utilizando la fórmula de la derivada, es decir, realizar este cálculo por cada parámetro de la red neuronal (Ecuación 2.11).

$$\frac{\delta L}{\delta w_i} \approx \frac{L(\theta + \epsilon \cdot e_i) - L(\theta)}{\epsilon} \tag{2.11}$$

Donde ϵ es un número pequeño y e_i es un vector *one-hot* con el valor de 1 en la posición i y 0 en todas las demás.

Como se ha mencionado anteriormente, una red neuronal puede llegar a contener decenas de millones de parámetros y pesos, por lo que, en caso de elegir este enfoque para obtener el gradiente, se tendría que calcular la derivada millones de veces, lo que se traduce en una pobre eficiencia computacional.

Debido a la gran cantidad de desventajas del enfoque anterior, surgieron nuevas ideas que consiguieran lograr la viabilidad al momento de entrenar una red

neuronal. Es aquí, donde surgió la propagación hacia atrás como una alternativa para el entrenamiento de redes neuronales.

La meta de la propagación hacia atrás es la de hallar las derivadas parciales de la función de pérdida con respecto a todos los pesos y parámetros de la red, es capaz de calcular todos los elementos del gradiente. El proceso comienza con la propagación hacia delante, donde realiza un seguimiento y almacenamiento de todos los parámetros y pesos (Figura 2.7(a)), para después realizar un recorrido inverso, iniciando desde la capa de salida hasta la entrada, es en este recorrido en donde se van actualizando los parámetros y pesos de la red (Figura 2.7(b)), utilizando como principal instrumento la regla de cadena (22) (Figura 2.8).

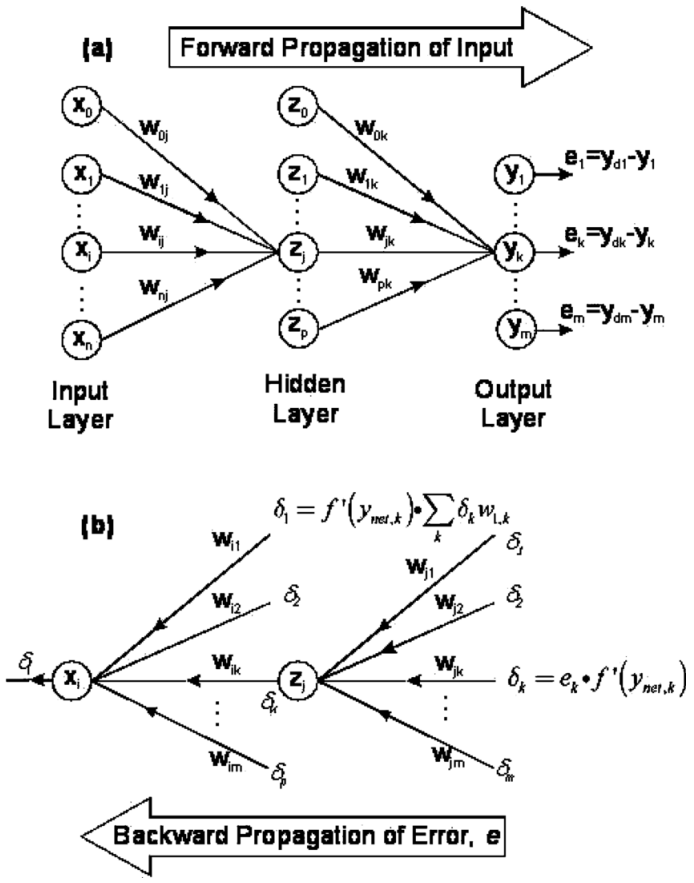

Figura 2.7: (a) Las entradas x_i son procesadas por la red neuronal hasta llegar a la capa de salida y obtener y_i, posteriormente se calcula el error e_i. (b) El error se propaga hacia atrás, calculando las derivadas parciales respecto a cada parámetro de la red.

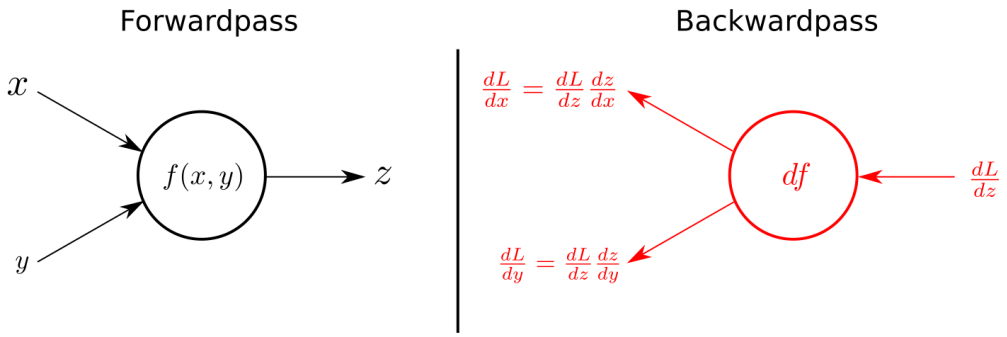

Figura 2.8: Operaciones realizadas en el paso hacia delante (*forwardpass*) y el paso hacia atrás(*backwardpass*). (Tomado de (3))

Y es que, es gracias a la regla de la cadena (Ecuación 2.12) que somos capaces de descomponer una derivada parcial en un producto de muchas otras, es así como se van "encadenando" operaciones de tal forma que podremos obtener las derivadas parciales de la función de perdida respecto a cada parámetro de la red, solo hay que seguir el camino correcto (Figura 2.9).

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx} \tag{2.12}$$

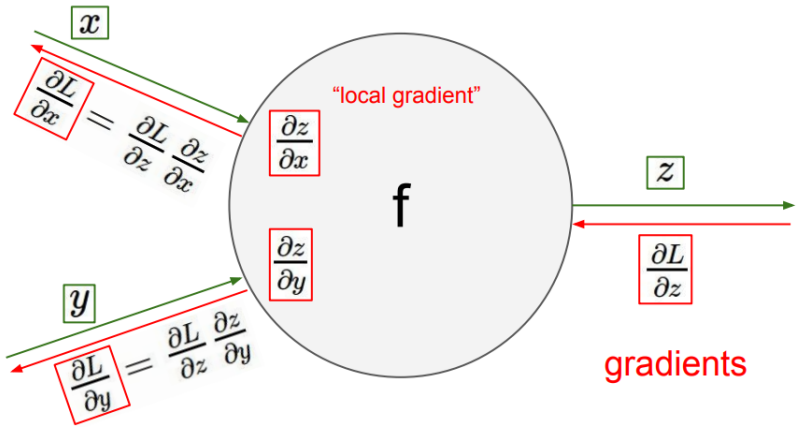

Figura 2.9: Encadenado de derivadas parciales en una neurona. (Tomado de (3))

La propagación hacia atrás se ha convertido en el procedimiento preferido para el entrenamiento de las redes neuronales, incluso ha sido introducido con éxito a varios tipos de redes, como las redes neuronales convolucionales (12).

2.3. Redes neuronales convolucionales (CNN)

Es de suponer que las redes neuronales son utilizadas en varios campos de la inteligencia artificial, ya que distintas tareas requieren distintas arquitecturas. El uso de las redes neuronales convolucionales es popular en tareas de visión computacional y reconocimiento de patrones, ya que muestran resultados bastante satisfactorios (31). Un ejemplo claro es el concurso ImageNet (7).

Una de las principales características las redes neuronales convolucionales (CNN) es que estas tienen un número de neuronas reducido, lo que lleva a dos ventajas principales: la primera, es que son mucho más baratas de entrenar y la segunda, es que, al reducir la cantidad de neuronas, el riesgo de sobreajuste es menor.

El funcionamiento de las redes neuronales convolucionales puede ser explicado a través de las distintas capas que las componen: la capa convolucional (*convolutional layer*), la capa de pooling (*pooling layer*) y la capa completamente conectada (*fully-connected layer*) (Figura 2.10).

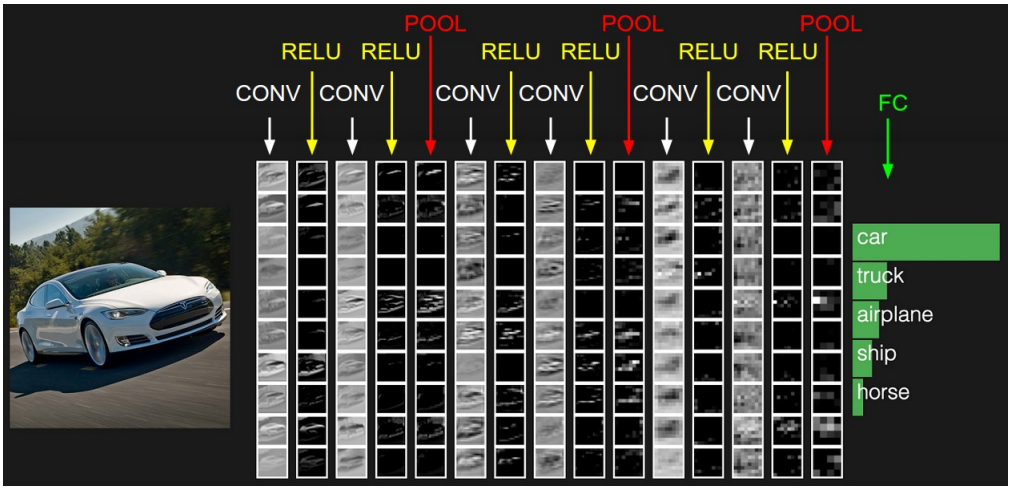

Figura 2.10: Arquitectura de una red neuronal convolucional. Se puede observar a la izquierda una imagen de entrada que pasa por las capas de convolucionales produciendo mapas de activación, capas de pooling que reducen la dimensionalidad de las salidas y una capa completamente conectada que arroja las probabilidades de pertenencia a cada clase. (Tomado de (31))

2.3.1. Capa convolucional

La capa convolucional es la capa más importante y donde la mayor parte de las operaciones ocurren, requiere varios componentes, como los datos de entrada, un filtro y un mapa de características. El uso de kernels o filtros se vuelve fundamental en esta capa, ya que se encargan de detectar características moviéndose a través de la imagen (18). Este proceso es, en resumidas cuentas, la convolución.

Si se superpone el kernel en un área de la imagen de entrada, se puede calcular el producto entre los números de la misma ubicación del kernel y la entrada, después se obtiene un solo número sumando estos productos.

La idea es ir moviendo el kernel desde el extremo superior de la imagen hasta el extremo inferior, es decir, el proceso se repite hasta que el kernel haya recorrido toda la imagen. Al final, se obtiene un vector conocido como un mapa de características o de activación. Cada kernel tendrá su mapa de activación correspondiente, los cuales se apilarán generando la salida de la capa convolucional (8) (Figura 2.11).

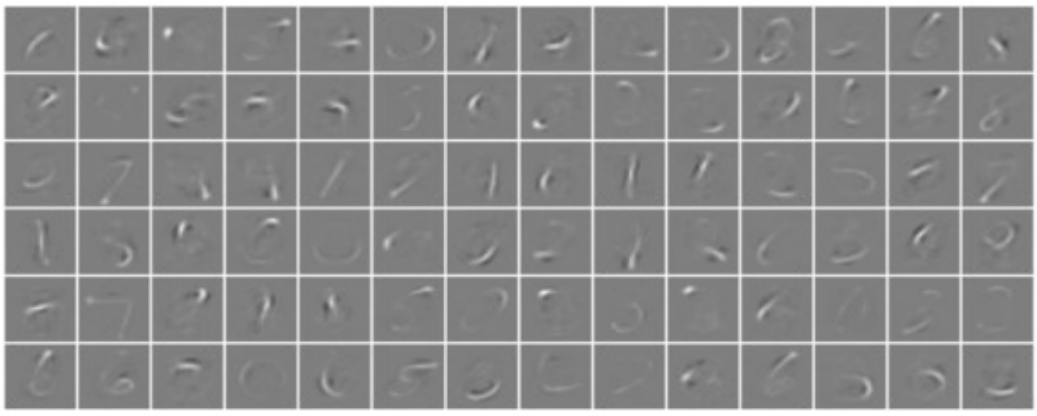

Figura 2.11: Mapas de activación de la base de datos MNIST de dígitos escritos a mano después de pasar por una capa convolucional. (Tomado de (18))

La salida de esta capa convolucional puede ser determinante para la complejidad del modelo y su funcionamiento, por lo que resulta fundamental optimizar de manera adecuada los hiper parámetros que intervienen en esta.

- La profundidad (*depth*) de la salida generada por las capas convolucionales es establecida por el número de neuronas dentro de la capa que están asignadas a la misma área de entrada. Reducir la profundidad puede reducir el número de neuronas en la red, pero también puede disminuir drásticamente la capacidad de detección de patrones de nuestro modelo.

- El paso (*stride*) indica qué tanto vamos a deslizar el filtro, si el valor del paso es igual a uno, significa que el filtro se moverá un píxel a la vez, lo que deriva en campos receptivos superpuestos que producen más activaciones. Aumentar el valor del paso reduce la superposición y genera un volumen de salida más pequeño.
- El relleno de ceros (*zero padding*) nos permite rellenar el borde de la entrada y controlar la dimensión del volumen de la salida.

2.3.2. Capa de pooling

El objetivo de la capa de pooling es reducir gradualmente la dimensionalidad de la representación, disminuyendo el número de parámetros y complejidad del modelo. Opera sobre cada mapa de activación y escala su dimensionalidad usando la función "MAX", normalmente las capas de max pooling reducen el tamaño del mapa de activación un 75 % mientras mantiene el volumen de profundidad a su tamaño estándar (Figura 2.12).

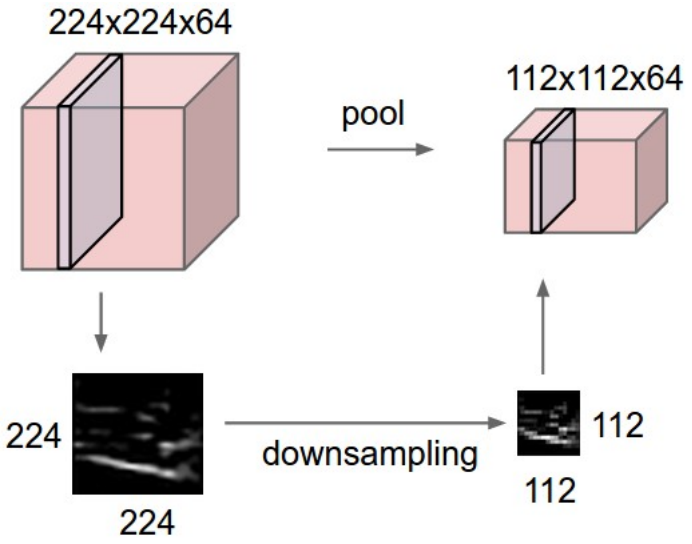

Figura 2.12: El volumen de entrada de dimensiones $[224 * 224 * 64]$ se agrupa con un filtro de *profundidad* 2 y *paso 2*, dando como resultado una salida de tamaño $[112 * 112 * 64]$, se conserva la profundidad del volumen. (Tomado de (31))

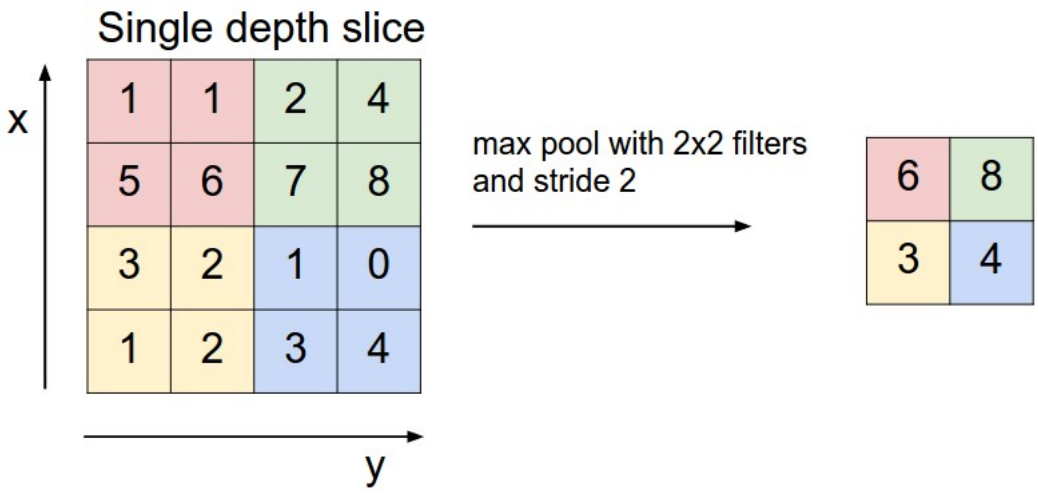

Figura 2.13: Escalado de dimensión a través de *maxpooling*, con un *paso* de 2. Se toma el máximo de cuatro números. (Tomado de (31))

Las capas de pooling son puestas después de cada capa convolucional y aunque se pierda una gran cantidad de información en estas, ayudan a reducir la complejidad (Figura 2.13), aumentar la eficiencia y limitar el riesgo de sobreajuste (18).

2.3.3. Capa completamente conectada

En la capa completamente conectada, cada nodo está directamente conectado a los nodos de las dos capas adyacentes. Realiza la tarea de clasificación basada en las características extraídas de las capas anteriores, generalmente utilizan una función de activación para clasificar de manea apropiada asociando probabilidades del 0 al 1 (31).

2.4. Redes neuronales siamesas (SNN)

Como ya se ha mencionado, las redes neuronales convolucionales resultan bastante útiles para tareas reconocimiento de patrones, así cómo la clasificación de objetos (35). Teniendo esto presente, surge la idea de como podríamos aprovechar estas tareas para encontrar o medir la similitud entre dos objetos.

Tradicionalmente, las redes neuronales se utilizan para predecir múltiples clases, lamentablemente, esto representa un problema cuando se necesita agregar o

eliminar clases, ya que, al momento de verse alterado el conjunto de datos, la red tiene que volver a ser entrenada desde un principio. Ahora, si se toma una red neuronal siamesa que aprenda una función de similitud, se puede eliminar estos problemas, ya que esto nos permite clasificar objetos en nuevas clases sin tener que volver a pasar por un nuevo proceso de entrenamiento. Esto es gracias al aprendizaje por similitud.

El aprendizaje por similitud es un área del aprendizaje automático supervisado en el que el objetivo es aprender una función de similitud que mida qué tan parecidos son dos objetos y a partir de esta información, devolver un valor de similitud (15). Se obtiene una puntuación de similitud más alta cuando los objetos son similares y una puntuación de similitud más baja cuando los objetos son diferentes.

Las redes neuronales siamesas involucran el uso de dos o más redes neuronales idénticas, es decir, redes que comparten la misma "configuración" con parámetros y pesos iguales. En este tipo de redes neuronales, se observan dos o más subredes trabajando en paralelo y generando salidas simultáneamente (34) (Figura 2.14).

