Departamento de Computación
Facultad de Informática
UNIVERSIDADE DA CORUÑA

TRABAJO FIN DE GRADO
GRADO EN INGENIERÍA INFORMÁTICA
MENCIÓN EN COMPUTACIÓN

Técnicas de deep learning para la caracterización de imágenes en sistemas de recomendación

Estudiante: Eva Blanco Mallo
Director/a/es/as: Verónica Bolón Canedo
Beatriz Remeseiro López

A Coruña, 6 de setembro de 2019.

A mi abuela.

Agradecementos

A mi familia, por estar siempre a mi lado. En especial a mi madre, por