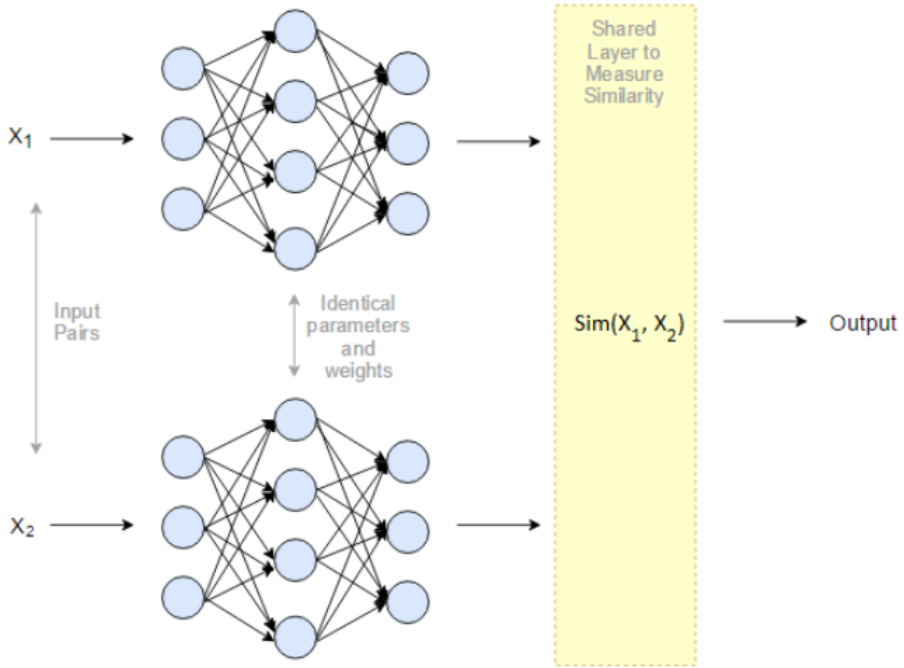

Figura 2.14: Arquitectura básica de una red neuronal siamesa. Se observan un par de entradas x_1 y x_2, dos redes neuronales idénticas y una función de similitud $sim(x_1, x_2)$. (Elaboración propia)

Estas salidas o vectores de representación deben pasar por alguna operación

que mida su similitud, en otras palabras, debemos verificar qué tan "parecidos" son los dos vectores. La verdadera cuestión aquí es la elección de una métrica adecuada.

Entendiendo el funcionamiento básico de una red neuronal siamesa y su relación con el aprendizaje por similitud, toca entender donde entran las redes neuronales convolucionales. Si nuestra tarea requiere procesar imágenes, la mejor opción es optar por utilizar redes neuronales convolucionales, ya que como se mencionó anteriormente, estas son ideales para su procesamiento y obtención de características.

Si se emplea una red neuronal siamesa con redes neuronales convolucionales como subredes (Figura 2.15), podemos recibir como entrada varias imágenes y encontrar la similitud existente entre estas de forma óptima. Dicho esto, no existe una "receta" para construir una red neuronal siamesa, las arquitecturas son diversas y dependen únicamente de qué tipos de tareas están destinadas a realizar.

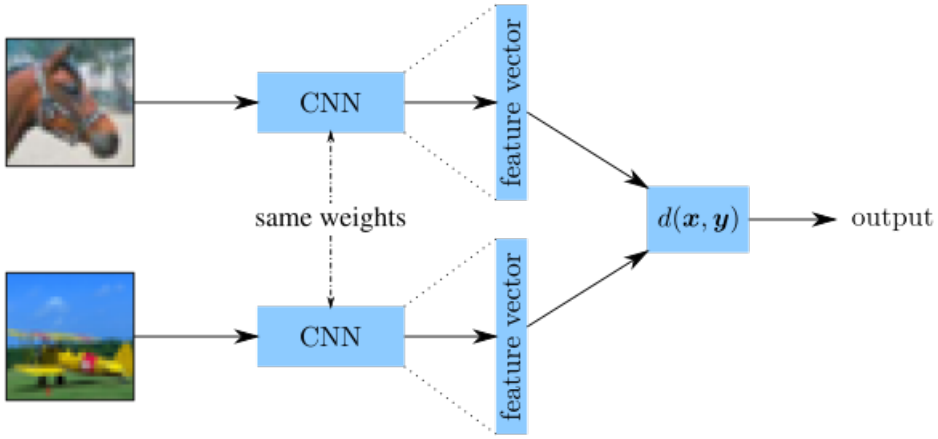

Figura 2.15: Arquitectura de una red neuronal siamesa con redes convolucionales como subredes. (Tomado de (24))

En muchos casos, las redes neuronales convolucionales deben pasar por un proceso de afinamiento, donde una o más capas se eliminan para garantizar que la salida sea un vector que contenga las características del objeto y no un vector de probabilidades de pertenencia a distintas clases (Figura 2.16).

Durante el entrenamiento, los pares de objetos son etiquetados como pares legítimos si ambos objetos pertenecen a la misma clase e impostores en caso contrario, es decir, que los objetos se encuentren en clases diferentes.

Es así como la red es capaz de generar un espacio multidimensional donde

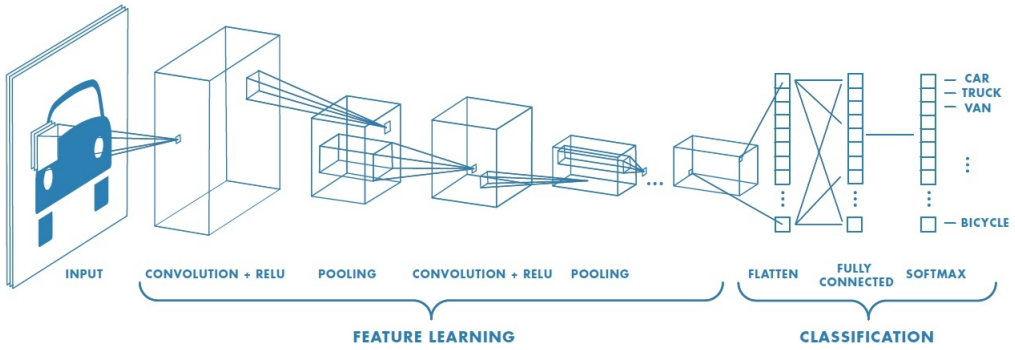

Figura 2.16: Arquitectura de una red neuronal convolucional, separando las capas de aprendizaje y las capas de clasificación. (Tomado de (10))

los pares legítimos se aproximan mientras que los pares impostores se alejan entre sí, básicamente, el objetivo del entrenamiento en estas redes es hallar los parámetros y pesos que permitan lograr que la distancia entre las categorías similares sea pequeña mientras que la distancia entre categorías distintas sea grande (13) (Figura 2.17).

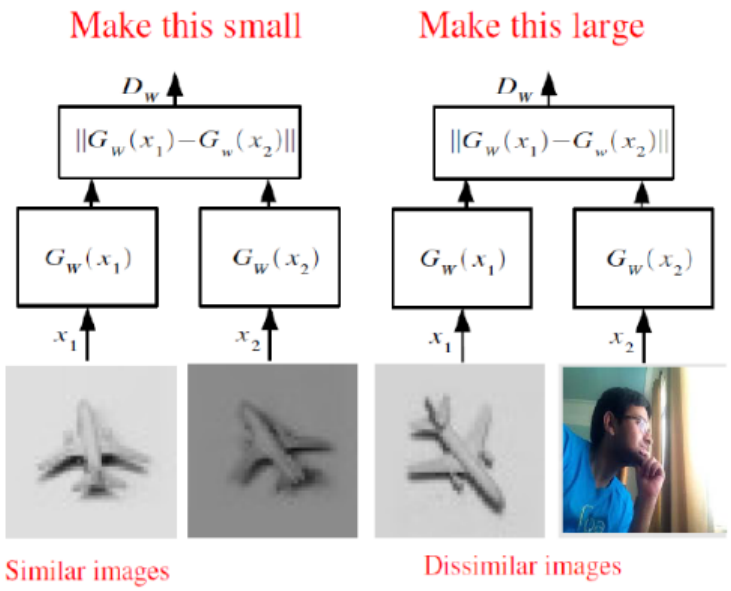

Figura 2.17: Entrenamiento de la red neuronal siamesa. Los objetos que pertenecen a la misma clase deben mantener una distancia pequeña entre ellos, en caso contrario, la distancia debe ser grande. (Tomado de (5))

Capítulo 3

Metodología

En este capítulo se muestran todas aquellas herramientas y técnicas utilizadas para la creación, desarrollo y validación de los distintos modelos que tienen como principal objetivo la clasificación satisfactoria de memes aleatorios. Incluye la creación del conjunto de datos, las arquitecturas de redes convolucionales empleadas y las funciones de pérdida.

3.1. Minado de tweets con *Tweepy*

Tweepy es una librería de Python cuya función es acceder a la API de Twitter, permitiendo realizar ciertas operaciones, entre ellas el minado de tweets. En este caso, el uso de Tweepy se limitó a la obtención del contenido de tweets de las cuentas ShitpostBot5000 y ShitpostContext (Figura 3.1).

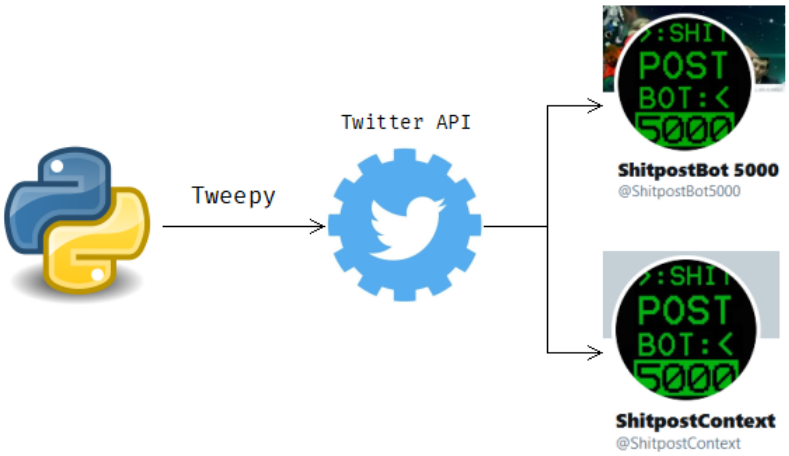

Figura 3.1: Proceso de minado de tweets con *Tweepy*. (Elaboración propia)

En cuanto a la cuenta de ShitpostBot5000 que es la cuenta encargada de postear los memes, entre más popular sea un tweet, más popular es el meme posteado en la comunidad, por lo que se puede decir que se trata de un meme gracioso o bien, que tiene cierto grado de sentido.

Caso contrario, la cuenta de ShitpostContext funciona como complemento de ShitpostBot5000, ya que, como su nombre indica, se encarga de postear las imágenes que componen cada meme, estas imágenes están almacenadas en la web de ShitpostBot5000 y son ajenas a twitter.

3.2. Descarga de imágenes con *requests*

Requests es una librería de Python capaz de enviar solicitudes HTTP/1.1, es utilizada para la obtención de datos provenientes de páginas web. En este caso, esta librería es utilizada para descargar los memes posteados por ShitpostBot500 de forma automática, así como las plantillas e imágenes de origen que lo componen.

Figura 3.2: Proceso de descarga de imágenes con *requests*. (Elaboración propia)

Con ayuda previa de Tweepy es posible obtener la url que redirecciona a la imagen del tweet correspondiente, lo que permite utilizar requests para enviar una petición de descarga de contenido, descargando así la imagen. El funcionamiento es idéntico para la descarga de las plantillas e imágenes de origen provenientes del servidor de ShitpostBot5000 (Figura 3.2).

3.3. Arquitecturas de redes neuronales convolucionales utilizadas

En esta sección se dará una breve introducción a las tres arquitecturas de redes neuronales convolucionales que constituyen las redes neuronales siamesas modeladas en este trabajo. El objetivo es dar ciertos datos tales como su rendimiento en el concurso ImageNet, así como desglosar su composición y tratar de explicar brevemente cada una se sus partes.

3.3.1. AlexNet

AlexNet es, probablemente, la red neuronal convolucional que más impacto ha tenido en el campo del aprendizaje profundo aplicado a la visión computacional. Fue desarrollada en 2012 por Alex Krizhevsky, Ilya Sutskever y Geoffrey E. Hinton, logrando una tasa de error de $15,3\,\%$ en la competición ImageNet LSVRC 2012 (14). El rendimiento de *AlexNet* fue tan destacado, que en solo $15,3\,\%$ de los casos no logró etiquetar de forma adecuada los objetos, es decir, no ubicó la etiqueta correcta dentro de las primeras cinco predicciones.

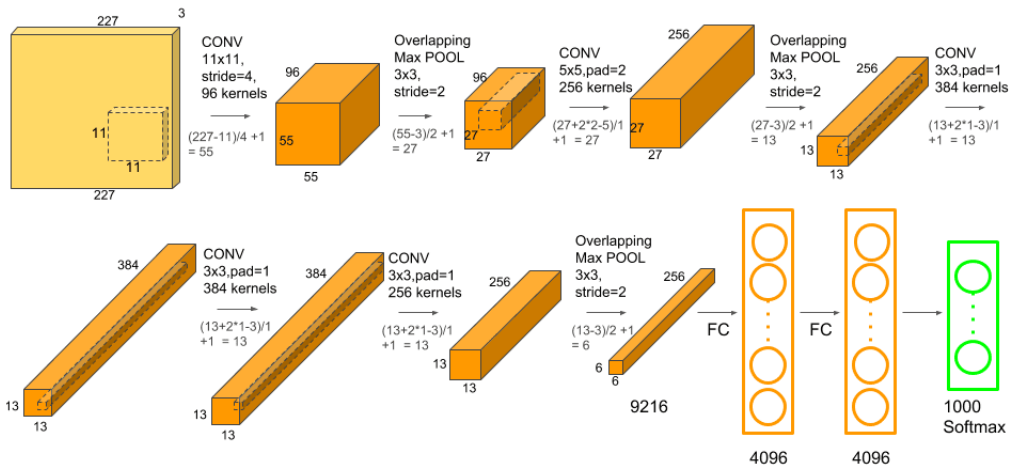

Figura 3.3: Arquitectura de *AlexNet*. (Tomado de (17))

AlexNet era mucho más grande que las redes neuronales convolucionales an-

teriores utilizadas para tareas de visión computacional. Tiene 60 millones de parámetros y 650 mil neuronas. Su arquitectura consta de cinco capas convolucionales, tres capas *max-pooling*, dos capas completamente conectadas y una capa *softmax* (Figura 3.3).

Las dos primeras capas convolucionales son seguidas por capas de *max-pooling*, La tercera, cuarta y quinta capas convolucionales están conectadas consecutivamente. La quinta capa convolucional va seguida de una capa *max-pooling*, cuya salida pasa a una serie de dos capas completamente conectadas, la salida de estas últimas capas pasa por una última capa *softmax* (17).

La función de activación ReLU se aplica después de todas las capas convolucionales y las capas completamente conectadas. La ReLU de la primera y la segunda capa de convolución es seguida por una capa de normalización local antes de llegar a la capa de *max-pooling*.

3.3.2. ResNet18

Fue en la competición ImageNet ILSVRC 2015, donde Kaiming He introdujo una arquitectura novedosa con conexiones de salto y una fuerte normalización de lotes. Estas conexiones de salto también se conocen como unidades cerradas o unidades recurrentes cerradas y tienen una gran similitud con elementos existentes en las redes neuronales recurrentes. Alcanza una tasa de error del $3,57\,\%$ (9), que supera el rendimiento del ojo humano.

ResNet se basa en una arquitectura conocida como red residual, que aplica el mapeo de identidad. Esto significa que la entrada a alguna capa se pasa a través de una conexión de salto a otra capa, siendo esto la definición de un bloque residual. Una red residual consta de varios bloques residuales básicos, no obstante, las operaciones realizadas en un bloque residual pueden variar dependiendo de las distintas arquitecturas de redes residuales.

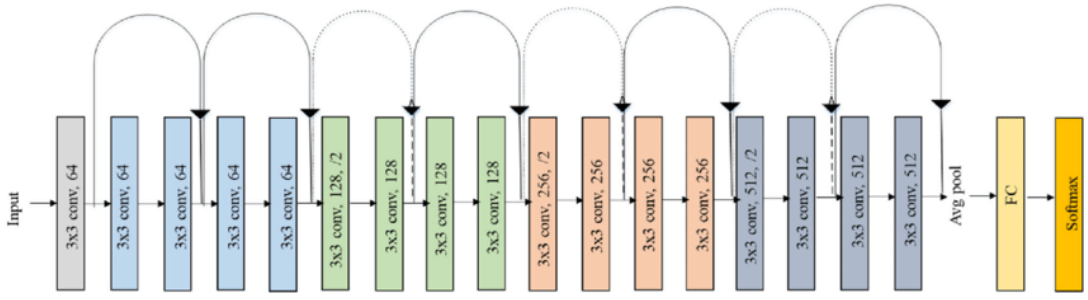

Figura 3.4: Arquitectura de *ResNet18*. (Tomado de (21))

Para el caso de *ResNet18*, comienza con una capa convolucional, justo después,

está el comienzo de la conexión de salto. La entrada de aquí se agrega a la salida proveniente de una capa *max-pooling* y dos capas convolucionales. Este es el primer bloque residual.

Luego, desde aquí, la salida de este bloque residual se agrega a la salida de dos capas convolucionales, constituyendo el segundo bloque residual. El tercer bloque residual se compone de la salida del segundo bloque a través de la conexión de salto y la salida de dos capas convolucionales.

El cuarto y último bloque residual comienza con la salida del tercer bloque y pasa a través de una conexión de salto y la salida de dos capas convolucionales. Finalmente, una capa de *average-pooling* se aplica a la salida del último bloque residual y el mapa de características creado es procesado por las capas completamente conectadas (9) (Figura 3.4).

3.3.3. VGG16

VGG16 es una red neuronal convolucional propuesta por K. Simonyan y A. Zisserman de la Universidad de Oxford. Fue creada en 2014 y participó en el concurso ImageNet ILSVRC 2014 mostrando tan solo un $7,3\,\%$ de error (29), consolidándose en el primer lugar.

La entrada de la red pasa por dos capas convolucionales, después, es procesada por una capa *max-pooling*. Continuando su camino, pasa por otras dos capas convolucionales seguidas de una capa *max-pooling*. Después de eso, hay tres conjuntos de tres capas convolucionales y una capa *max-pooling*.

Tres capas completamente conectadas están colocadas después de todos los conjuntos de capas convolucionales, la última capa es la capa *softmax* (Figura 3.4). La configuración de las capas completamente conectadas es la misma en todas las variaciones. Todas las capas ocultas van seguidas de la función de activación ReLU.

A pesar de tener un gran rendimiento, *VGG16* tiene dos problemas mayores: su entrenamiento es muy costoso y el modelo es bastante pesado.

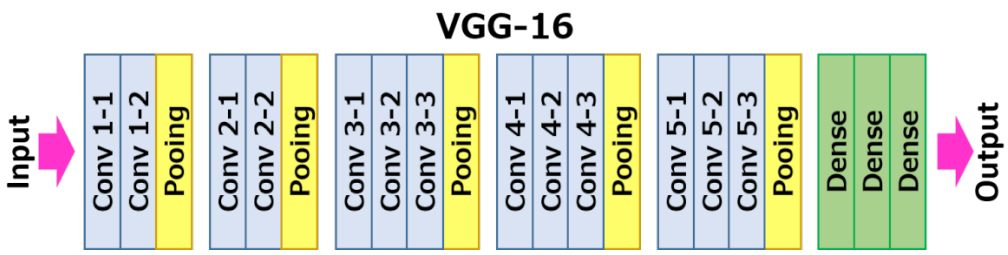

Figura 3.5: Arquitectura de *VGG16*. (Tomado de (19))

3.4. Funciones de pérdida empleadas

En esta sección, se detallarán a fondo las funciones de pérdida utilizadas en la construcción de los modelos correspondientes a este trabajo.

3.4.1. Pérdida de tripleta (*Triplet Margin Loss*)

Es una función de pérdida que utiliza una tripleta de entradas. Tenemos una entrada base o ancla (*anchor*) que se compara con una entrada positiva (*positive*) y una entrada negativa (*negative*). La distancia desde el ancla a la entrada positiva se minimiza mientras que, en caso contrario, la distancia entre el ancla y la entrada negativa se maximiza.

$$L(A, P, N) = max(dist(f(A) - f(P)) - dist(f(A) - f(N)) + \alpha, 0) \qquad (3.1)$$

En la ecuación anterior (Ecuación 3.1), α es un margen que se utiliza para estirar las diferencias de distancias y determina cuando estas se han vuelto lo suficientemente grandes para que el modelo deje de ajustar sus parámetros.

Durante el entrenamiento, una tripleta de objetos (objeto ancla, objeto positivo y objeto negativo) son las entradas de la red neuronal siamesa, cada una de estas pasa por la misma red neuronal y su respectivo vector de características es creado.

El objetivo es que la distancia entre el vector de características del objeto ancla y el objeto positivo sea menor que la distancia existente entre el vector de características del objeto ancla y el del objeto negativo (30) (Figura 3.6).

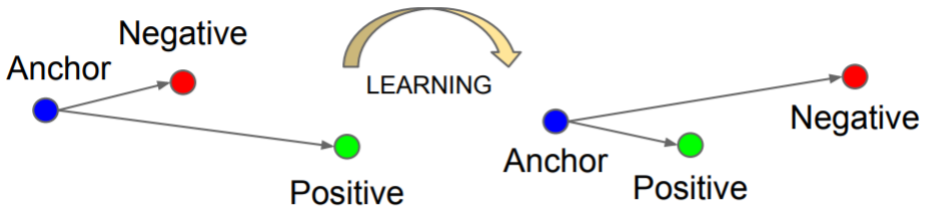

Figura 3.6: Proceso de entrenamiento con *Triplet Margin Loss*. (Tomado de (30))

3.4.1.1. Funciones de similitud

Una vez comprendido el funcionamiento de la pérdida de tripleta, debemos profundizar en un aspecto fundamental, que es, por supuesto, cómo medir la

similitud entre dos vectores. Aunque existen infinidad de funciones de similitud o distancia, son dos las que destacan por su simplicidad: la similitud de coseno y la distancia euclidiana.

Similitud de coseno

Es la medida de la similitud existente entre dos vectores en un espacio que posee un producto interior con el que se evalúa el valor del coseno del ángulo comprendido entre ellos (27). Proporciona un valor igual a 1 si el ángulo comprendido es cero, es decir, en caso de que ambos vectores sean paralelos. En caso contrario, el valor es -1 cuando ambos vectores apuntan a direcciones opuestas.

$$sim(x, y) = \frac{\sum_{i=1}^{n} x_i y_i}{\sqrt{\sum_{i=1}^{n} x_i}\sqrt{\sum_{i=1}^{n} y_i}} \tag{3.2}$$

El valor de esta función varía en un intervalo cerrado que va desde -1 a 1, se asume que dos vectores son similares cuando el coseno del ángulo comprendido entre estos es cercano a 1 (Figura 3.7).

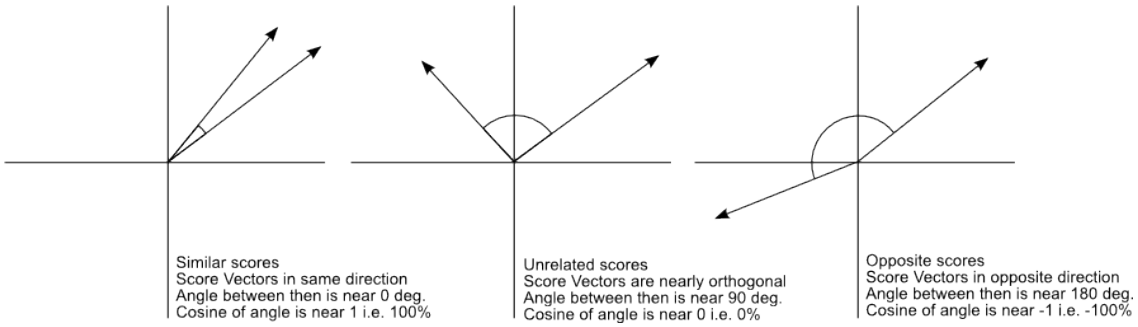

Figura 3.7: Valores de la similitud de coseno y su interpretación. (Tomado de (27))

Distancia euclidiana

La distancia euclidiana se deduce a partir del teorema de Pitágoras y se define como la distancia ordinaria entre dos puntos en un espacio euclídeo de n dimensiones. Esta función es la forma más sencilla de interpretar la distancia entre dos puntos, ya que implica el trazo de una línea recta entre estos (27).

$$sim(x, y) = \left(\sum_{i=1}^{n}(x_i - y_i)^2\right)^{\frac{1}{2}} \tag{3.3}$$

Una de las desventajas de esta función es que no varía en un intervalo cerrado, se dice que dos vectores p y q son similares cuando la distancia euclidiana d entre estos es cercana a 0 (Figura 3.8).

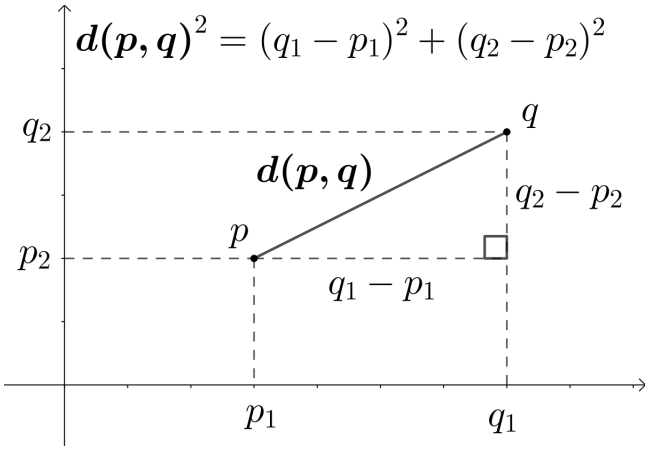

Figura 3.8: Representación gráfica de la distancia euclidiana. (Tomado de (27))

3.4.2. Pérdida de error cuadrático medio (*Mean Squared Error Loss*)

El error cuadrático medio es el criterio de evaluación más utilizado en problemas de regresión, indica el ajuste absoluto del modelo a los datos, es decir, que tan cerca están los puntos de datos observados de los valores predichos del modelo.

$$L(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 \tag{3.4}$$

Como se observa en la ecuación anterior (Ecuación 3.4), esta función de pérdida mide el promedio de los cuadrados de los errores, es decir, la diferencia cuadrática media entre los valores estimados $\hat{y}$ y el valor real y.

Como se mencionó antes, este es un problema de regresión, por lo que el objetivo del entrenamiento es hallar una recta o pendiente ideal que represente el mejor ajuste posible con el error cuadrático medio más pequeño, en palabras más simples, minimizar el error es minimizar el área de los cuadrados (Figuras 3.9 y 3.10).

Idealmente, el error cuadrático medio debe ser cercano a 0, lo cual nos indica que hemos llegado a un ajuste casi perfecto, es decir, los valores predichos por el modelo son iguales a los valores reales u observados.

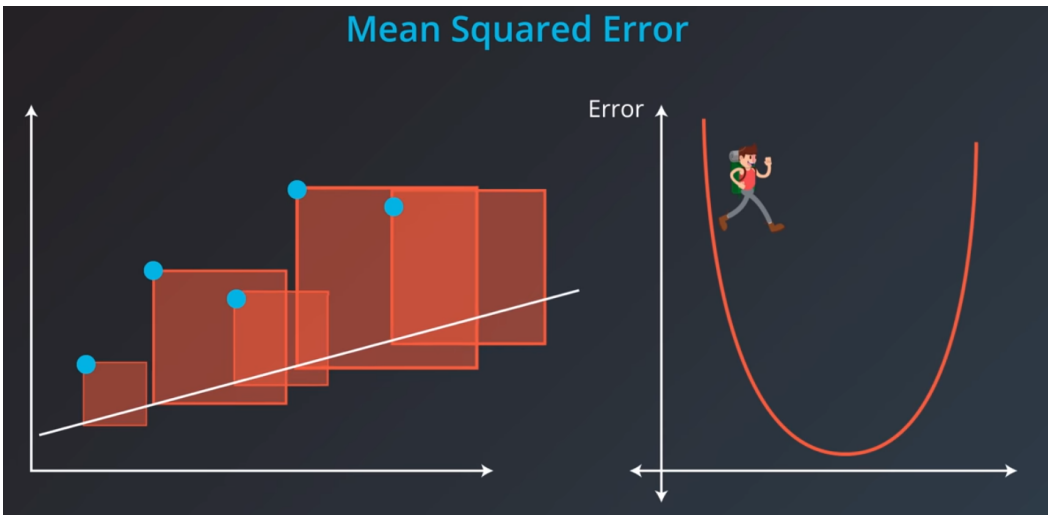

Figura 3.9: Error cuadrático medio al principio del entrenamiento. (Tomado de (4))

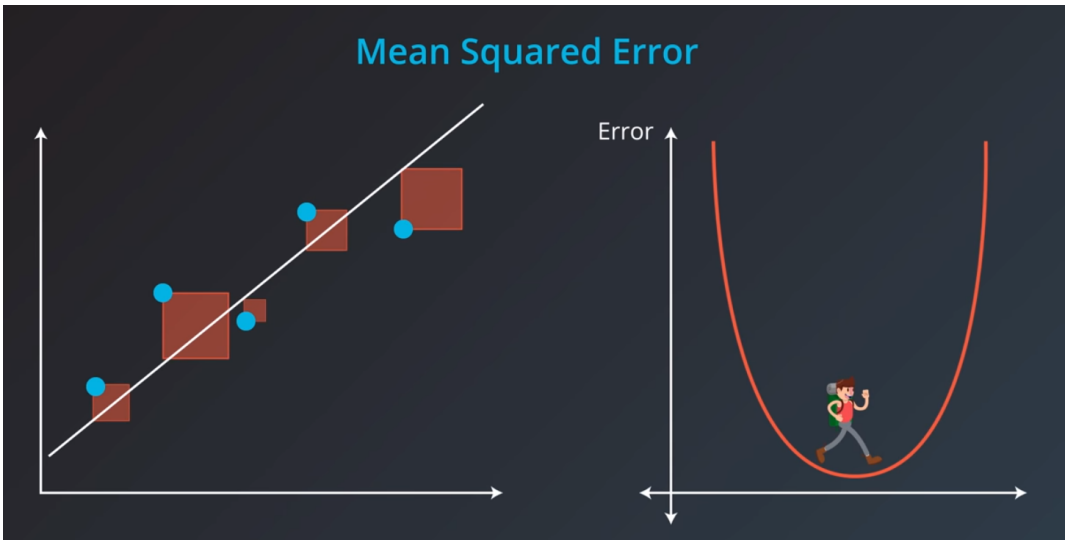

Figura 3.10: Error cuadrático medio al final del entrenamiento. (Tomado de (4))

Capítulo 4

Experimentación y análisis de resultados

En este capítulo se detallarán los experimentos realizados con distintas arquitecturas de redes neuronales siamesas implicando el uso de redes neuronales convolucionales como sus respectivas subredes. El objetivo es observar y analizar el desempeño de estas arquitecturas, además de comparar los diversos funcionamientos, tomando en cuenta las diferencias existentes entre las subredes utilizadas.

4.1. Estructura y conjunto de datos

Evidentemente, cuando hablamos de entrenar y validar el funcionamiento de redes neuronales, toca profundizar en qué datos estamos utilizando y cómo estos están distribuidos. Como ya fue mencionado anteriormente, nuestros datos fueron obtenidos de la cuenta de twitter de ShitpostBot5000.

Gracias a Tweepy, se pueden descomponer los tweets en sus metadatos, tales como su identificador único, número de likes, texto del tweet, url de multimedia, y las urls correspondientes a las plantillas e imágenes de origen (Figura 4.1).

Se construyó un dataframe con ayuda de la librería *pandas*, dicho dataframe contenía la información de todos aquellos tweets de ShitpostBot5000 que sobrepasarán la cantidad de 750 likes, dejando este límite como el "umbral de popularidad" mínimo que se busca para que un meme sea lo suficientemente gracioso.

Con ayuda del dataframe anterior, se generó un dataset conformado de todos los memes obtenidos anteriormente, dividiéndolos en dos secciones: aquellos que se utilizan para entrenar la red neuronal (training) y aquellos que se utilizaron para la validación de esta (testing). De forma empírica se optó por utilizar un 80 % de los datos para el entrenamiento y un 20 % para la validación (Figuras 4.2 y 4.3).

Figura 4.1: Estructura de un tweet. (Elaboración propia)

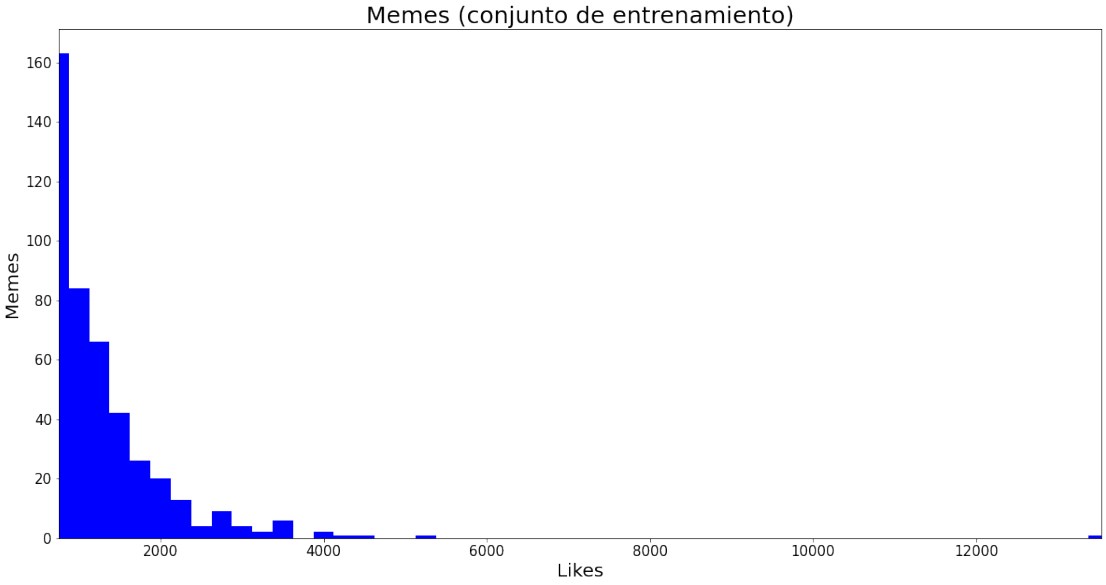

Figura 4.2: Histograma del conjunto de datos de entrenamiento.

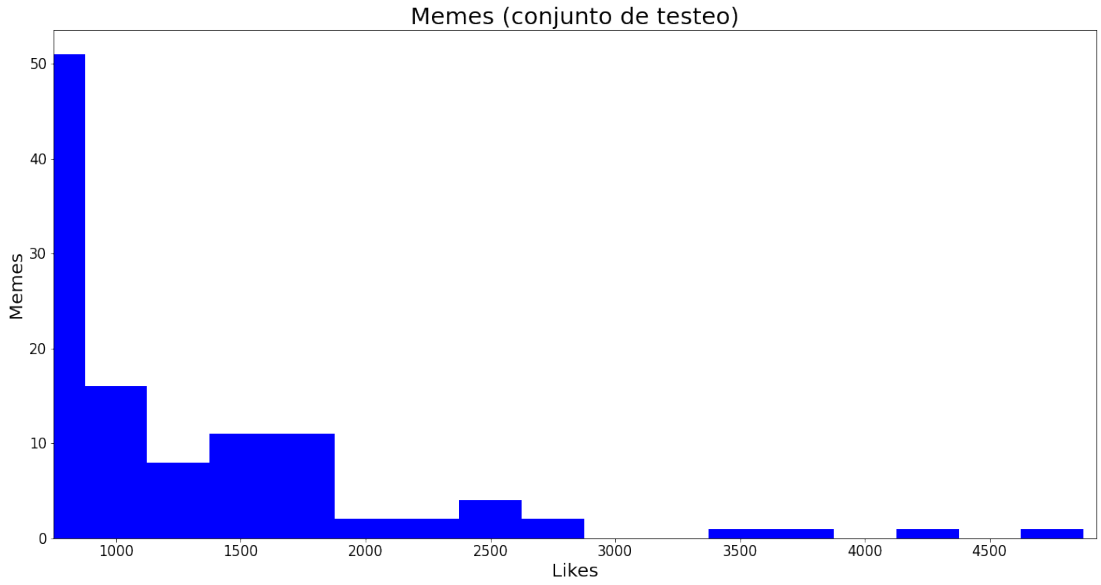

Figura 4.3: Histograma del conjunto de datos de testeo o verificación.

4.2. Red neuronal siamesa con tripleta de entradas (*Triplet Margin Loss*)

El principal objetivo de la utilización de este enfoque es la construcción de un espacio donde los memes puedan relacionarse a partir del número de likes que tuvieron, es decir, agruparlos de tal manera que los memes que tuvieron una gran cantidad de likes se encuentren lejos de aquellos que tuvieron una popularidad baja.

Para lograr la construcción de este modelo, hace falta utilizar una arquitectura "modificada" de la red neuronal siamesa, conocida como una red neuronal de tripleta. Durante el entrenamiento, se cargan tres imágenes: un *meme ancla*, un *meme positivo* y un *meme negativo* (Figura 4.4).

El meme ancla es aquel en que basamos la iteración de entrenamiento, el meme positivo debe tener un número similar de likes y el meme negativo debe tener un número de likes lejano.

Es así, como después de varias iteraciones, el modelo es capaz de "acercar" los memes con un número de likes similar y "alejar" a aquellos con un número de likes distintos (Figura 4.5). Estos son procesados por la misma red neuronal convolucional y a su vez, su vector de características es generado para que pase

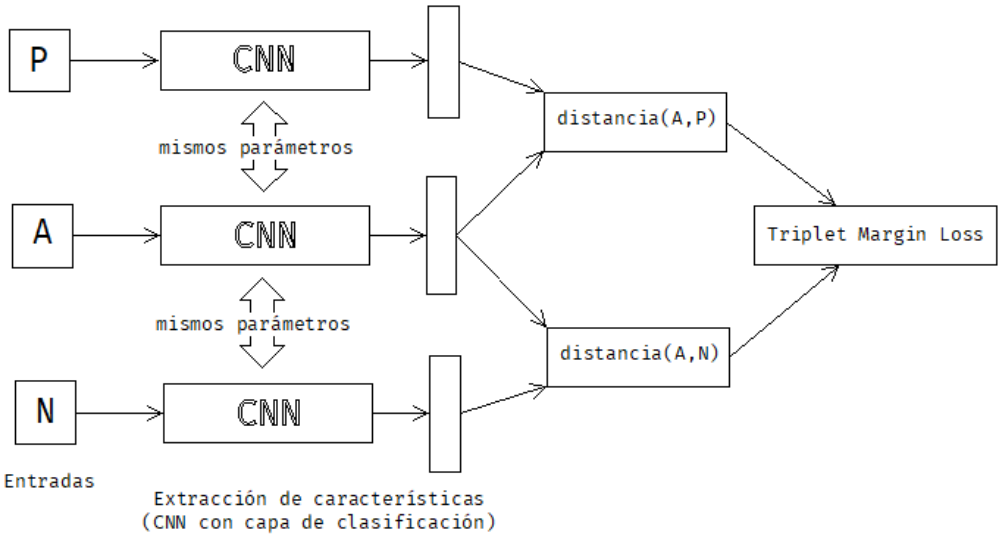

Figura 4.4: Arquitectura de la red neuronal siamesa con tripleta de entradas.

por la función de pérdida, en este caso *Triplet Margin Loss*.

Es importante mencionar que, en esta arquitectura, las subredes cuentan con sus respectivas capas de clasificación o capas completamente conectadas. Se optó por construirla así, ya que, el objetivo es modificar los parámetros de estas últimas durante el entrenamiento, en otras palabras, se busca modificar las capas completamente conectadas y dejar intactas aquellas que se encargan de la extracción de características.

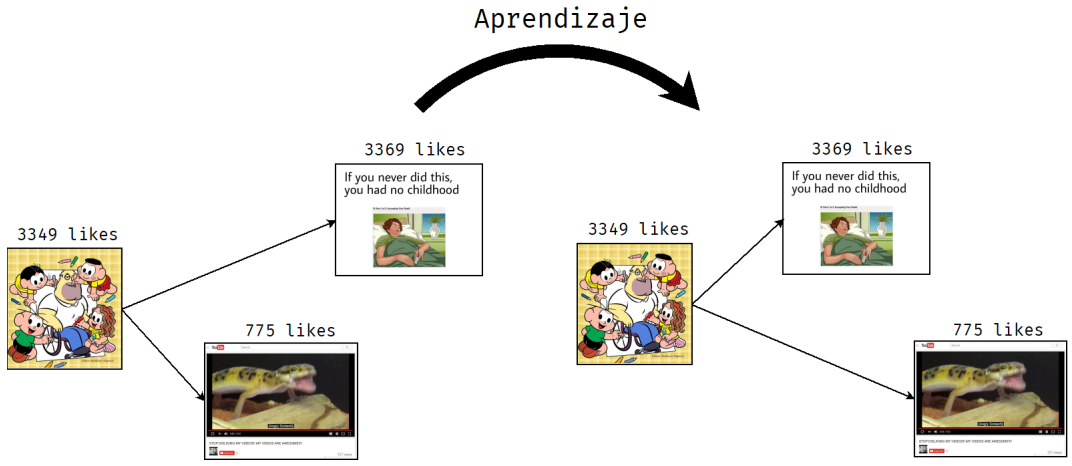

Figura 4.5: Proceso de aprendizaje con *Triplet Margin Loss*.

Debido a que esta arquitectura requiere calcular las distancias existentes entre

vectores, se deben elegir funciones o métricas de distancia adecuadas para nuestro propósito, por lo que se optó por utilizar dos: la distancia euclidiana (Ecuación 3.3) y la similitud de coseno (Ecuación 3.2).

Para predecir los likes de un meme, se deben encontrar aquellos memes del conjunto de entrenamiento más cercanos en el espacio, es decir, aquellos con los que comparta la distancia más corta. Una vez encontrados, solo queda promediar el número de likes de estos últimos. La elección de cuantos memes "cercanos" tomar es totalmente empírica, pero con 10 es más que suficiente.

4.2.1. Distancia euclidiana

4.2.1.1. Entrenamiento

Para el entrenamiento del modelo con la distancia euclidiana como métrica de distancia, se optó por tomar un margen α de 1. La idea de este experimento es crear un espacio donde la distancia euclidiana entre memes con un número de likes similar sea reducida, y en caso contrario, cuando dos memes tengan cantidades de likes diferente o lejanas, asegurarse de que la distancia sea grande.

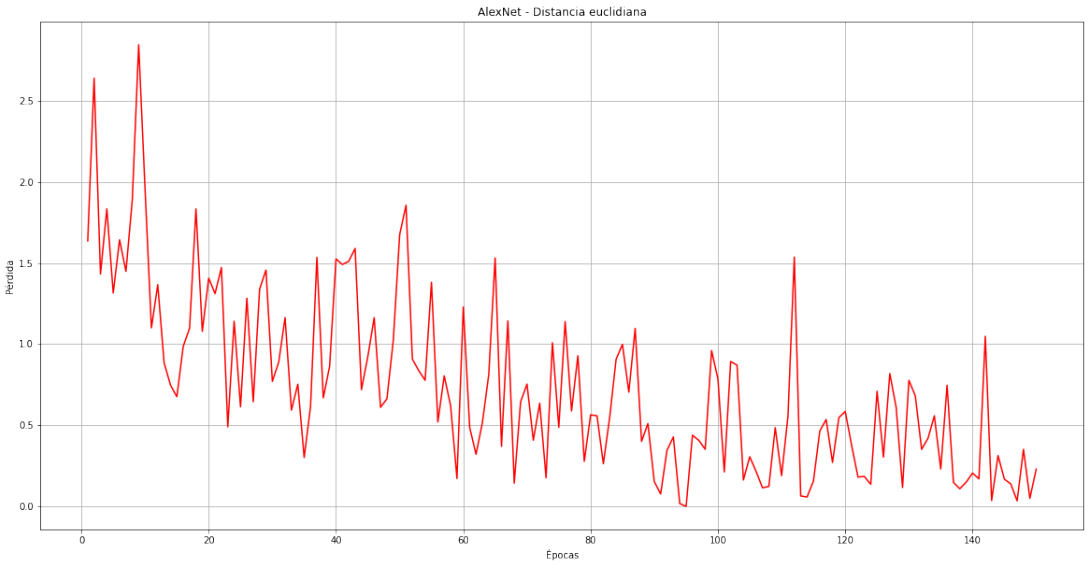

Figura 4.6: Entrenamiento del modelo con *AlexNet* y distancia euclidiana.

Evidentemente, lo que se busca es que conforme pasan las épocas de entrenamiento, la pérdida disminuya de forma gradual, se observa que *AlexNet* (Figura 4.6) dio las mejores impresiones, ya que mostró una clara disminución de pérdi-

da. En cuanto a *ResNet* (Figura 4.7) y *VGG16* (Figura 4.8), se observó que la pérdida se mantuvo constante con algunas variaciones.

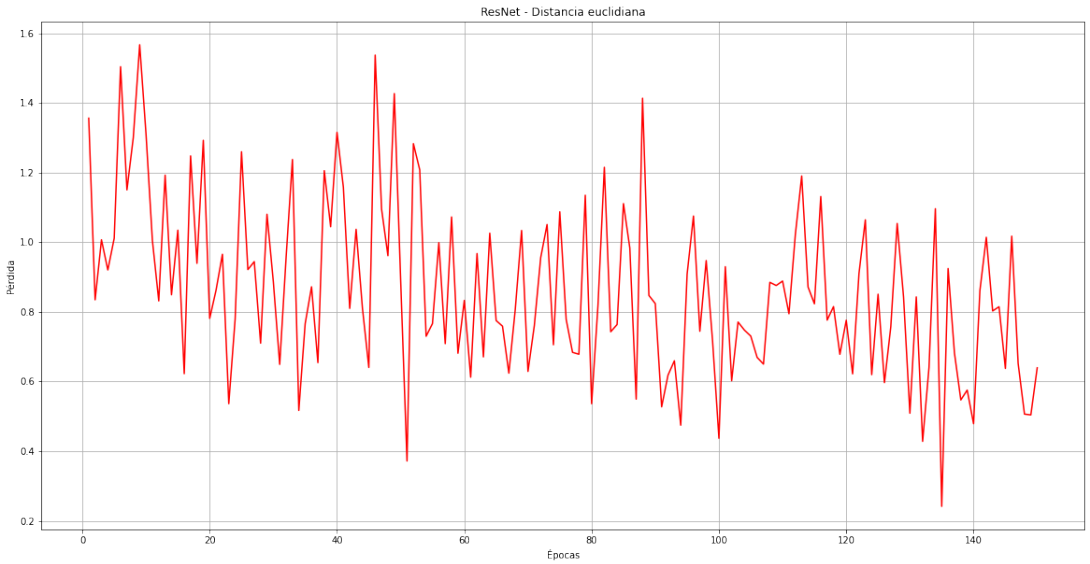

Figura 4.7: Entrenamiento del modelo con *ResNet* y distancia euclidiana.

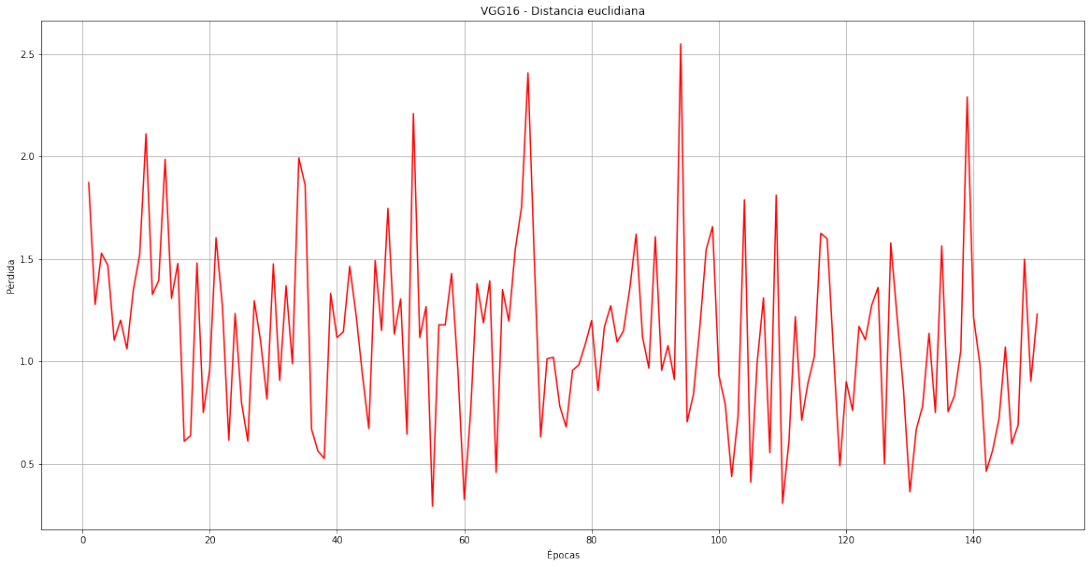

Figura 4.8: Entrenamiento del modelo con *VGG16* y distancia euclidiana.

Algo que hay que destacar es que *ResNet* y *VGG16* mantuvieron una pérdida constante en un valor mayor a 1, igual al margen, esto significa que la distancia

entre los memes similares no está siendo reducida o bien, el modelo no logra ajustar sus parámetros de forma óptima.

Para observar los elementos del conjunto de entrenamiento en el espacio, se utilizaron las herramientas de UMAP y TSNE, que permitieron reducir la dimensionalidad de los vectores para poder observarlos de forma más clara.

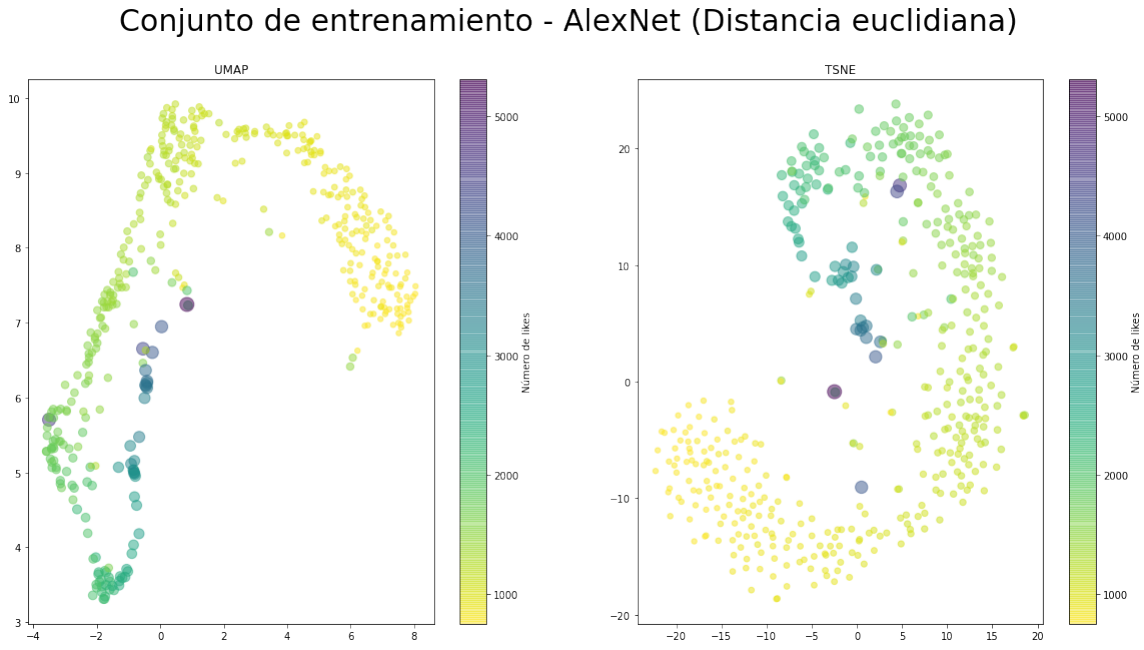

Figura 4.9: Representación del conjunto de entrenamiento (*AlexNet*).

Una vez observados los elementos del conjunto de entrenamiento representados por las distintas subredes ya entrenadas, se dedujo que, de forma gráfica, *AlexNet* (Figura 4.9) logró una separación más clara de los elementos, ya que fue capaz de separar satisfactoriamente los memes a partir de su número de likes. En cuanto a *ResNet* (Figura 4.10), se observaron los elementos desordenados y mezclados unos con los otros, no parece haber existido una definición de conjuntos distintos.

VGG16 (Figura 4.11) mostró un comportamiento muy curioso, ya que pareció haber logrado crear conjuntos más o menos definidos, pero hay un problema, plos memes más populares se encuentran en un punto medio, cuando debieron estar colocados a distancias lejanas de los demás.

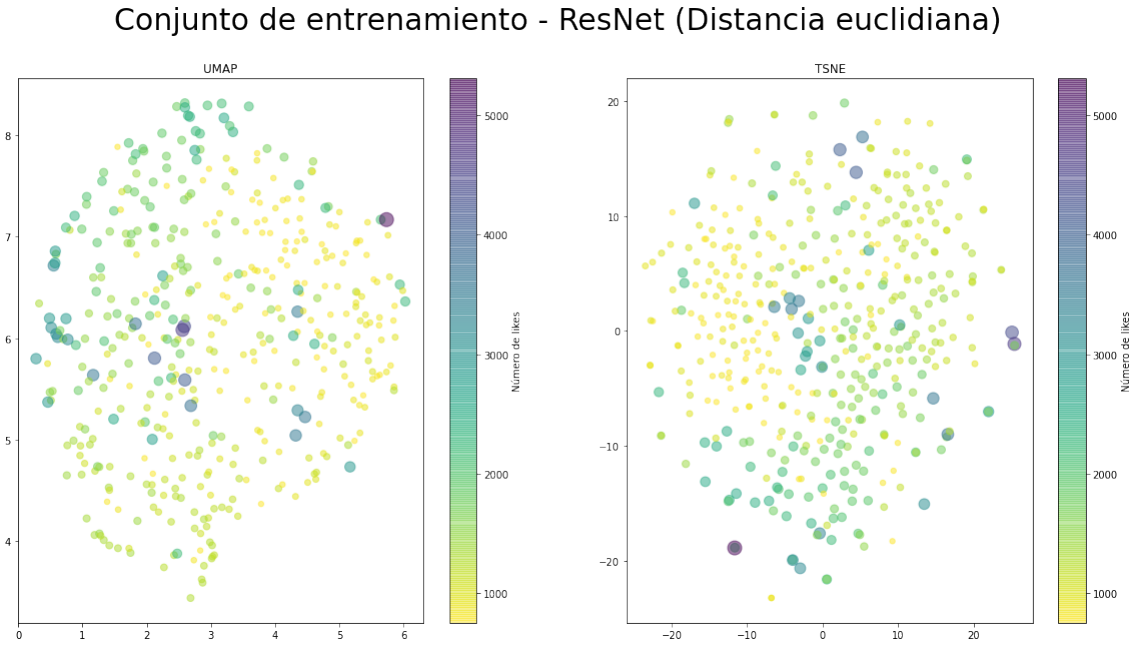

Figura 4.10: Representación del conjunto de entrenamiento (*ResNet*).

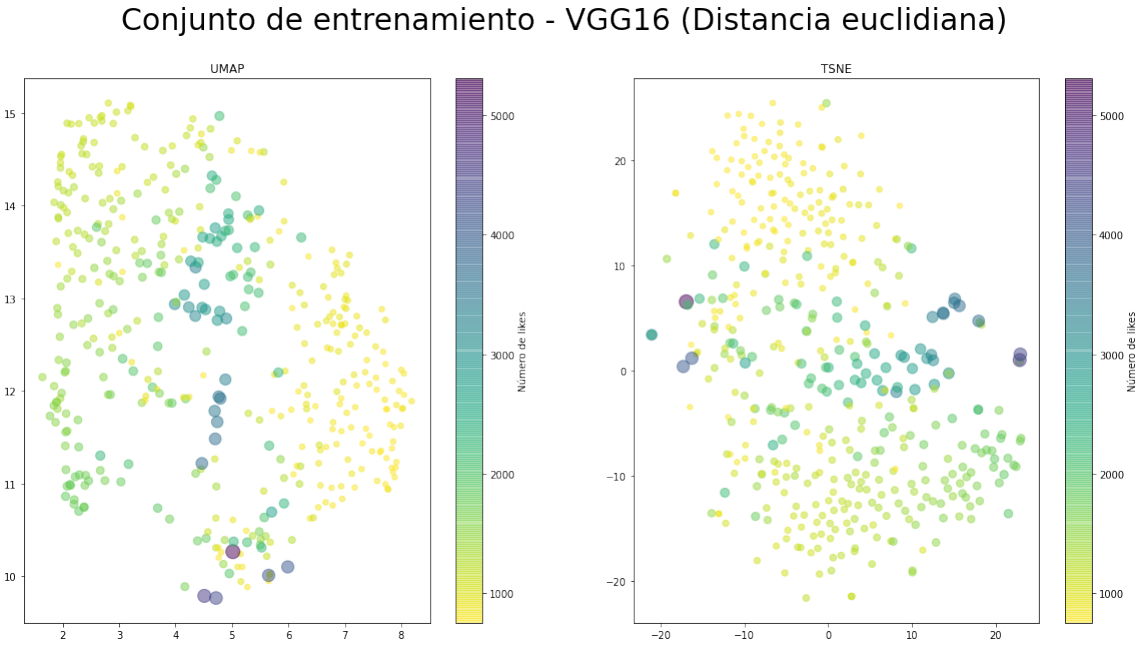

Figura 4.11: Representación del conjunto de entrenamiento (*VGG16*).

4.2.1.2. Verificación

Después de observar el desempeño de los modelos anteriores en la fase de entrenamiento, se realizaron pruebas de verificación para ver qué tan bien se comporta el modelo ante datos que no conoce. Es para esta fase que se utilizó el conjunto de testeo o verificación creado anteriormente.

Observando los elementos del conjunto de verificación gráficamente, se dedujo que tanto *AlexNet* (Figura 4.12), *ResNet* (Figura 4.13) y *VGG16* (Figura 4.14) no lograron clasificar o separar de forma adecuada los memes. Están completamente mezclados y no parece existir un indicio de construcción de conjuntos.

En cuanto a los estadísticos correspondientes, las tres arquitecturas de CNN mostraron cifras muy similares (Tablas 4.1 y 4.2), ninguna destacó especialmente. En cuanto a la precisión, es decir, de cuantos memes tuvieron una predicción de likes en un rango satisfactorio, *VGG16* es la que mostró un mejor comportamiento o más bien, más "equilibrado" en los distintos rangos de likes (Tabla 4.3).

La precisión de *VGG16* resultó ser más uniforme en los distintos rangos de likes, además de mostrar el mejor rendimiento en dos de los tres rangos. Sin embargo, se vio superada por *ResNet* en el segundo rango de likes.

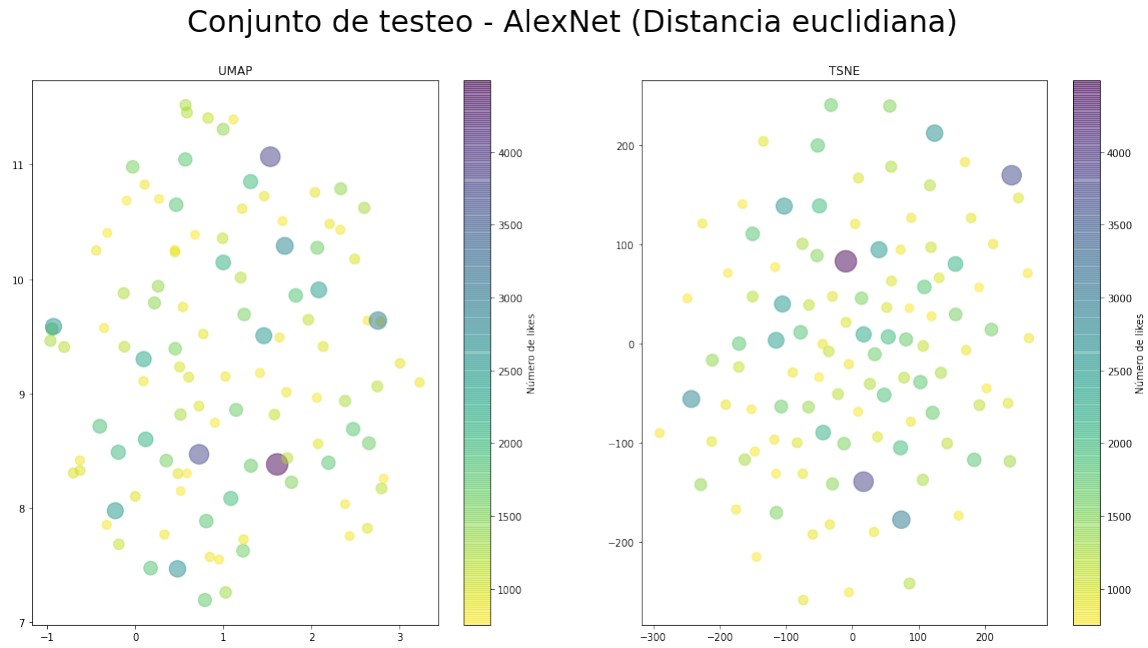

Figura 4.12: Representación del conjunto de verificación (*AlexNet*).

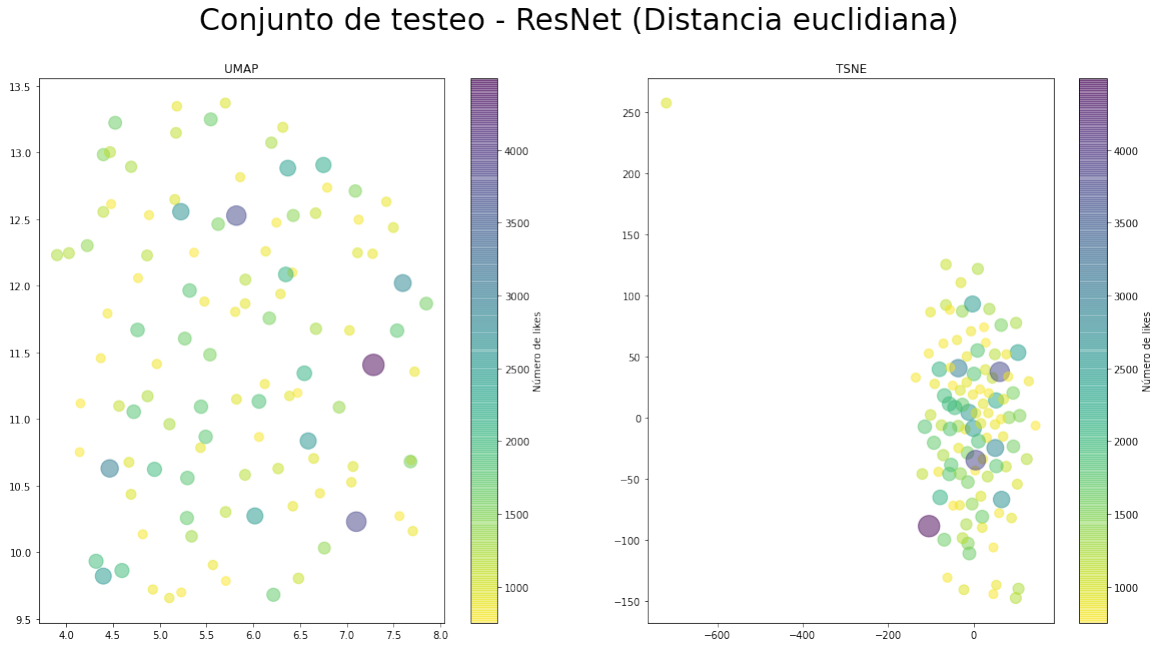

Figura 4.13: Representación del conjunto de verificación (*ResNet*).

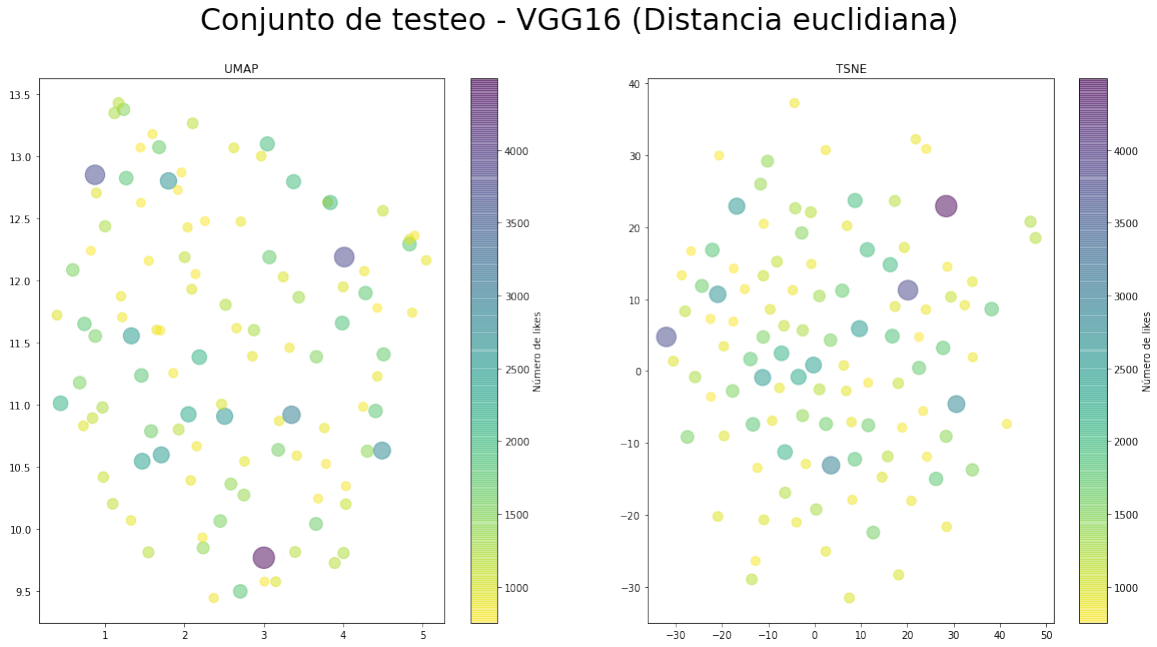

Figura 4.14: Representación del conjunto de verificación (*VGG16*).

Error cuadrático medio (ECM)				
CNN	750 a 1,000 likes	1,001 a 1,500 likes	1,501 a 2,000 likes	2,001 +
AlexNet	242,542.65	136,068.96	317,240.18	3,847,021.64
ResNet	254,890.71	77,829.33	296,586.45	4,134,475.64
VGG16	182,378.47	151,213.96	358,086.32	4,025,909.36

Tabla 4.1: Error cuadrático medio de la red neuronal siamesa con tripleta de entradas y distancia euclidiana.

Raíz del error cuadrático medio (RMSE)				
CNN	750 a 1,000 likes	1,001 a 1,500 likes	1,501 a 2,000 likes	2,001 +
AlexNet	492.49	368.88	563.24	1,961.38
ResNet	504.87	278.98	544.6	2,033.34
VGG16	427.06	388.86	598.4	2,006.47

Tabla 4.2: Raíz del error cuadrático medio de la red neuronal siamesa con tripleta de entradas y distancia euclidiana.

Precisión				
CNN	750 a 1,000 likes	1,001 a 1,500 likes	1,501 a 2,000 likes	2,001 +
AlexNet	0.16	0.29	0.14	0.00
ResNet	0.16	0.38	0.09	0.00
VGG16	0.20	0.33	0.14	0.00

Tabla 4.3: Precisión de la red neuronal siamesa con tripleta de entradas y distancia euclidiana.

4.2.1.3. Resultados visuales (memes del conjunto de verificación y su mejor pareja del conjunto de entrenamiento)

Figura 4.15: Memes y su pareja más cercana (*AlexNet* y distancia euclidiana).

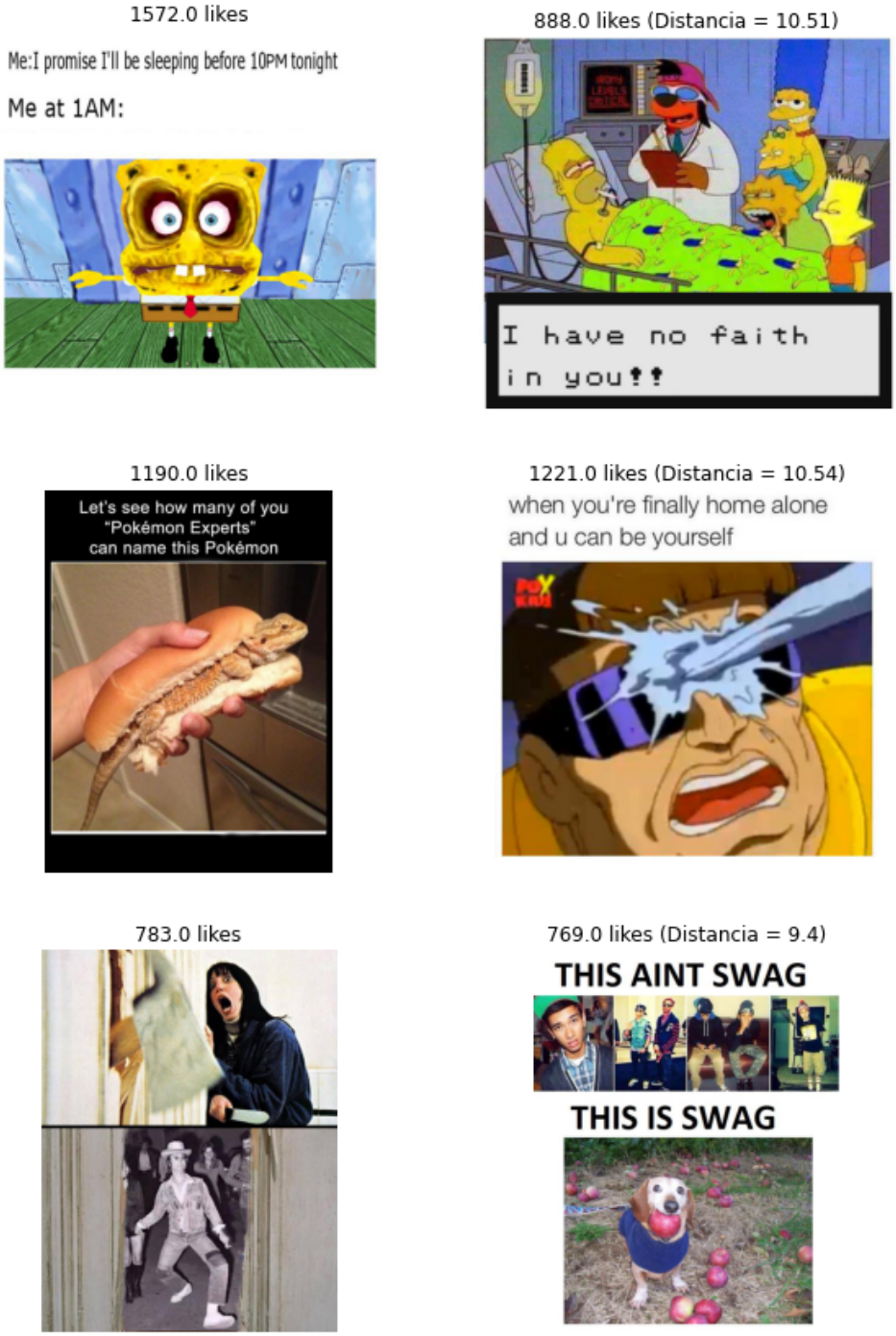

Figura 4.16: Memes y su pareja más cercana (*ResNet* y distancia euclidiana).

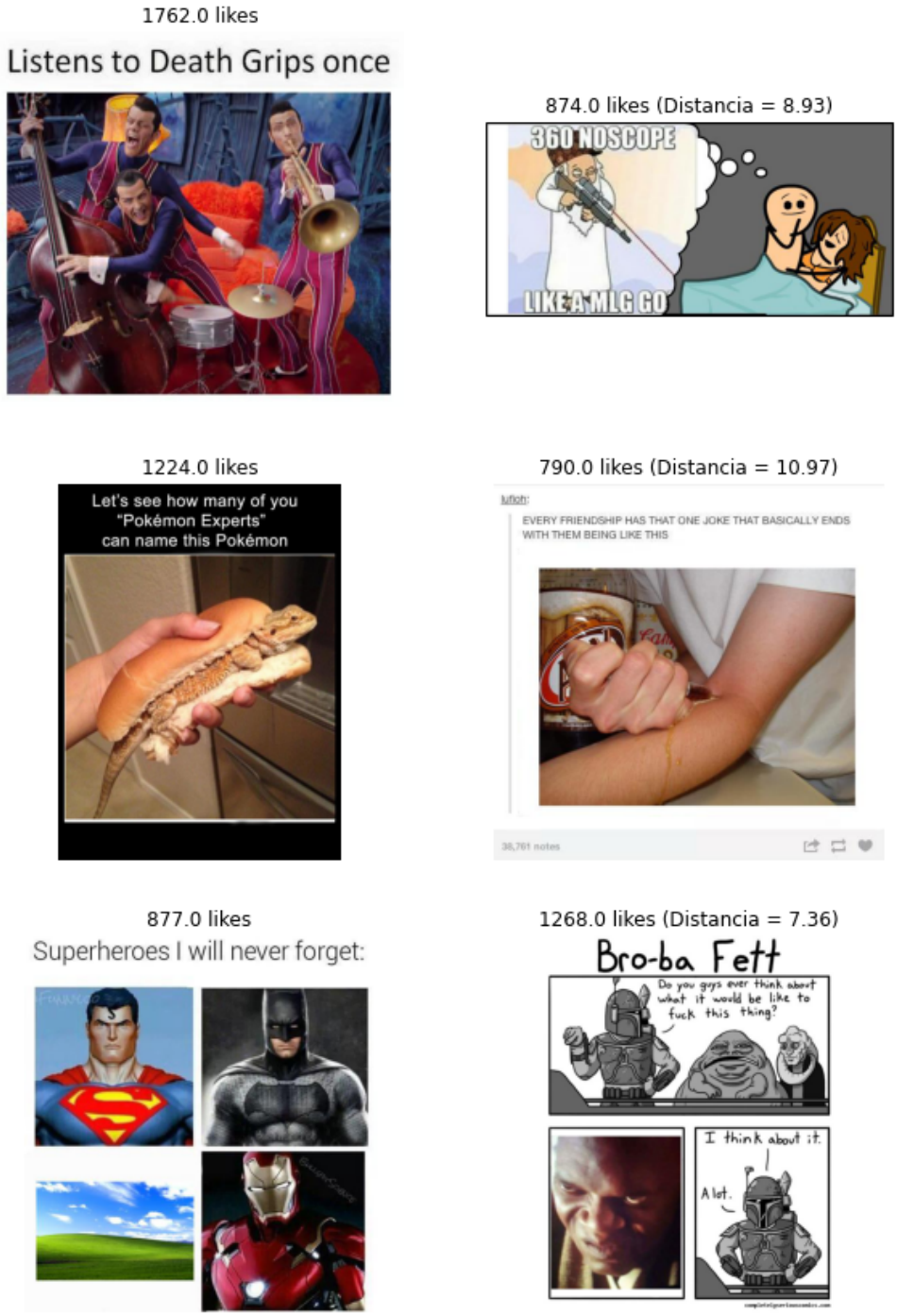

Figura 4.17: Memes y su pareja más cercana (*VGG16* y distancia euclidiana).

4.2.2. Similitud de coseno

4.2.2.1. Entrenamiento

El siguiente experimento consistió en tomar como métrica de distancia a la similitud de coseno, para el entrenamiento se decidió tomar un margen α de $\frac{\sqrt{3}}{2}$, representando el coseno de 30°.

En este caso, el entrenamiento busca que los vectores que representen memes con una popularidad similar formen un ángulo de 0°, es decir, que los vectores sean paralelos. Ahora, si existen dos vectores que representen memes con popularidad dispar, estos deberán formar un ángulo cada vez más grande dependiendo de la diferencia existente entre la cantidad de likes de estos, en otras palabras, entre más diferencia exista en la cantidad de likes, más grande será el ángulo existente entre ellos.

Se observó que la pérdida disminuye gradualmente con las tres arquitecturas utilizadas. Tanto *AlexNet* (Figura 4.18), *ResNet* (Figura 4.19) y *VGG16* (Figura 4.20) mostraron buenos indicios respecto a la pérdida. Sin embargo, *AlexNet* fue la que presentó menor pérdida, ya que al final del entrenamiento se mantuvo constante en 0.2.

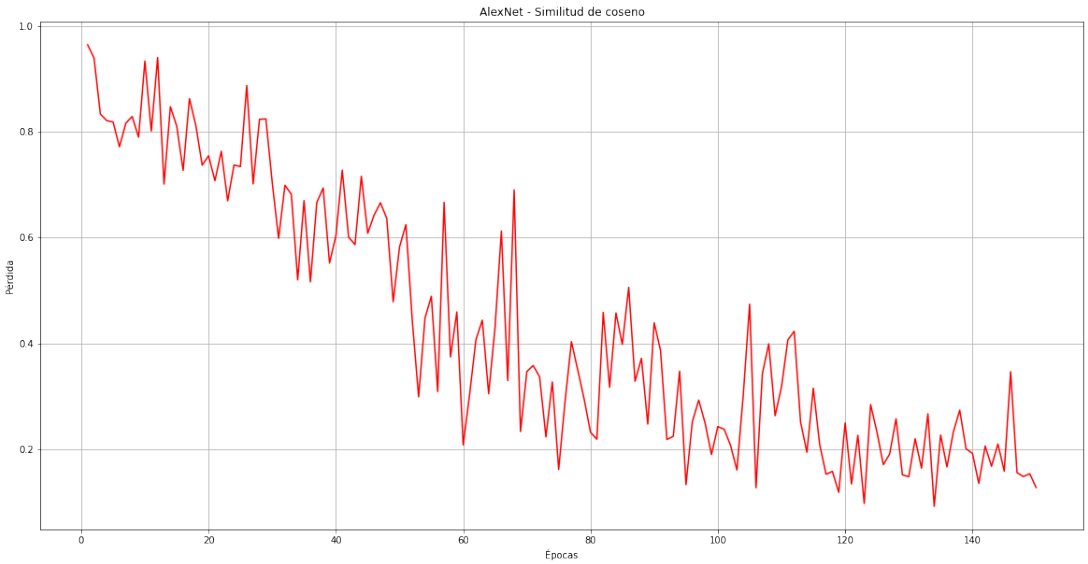

Figura 4.18: Entrenamiento del modelo con *AlexNet* y similitud de coseno.

ResNet (Figura 4.22) y *VGG16* (Figura 4.23) lograron separar de manera satisfactoria dos conjuntos, uno representando a los elementos con menos likes y otro representando a los elementos con un número de likes en el rango intermedio.

Sin embargo, los elementos con la popularidad más alta se agruparon entre los dos conjuntos mencionados anteriormente.

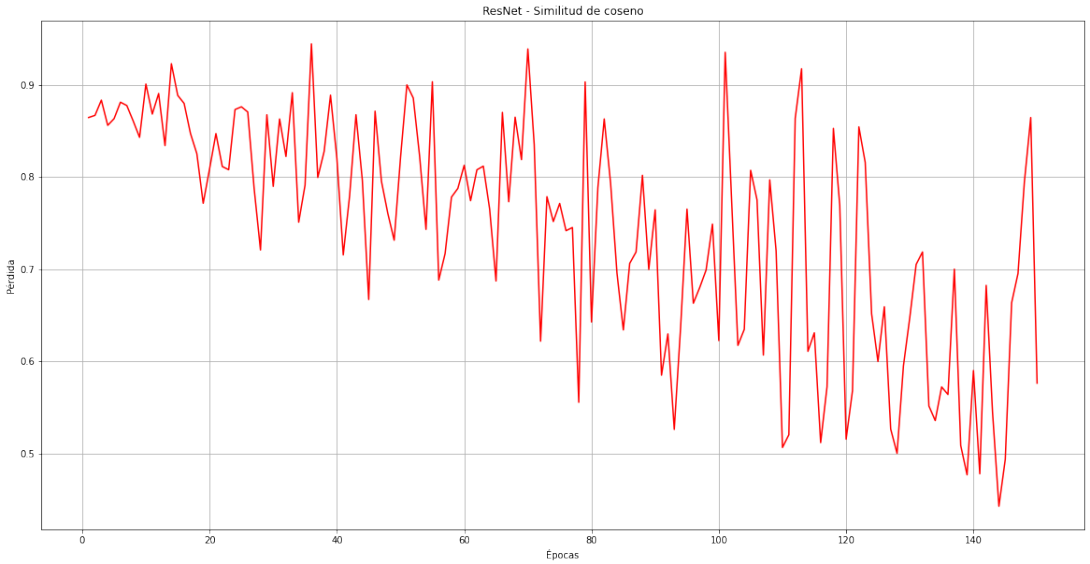

Figura 4.19: Entrenamiento del modelo con *ResNet* y similitud de coseno.

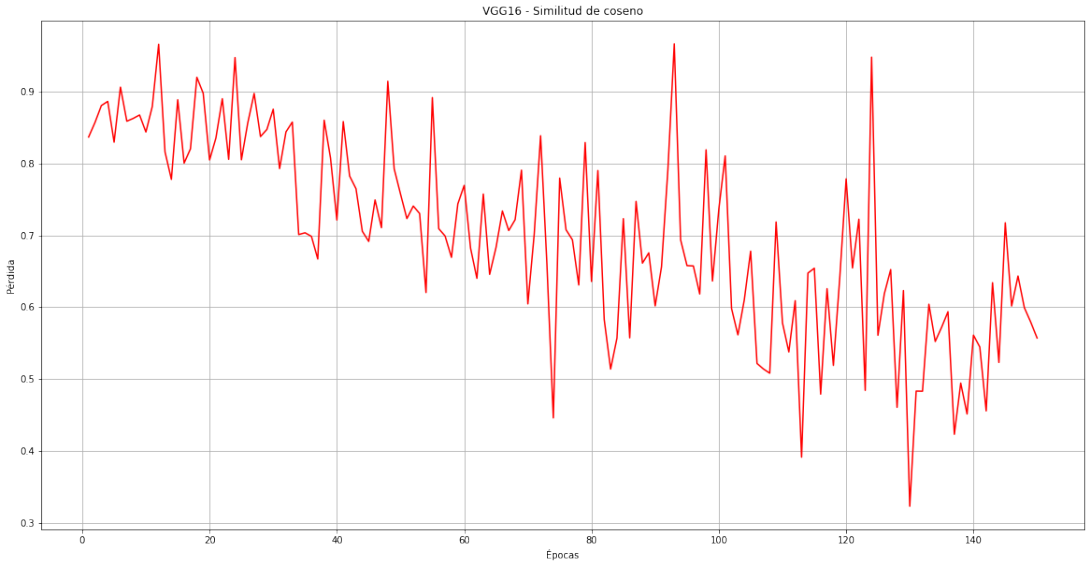

Figura 4.20: Entrenamiento del modelo con *VGG16* y similitud de coseno.

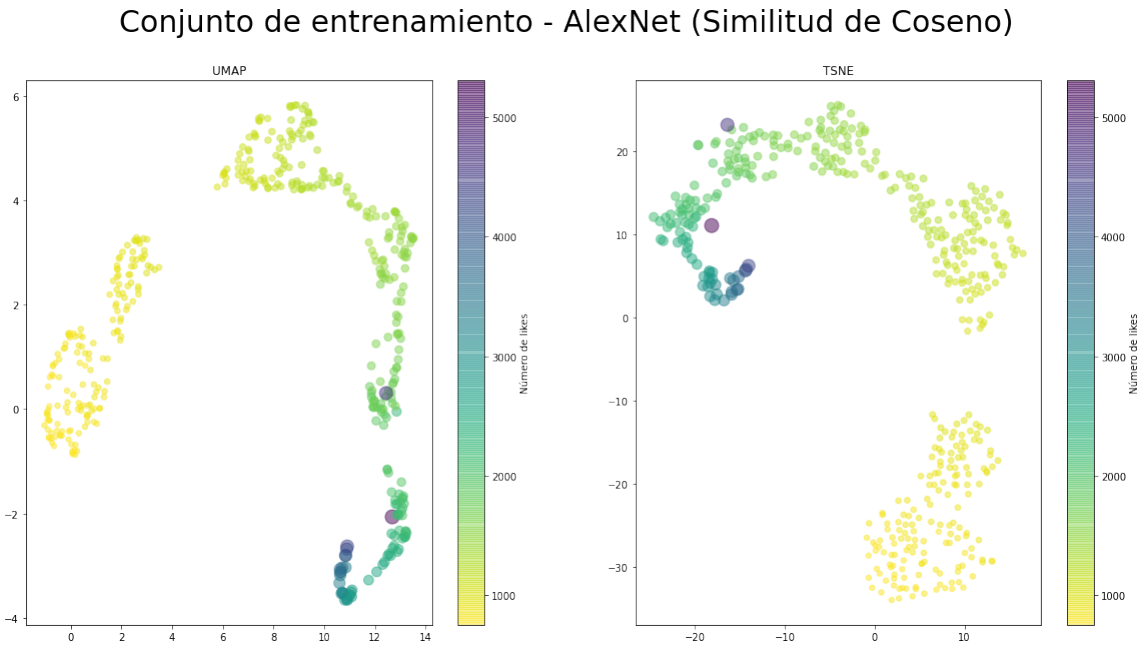

Figura 4.21: Representación del conjunto de entrenamiento (*AlexNet*).

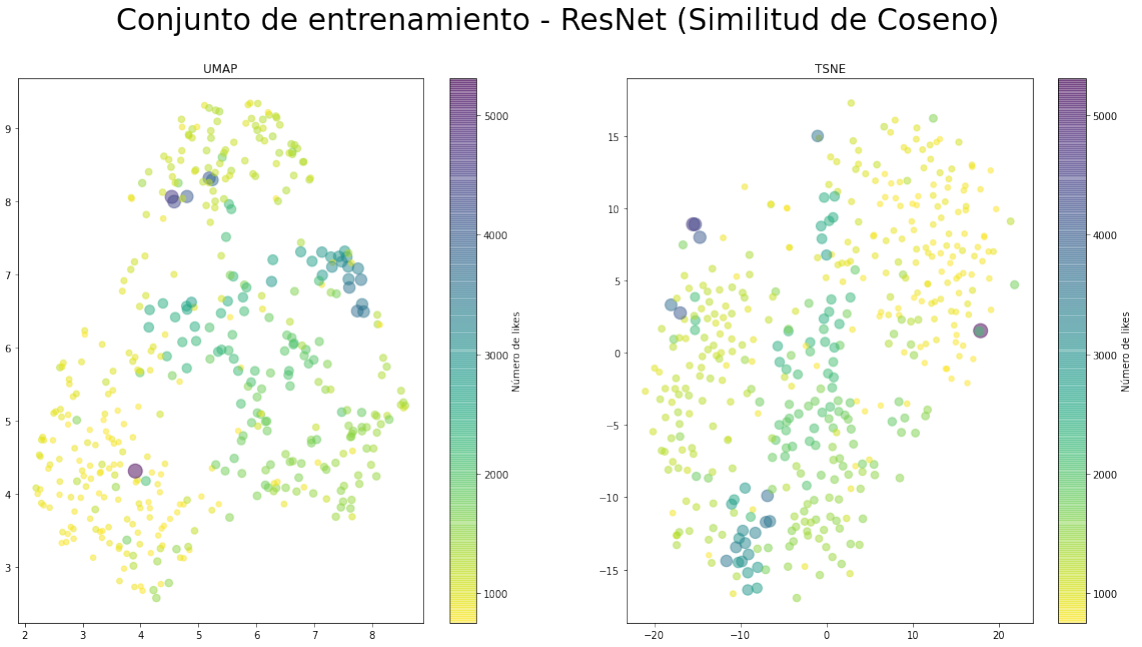

Figura 4.22: Representación del conjunto de entrenamiento (*ResNet*).

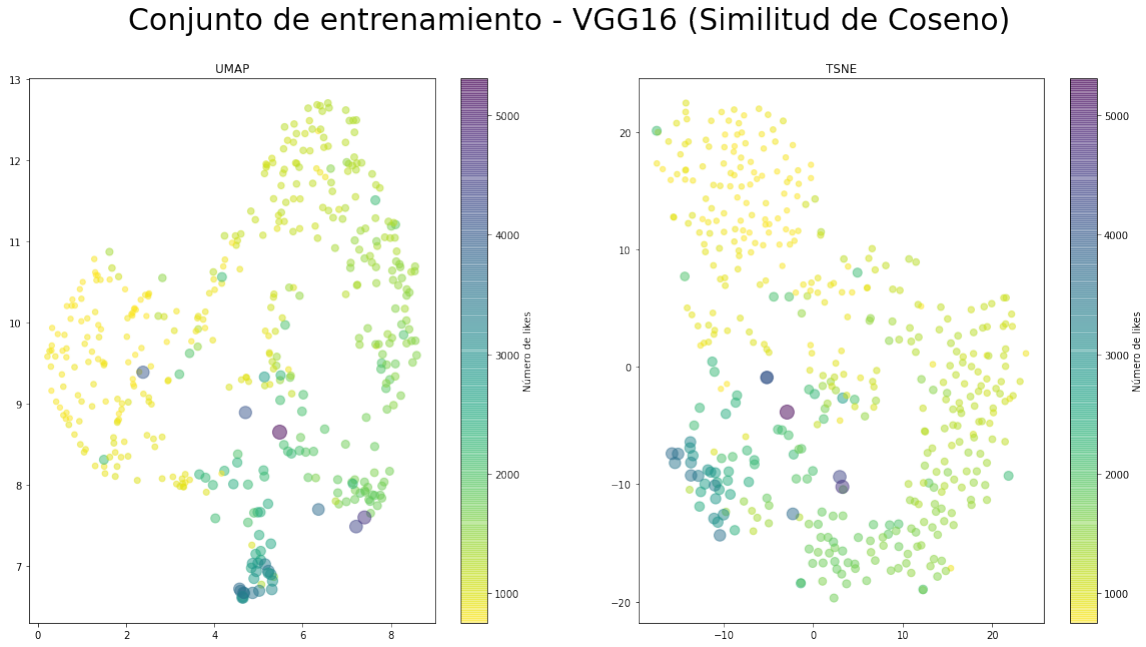

Figura 4.23: Representación del conjunto de entrenamiento (*VGG16*).

AlexNet (Figura 4.21) tuvo el mejor comportamiento, ya que el entrenamiento logró crear y separar de manera satisfactoria conjuntos de elementos. Gráficamente, se observó claramente un camino marcado desde los memes con menos likes hasta aquellos que tuvieron mayor éxito.

4.2.2.2. Verificación

En cuanto a la representación de los elementos del conjunto de verificación en el espacio, se hace evidente que no existió ningún tipo de agrupación o criterio para formar conjuntos, es decir, no fueron capaces de reaccionar de manera adecuada ante elementos que desconoce (Figuras 4.24, 4.25 y 4.26).

Continuando con el análisis del error cuadrático medio (Tabla 4.4) y la raíz del error cuadrático medio (Tabla 4.5), se observó que no existió una arquitectura que haya destacado especialmente por su rendimiento. Aunque alguna arquitectura mostró superioridad en rangos definidos, no es concluyente.

Hablando de precisión en la predicción de likes (Tabla 4.6), todas las arquitecturas mostraron un éxito similar, ya que *AlexNet* mostró superioridad en el primer rango, mientras que *ResNet* dominó el segundo rango.

VGG16 no destacó especialmente en ningún rango, sin embargo, sus cifras en los dos primeros rangos fueron bastante buenas respecto a sus competidoras.

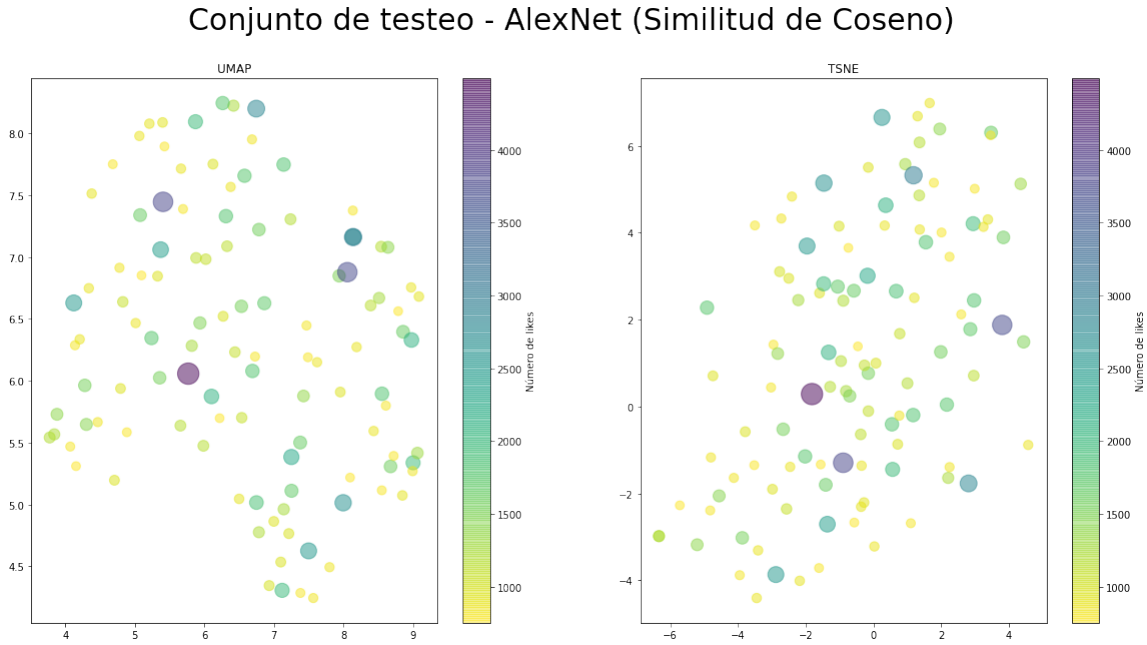

Figura 4.24: Representación del conjunto de verificación (*AlexNet*).

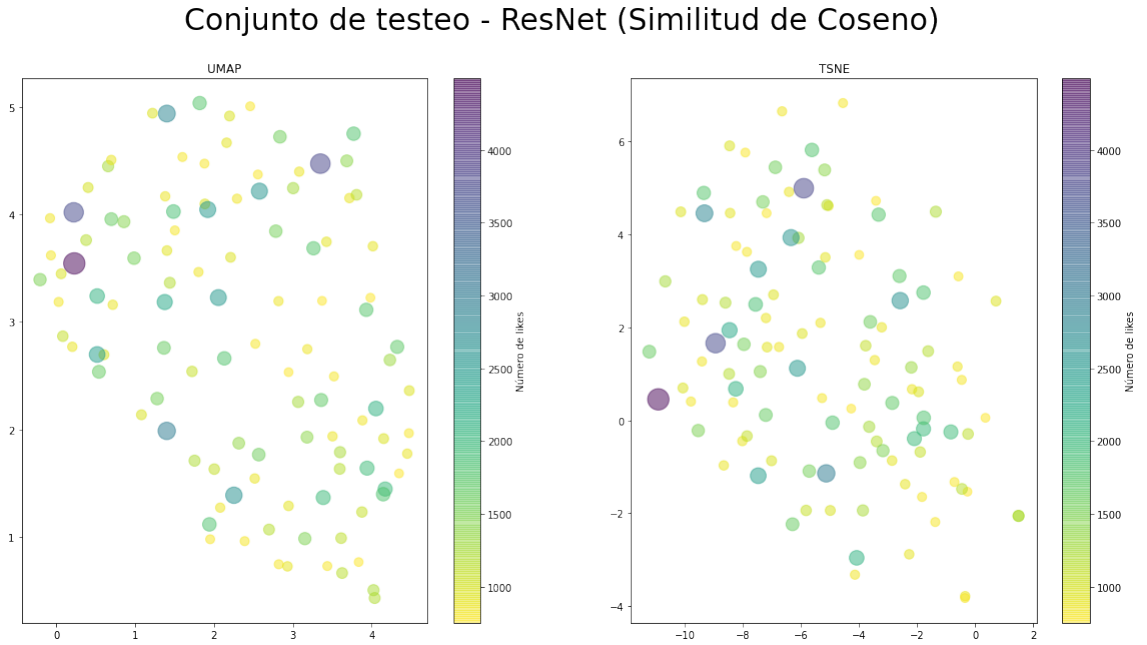

Figura 4.25: Representación del conjunto de verificación (*ResNet*).

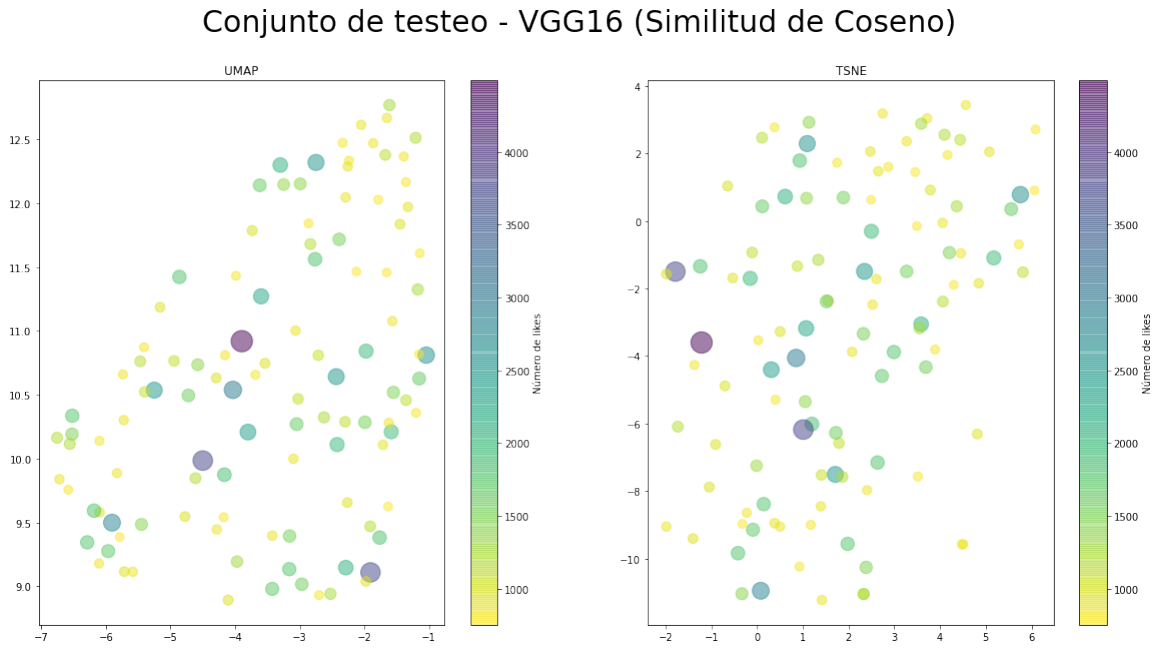

Figura 4.26: Representación del conjunto de verificación (*VGG16*).

Error cuadrático medio (ECM)				
CNN	750 a 1,000 likes	1,001 a 1,500 likes	1,501 a 2,000 likes	2,001 +
AlexNet	530,953.86	689,129.58	381,964.09	4,187,837.93
ResNet	841,733.61	172,894.0	266,153.73	3,404,201.5
VGG16	373,650.92	243,714.96	418,361.82	3,394,718.29

Tabla 4.4: Error cuadrático medio de la red neuronal siamesa con tripleta de entradas y similitud de coseno.

Raíz del error cuadrático medio (RMSE)				
CNN	750 a 1,000 likes	1,001 a 1,500 likes	1,501 a 2,000 likes	2,001 +
AlexNet	728.67	830.14	618.03	2,046.42
ResNet	917.46	415.81	515.9	1,845.05
VGG16	611.27	493.67	646.81	1,842.48

Tabla 4.5: Raíz del error cuadrático medio de la red neuronal siamesa con tripleta de entradas y similitud de coseno.

Precisión				
CNN	750 a 1,000 likes	1,001 a 1,500 likes	1,501 a 2,000 likes	2,001 +
AlexNet	0.29	0.17	0.23	0.00
ResNet	0.12	0.29	0.18	0.07
VGG16	0.25	0.25	0.14	0.07

Tabla 4.6: Precisión de la red neuronal siamesa con tripleta de entradas y similitud de coseno.

4.2.2.3. Resultados visuales (memes del conjunto de verificación y su mejor pareja del conjunto de entrenamiento)

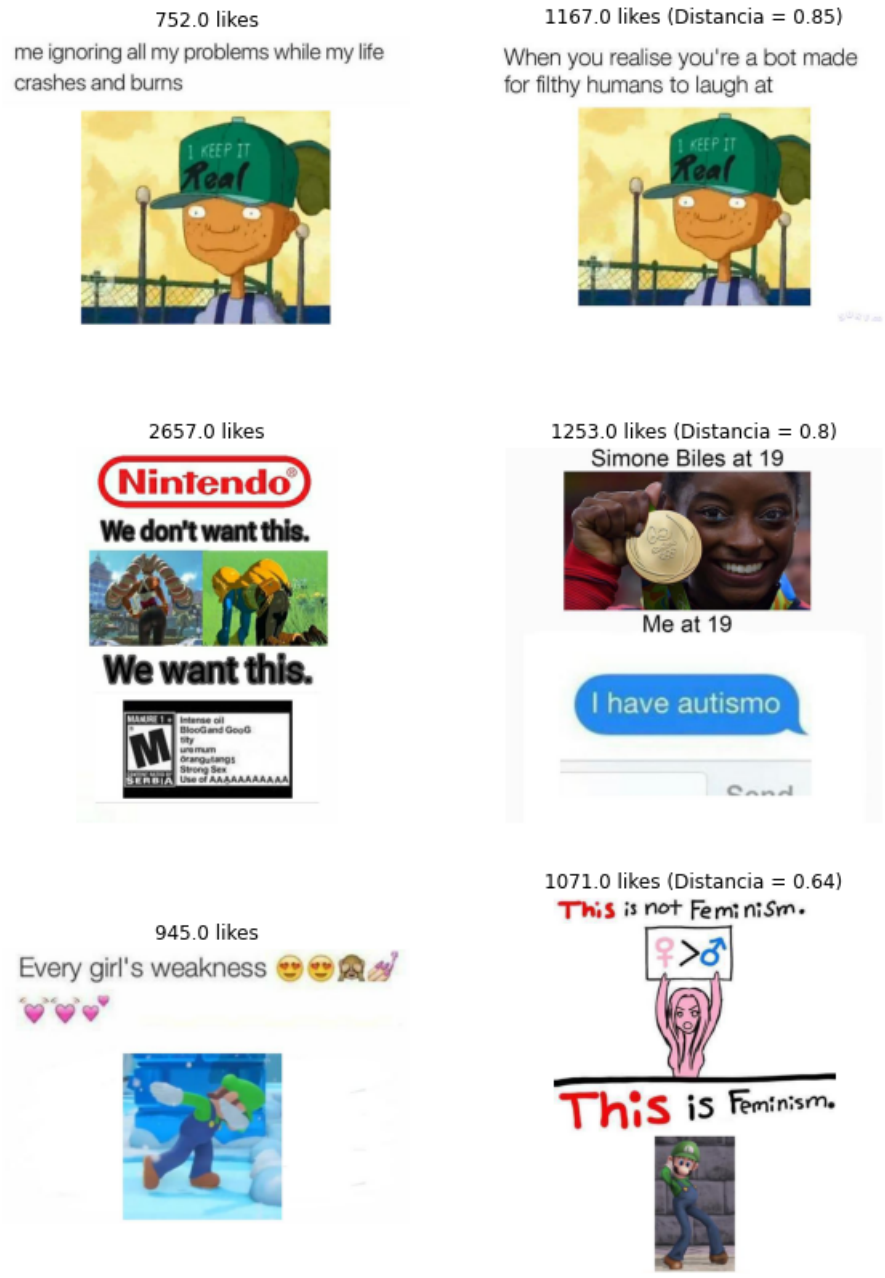

Figura 4.27: Memes y su pareja más cercana (*AlexNet* y similitud de coseno).

Figura 4.28: Memes y su pareja más cercana (*ResNet* y similitud de coseno).

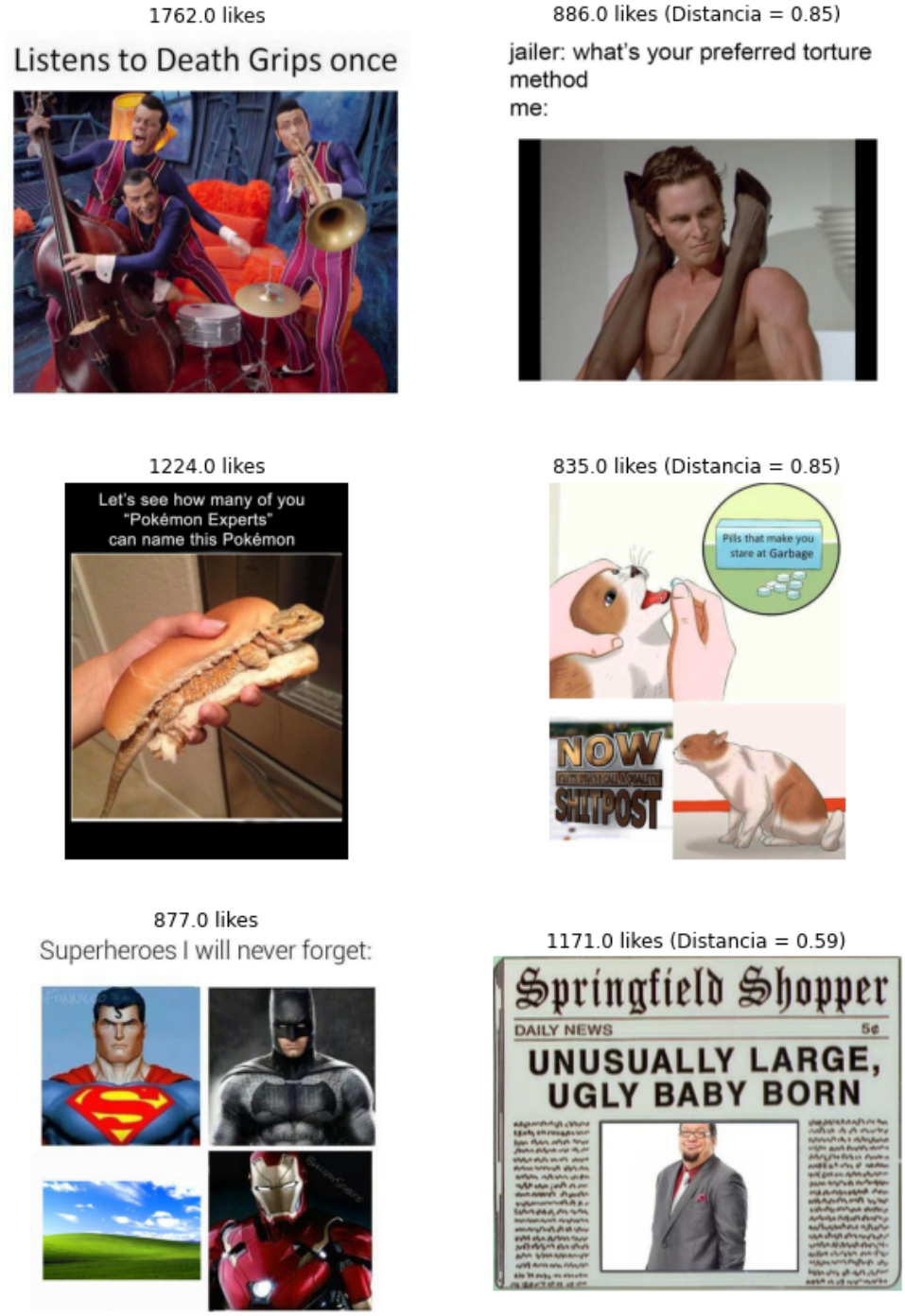

Figura 4.29: Memes y su pareja más cercana (*VGG16* y similitud de coseno).

4.3. Red neuronal siamesa con doble entrada (*Mean Squared Error Loss*)

Esta arquitectura se enfoca en tomar la salida de las capas de extracción de características de la red neuronal convolucional para luego ser procesada en una capa totalmente conectada. En este caso, la salida de la red neuronal siamesa se concentra en una sola célula, que representa la diferencia de likes entre los memes de entrada.

Durante el entrenamiento, se cargan dos memes que pasan por la misma red neuronal convolucional, una vez obtenidos sus vectores de características correspondientes, estos son procesados por una capa densa, obteniendo al final, una cifra que representa la diferencia de likes entre ambos (Figura 4.30). La diferencia predicha por la red neuronal pasa por una función de pérdida, en este caso, *Mean Squared Error Loss*.

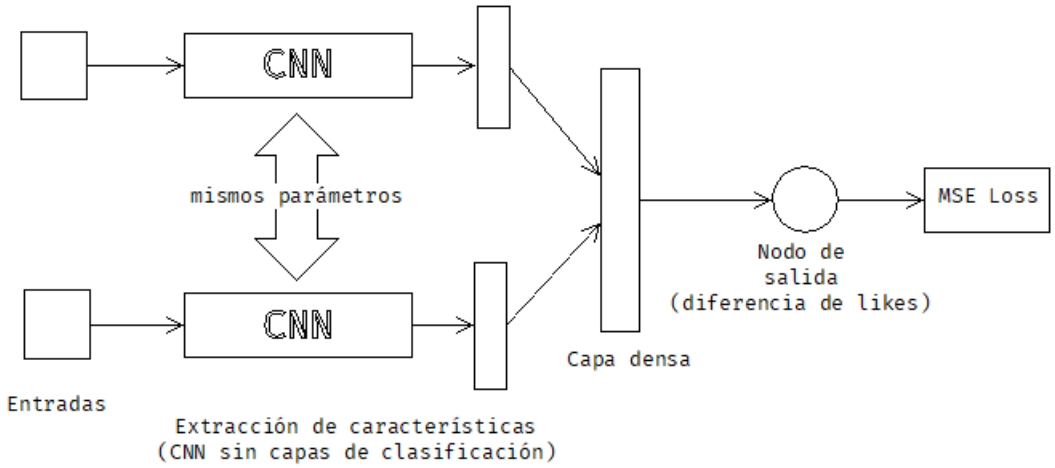

Figura 4.30: Arquitectura de la red neuronal siamesa con entrada doble.

La meta del entrenamiento es la construcción de un modelo capaz de diferenciar memes, es decir, debe poder calcular la diferencia de likes entre dos memes (Figura 4.31). Si procesa dos memes con un éxito similar, lanzará como salida una cifra cercana a 0, sea positiva o negativa.

Si se desea predecir los likes de un meme en particular, solo debe ser cargado como entrada de la red neuronal e iterativamente, cargar como su pareja correspondiente a los elementos del conjunto de entrenamiento. Debemos tomar aquellos que mostraron las diferencias más pequeñas y a partir de sus likes correspondientes, obtener un promedio.

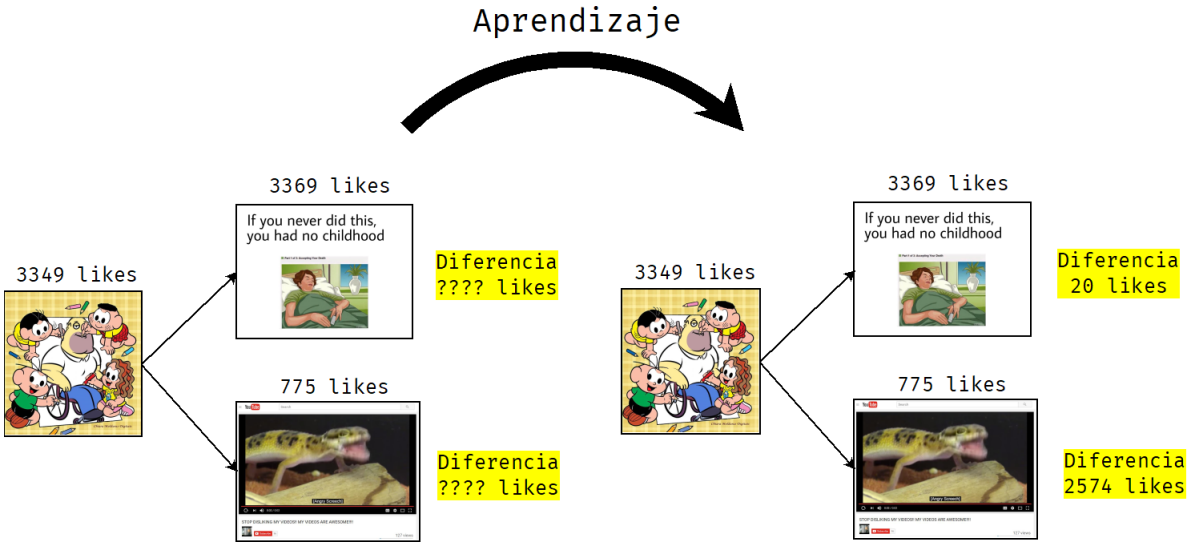

Figura 4.31: Proceso de aprendizaje con *Mean Squared Error Loss*.

4.3.1. Entrenamiento

Antes de examinar las gráficas del entrenamiento, es importante recalcar que la función de pérdida del error cuadrático medio arroja datos muy grandes por la simple definición de la fórmula (Ecuación 3.4).

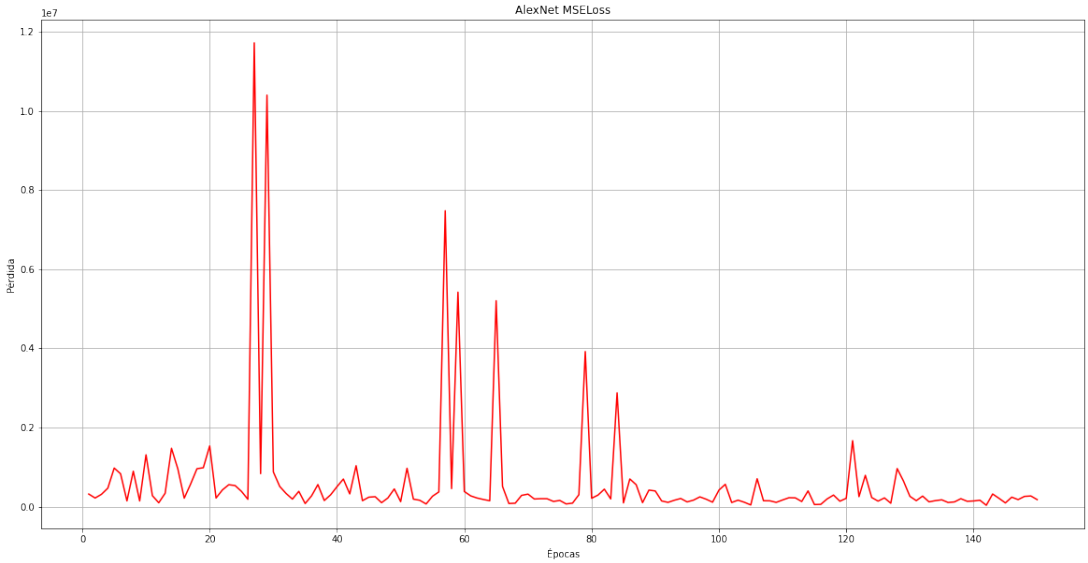

Figura 4.32: Entrenamiento del modelo con *AlexNet*.

Al trabajar con diferencias de likes que podían llegar hasta los miles, se hallaron valores de pérdida alarmantes, por lo que, se volvió prácticamente imposible llegar a una pérdida cercana a 0.

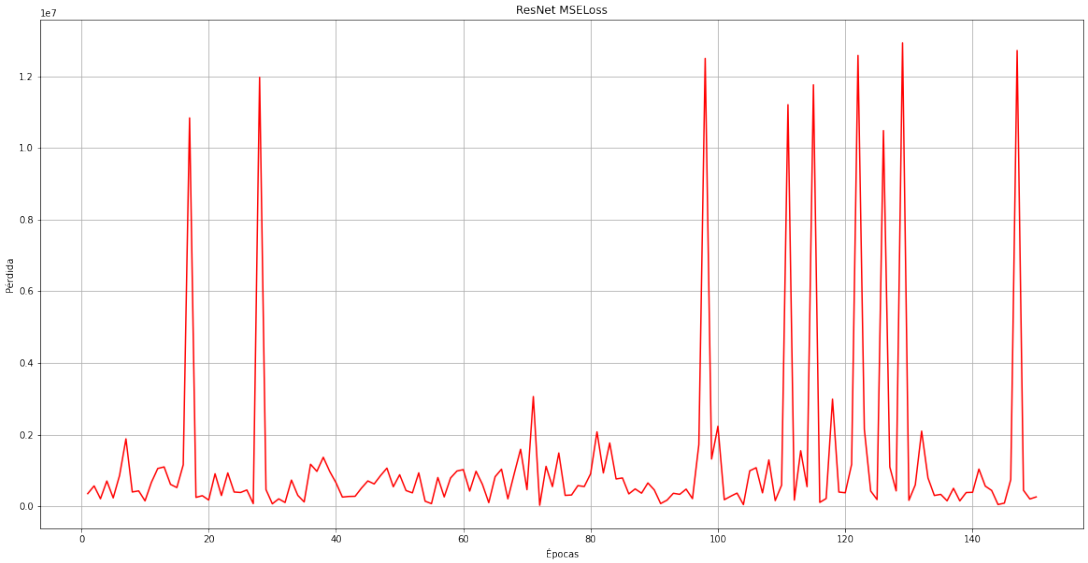

Figura 4.33: Entrenamiento del modelo con *ResNet*.

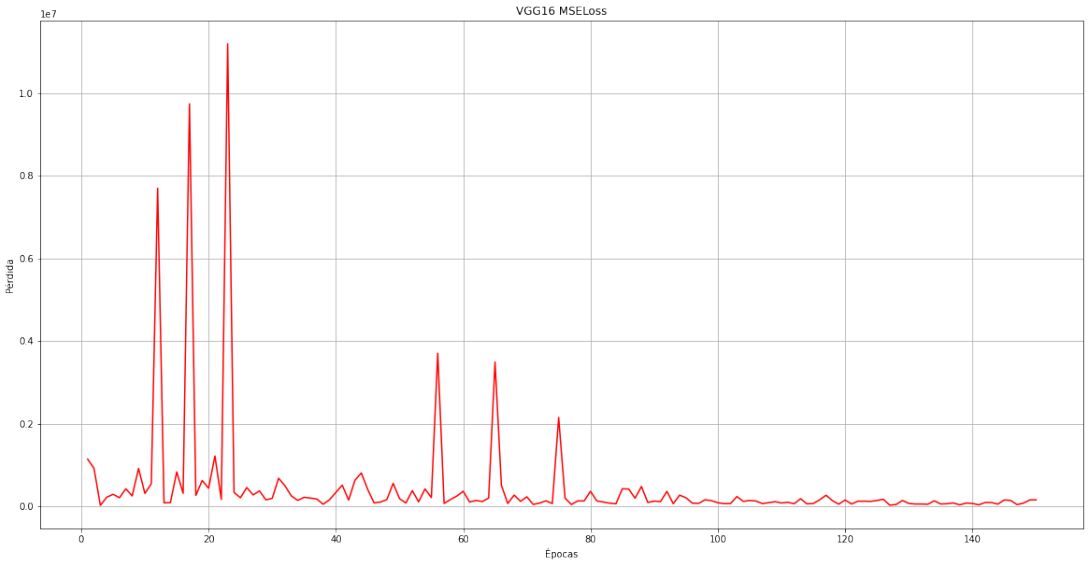

Figura 4.34: Entrenamiento del modelo con *VGG16*.

Se observó *VGG16* (Figura 4.34) mostró un mejor comportamiento, ya que

conforme avanzaron las épocas de entrenamiento convergieron a un valor. Mientras tanto, *ResNet* (Figura 4.33) y *AlexNet* (Figura 4.32) mostraron fluctuaciones graves, pues una vez que parecieron converger, la pérdida se elevó nuevamente.

Sin embargo, y de manera sorpresiva, las tres arquitecturas mostraron un comportamiento similar en cuanto a los picos máximos y mínimos de la pérdida con valores prácticamente idénticos.

4.3.2. Verificación

VGG16 mostró los valores de error cuadrático medio más bajos en el segundo y tercer rango de likes (Tabla 4.7), a diferencia de *ResNet* y *AlexNet*, cuyos valores se dispararon. Es de destacar que ninguna de las tres arquitecturas mostró un rendimiento claramente superior con respecto a las demás.

Error cuadrático medio (ECM)				
CNN	750 a 1,000 likes	1,001 a 1,500 likes	1,501 a 2,000 likes	2,001 +
AlexNet	903,953.73	332,484.58	189,430.77	2,715,713.14
ResNet	393,553.78	267,777.33	252,984.77	3,718,948.71
VGG16	630,840.47	116,104.62	156,367.0	2,769,393.86

Tabla 4.7: Error cuadrático medio de la red neuronal siamesa con doble entrada.

Raíz del error cuadrático medio (RMSE)				
CNN	750 a 1,000 likes	1,001 a 1,500 likes	1,501 a 2,000 likes	2,001 +
AlexNet	950.76	576.61	435.24	1,647.94
ResNet	627.34	517.47	502.98	1,928.46
VGG16	794.25	340.74	395.43	1,664.15

Tabla 4.8: Raíz del error cuadrático medio de la red neuronal siamesa con doble entrada.

En cuanto a la precisión (Tabla 4.9), *ResNet* mostró muy buenos resultados en el segundo rango de likes, pues llegó hasta un 42 % de éxito, detrás quedó *VGG16*, que superó a *ResNet* en el tercer rango. *AlexNet* mostró un desempeño mediocre, pues no logró destacar en ningún rango.

Precisión				
CNN	750 a 1,000 likes	1,001 a 1,500 likes	1,501 a 2,000 likes	2,001 +
AlexNet	0.1	0.08	0.32	0.00
ResNet	0.06	0.42	0.23	0.00
VGG16	0.06	0.29	0.32	0.00

Tabla 4.9: Precisión de la red neuronal siamesa con doble entrada.

Curiosamente, la precisión para los memes con menos likes fue bastante baja, en ninguna arquitectura muestra cifras mínimamente aceptables.

4.3.3. Resultados visuales (memes del conjunto de verificación y su mejor pareja del conjunto de entrenamiento)

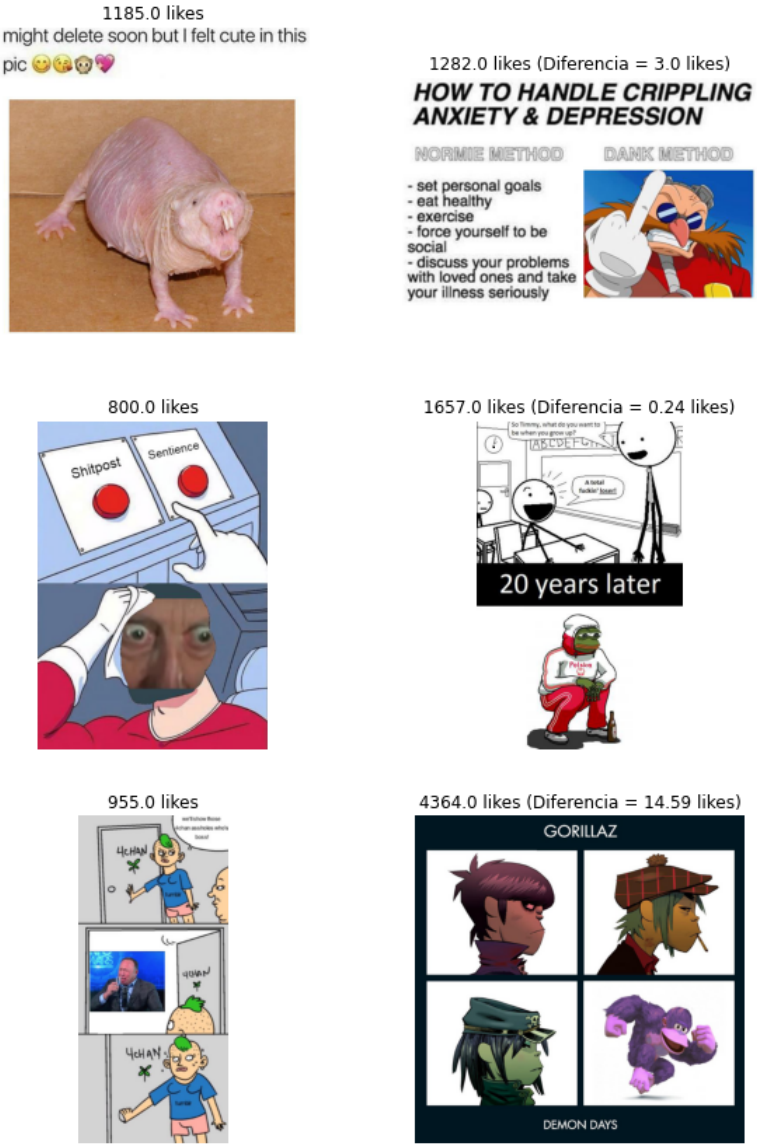

Figura 4.35: Memes y su pareja más cercana (*AlexNet*).

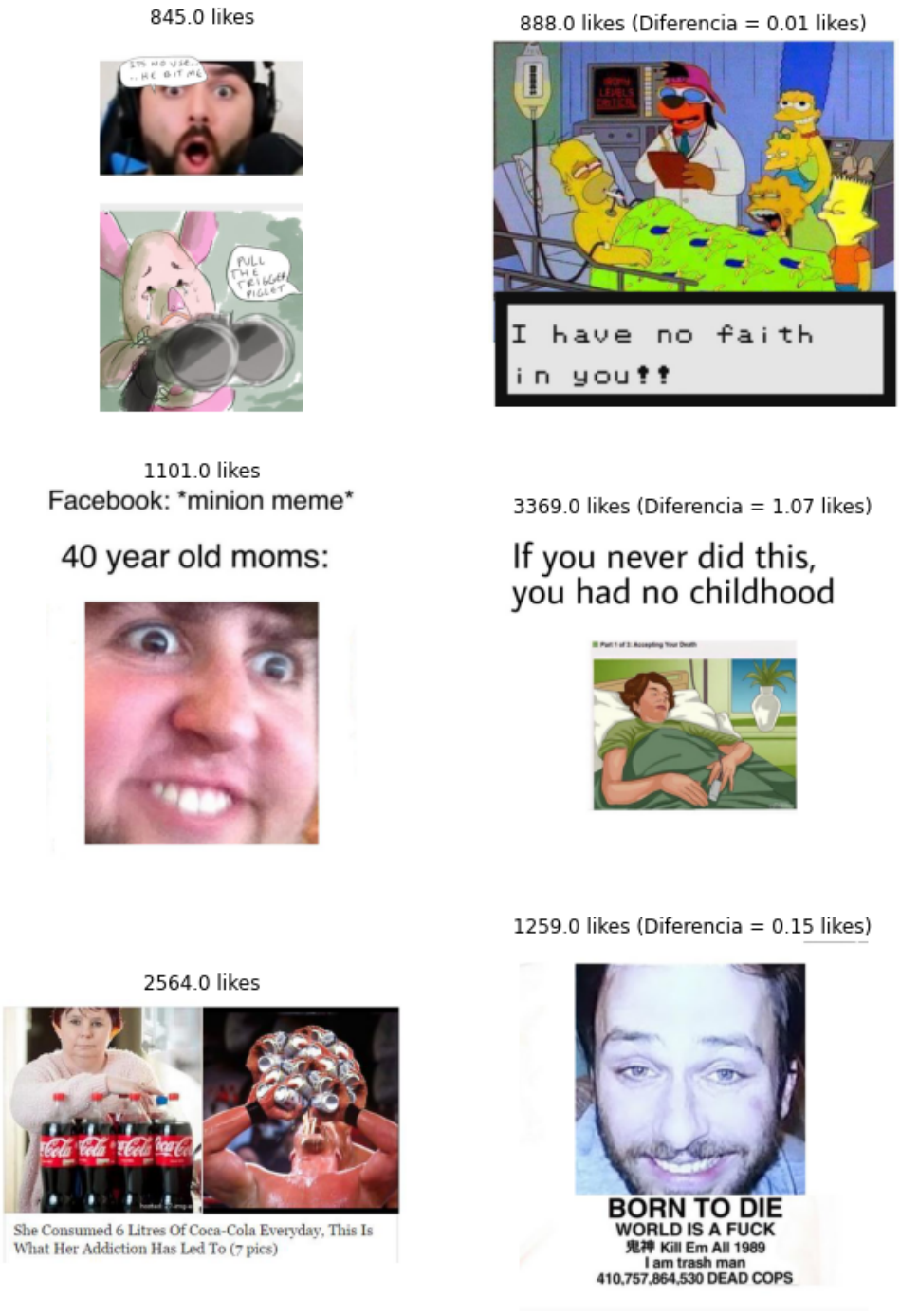

Figura 4.36: Memes y su pareja más cercana (*ResNet*).

Figura 4.37: Memes y su pareja más cercana (*VGG16*).

Capítulo 5

Conclusiones

Lamentablemente, la primera conclusión y la más evidente, es que la poca de cantidad de datos y desigual distribución de estos influyeron de forma negativa en el desempeño de nuestros modelos. La obtención de memes fue un proceso largo, cuyos resultados son completamente aleatorios, ya que recae en un factor externo imposible de predecir, como lo es la interpretación humana, y es que, son los mismos seres humanos los que determinamos que tan gracioso o exitoso es un meme.

Dicho esto, aún con la falta de datos, se obtuvieron resultados esperanzadores de hasta un 30 % de precisión. Aunque estas cifras puedan parecer mediocres, sobre todo para el campo de la visión computacional, no hay que olvidar que se exploró una problemática prácticamente nueva y de que, por lo que se ha planteado en el trabajo, no constituye una tarea *común* de clasificación de imágenes por clases predefinidas, sino más bien, una clasificación basándose en un contexto totalmente externo al contenido de la imagen en sí.

Algo que se puede rescatar es que el modelo que utilizó *Mean Squared Error Loss* logró clasificar los memes desprendiéndose del aspecto *visual*, es decir, los memes que eran parecidos visualmente o que compartían ciertos elementos dejaron de ser emparejados. Esto significa un gran logro, ya que las redes neuronales convolucionales utilizadas en este trabajo (*AlexNet*, *VGG16*, *ResNet18*) están diseñadas especialmente para relacionar a las imágenes por su contenido y para la realización de tareas de clasificación.

Como se explicó anteriormente, lo que otorga el éxito a un meme en mayor parte es su contexto y no el contenido como tal, por lo que no es descabellado pensar que si se continúa por este camino, la creación de un modelo eficiente no parece tan lejana.

Afortunadamente, tanto el aprendizaje profundo como la visión computacional se encuentran en constante innovación, por lo que se podría llegar a la construcción de una arquitectura *perfecta* para el análisis de humor en imágenes, como en

su tiempo lo fueron las redes neuronales convolucionales para el procesamiento de imágenes.

5.1. Trabajo futuro

Lo ideal sería complementar el conjunto de datos buscando llegar a los miles de memes, evidentemente, este proceso llevaría tiempo, ya que existe una dependiencia en las interacciones de *ShitpostBot5000*. Además, es necesario hallar una forma de etiquetar los memes, lo que significaría incluir trabajo externo, es decir, un equipo de trabajo capaz de incluir etiquetas a las imágenes indicando el contexto de esta, o bien, tal vez, incluir herramientas capaces de extraer el texto de las imágenes.

Todo este proceso sería incluido dentro de una API, donde los usuarios puedan acceder y etiquetar los memes según su percepción, es así, como además de recopilar imágenes, recopilamos la precepción que estas generan en distintos usuarios. El objetivo es complementar el uso de redes neuronales con la utilización de bases de datos y desarrollo web, buscando crear un conjunto de datos cada vez más diverso y sólido, lo que siempre deriva en mejores resultados.

Al momento de incluir etiquetas en nuestro conjunto, la idea es poder agregarlos a la representación de las imágenes dentro de la red neuronal siamesa, así podemos implicarlas dentro del procesamiento de estas, incluyendo así, tareas de procesamiento de lenguaje natural. Es así, como podemos agregar de forma más clara el contexto de los memes, además de su éxito correspondiente.

Bibliografía

[1] **The Bot Appreciation Society Wiki**. Disponible en: https://botappreciationsociety.fandom.com/wiki/The_Bot_Appreciation_Society_Wiki. 7

[2] **How neural networks are trained**. Disponible en: https://ml4a.github.io/ml4a/how_neural_networks_are_trained/. 12, 14, 15

[3] Mayank Agarwal. **Back Propagation in Convolutional Neural Networks — Intuition and Code**, Octubre 2020. Disponible en: https://tinyurl.com/3ab6c8f8. III, III, 11, 19

[4] Julien Beaulieu. **Error functions**, Junio 2019. Disponible en: https://julienbeaulieu.gitbook.io/wiki/sciences/machine-learning/linear-regression/error-functions-1. IV, IV, 35

[5] Moitreya Chatterjee and Yunan Luo. **Similarity Learning with (or without) Convolutional Neural Network**. page 75, 2017. Disponible en: https://slazebni.cs.illinois.edu/spring17/lec09_similarity.pdf. IV, 26

[6] Gideon Resnick Collins, Ben. **Palmer Luckey: The Facebook Near-Billionaire Secretly Funding Trump's Meme Machine**. *The Daily Beast*, Septiembre 2016. Disponible en: https://tinyurl.com/44edssxp. 5

[7] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. **ImageNet: A large-scale hierarchical image database**. pages 248–255, 2009. 20

[8] Adit Deshpande. **A Beginner's Guide To Understanding Convolutional Neural Networks**, Julio 2016. Dis-

ponible en: https://adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks/. 21

[9] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. **Deep Residual Learning for Image Recognition**. *arXiv:1512.03385 [cs]*, Diciembre 2015. arXiv: 1512.03385. Disponible en: http://arxiv.org/abs/1512.03385. 30, 31

[10] IBM. **What are Convolutional Neural Networks?**, Octubre 2020. Disponible en: https://www.ibm.com/cloud/learn/convolutional-neural-networks. iv, 15, 17, 26

[11] IBM. **What is Gradient Descent?**, Octubre 2020. Disponible en: https://www.ibm.com/cloud/learn/gradient-descent. iii, 16

[12] Jefkine. **Backpropagation In Convolutional Neural Networks**, Septiembre 2016. Disponible en: https://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/. 19

[13] Gregory Koch, Richard Zemel, and Ruslan Salakhutdinov. **Siamese Neural Networks for One-shot Image Recognition**. page 8, 2015. Disponible en: https://www.cs.cmu.edu/~rsalakhu/papers/oneshot1.pdf. 26

[14] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. **ImageNet Classification with Deep Convolutional Neural Networks**. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, **25. Curran Associates, Inc., 2012. Disponible en:** https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf**. 29**

[15] Hussain Mujtaba**. Similarity learning with Siamese Networks | What is Siamese Networks, Diciembre 2020. Section: Artificial Intelligence. Disponible en:** https://www.mygreatlearning.com/blog/siamese-networks/**. 24**

[16] M P Mulder and A Nijholt. **Humour Research: State of the Art. page 24.** 5

[17] Sunita Nayak. **Understanding AlexNet | LearnOpenCV #, Junio 2018. Disponible en:** https://learnopencv.com/understanding-alexnet/. iv, 29, 30

[18] Keiron O'Shea and Ryan Nash. **An Introduction to Convolutional Neural Networks.** ***arXiv:1511.08458 [cs]*****, Diciembre 2015. arXiv: 1511.08458. Disponible en:** http://arxiv.org/abs/1511.08458. iii, 21, 23

[19] Abhay Parashar. **Vgg 16 Architecture, Implementation and Practical Use, Octubre 2020. Disponible en:** https://medium.com/pythoneers/vgg-16-architecture-implementation-and-practical-use-e0fef1d14557. iv, 31

[20] Abel L. Peirson V and E. Meltem Tolunay. **Dank Learning: Generating Memes Using Deep Neural Networks.** ***arXiv:1806.04510 [cs]*****, Junio 2018. arXiv: 1806.04510. Disponible en:** http://arxiv.org/abs/1806.04510. 7

[21] Farheen Ramzan, Muhammad Usman Khan, Asim Rehmat, Sajid Iqbal, Tanzila Saba, Amjad Rehman, and Zahid Mehmood. **A Deep Learning Approach for Automated Diagnosis and Multi-Class Classification of Alzheimer's Disease Stages Using Resting-State fMRI and Residual Neural Networks.** ***Journal of Medical Systems*****, 44, 12 2019.** iv, 30

[22] Max Reynolds. **Backpropagation: Intuition and Explanation, Febrero 2021. Disponible en:** https://towardsdatascience.com/backpropagation-intuition-and-derivation-97851c87eece. **18**

[23] Graeme Ritchie, Ruli Manurung, Helen Pain, Annalu Waller, Rolf Black, and Dave O'Mara. **A practical application of computational humour. page 8. 3, 7**

[24] Roberto Castro Sundin, Tony Rönnqvist, Alejandro Sarmiento González. **2 Methods | Siamesifying the COVID-Net, Mayo 2020. Disponible en:** https://people.kth.se/~rosun/deep-learning/sec-methods.html#sec:arch. **iv, 25**

[25] Sebastian Ruder. **An overview of gradient descent optimization algorithms, Enero 2016. Disponible en:** https://ruder.io/optimizing-gradient-descent/. **16**

[26] Mikaela Sanchez. **What does it mean to train a Neural Network?, Agosto 2019. Disponible en:** https://medium.com/@mikaelaysanchez/what-does-it-mean-to-train-a-neural-network-64065fbc7bb0. **iii, 13**

[27] Natasha Sharma. **Importance of Distance Metrics in Machine Learning Modelling, Enero 2019. Disponible en:** https://tinyurl.com/2nn5tf9z. **iv, iv, 33, 34**

[28] Daniel Shiffman. **Chapter 10. Neural Networks, 2012. Disponible en:** https://natureofcode.com/book/chapter-10-neural-networks/. **iii, 9**

[29] Karen Simonyan and Andrew Zisserman. **Very Deep Convolutional Networks for Large-Scale Image Recognition.** ***arXiv:1409.1556***

[cs], **Abril 2015. arXiv: 1409.1556. Disponible en:** http://arxiv.org/abs/1409.1556. **31**

[30] Prabhat Singh. **Introduction to Siamese Networks, Septiembre 2020. Disponible en:** https://medium.com/analytics-vidhya/a-friendly-introduction-to-siamese-networks-283f31bf38cd. **iv, 32**

[31] Stanford. **CS231n Convolutional Neural Networks for Visual Recognition, Junio 2017. Disponible en:** https://cs231n.github.io/convolutional-networks/. **iii, iv, iv, 20, 22, 23**

[32] Ilan Stavans and Javier Adrada de la Torre. **Sobre la gramática del shitposting. Disponible en:** https://cultura.nexos.com.mx/sobre-la-gramatica-del-shitposting/. **3**

[33] Francis Tseng. **Neural networks, Abril 2016. Disponible en:** https://ml4a.github.io/ml4a/neural_networks/. **iii, 11, 14**

[34] Wanshun Wong. **What is a Siamese Neural Network?, Octubre 2020. Disponible en:** https://towardsdatascience.com/what-is-a-siamese-neural-network-b0dbeb1c6db7. **24**

[35] Matthew D. Zeiler and Rob Fergus. **Visualizing and Understanding Convolutional Networks.** ***arXiv:1311.2901*****, Noviembre 2013. Disponible en:** http://arxiv.org/abs/1311.2901. **23**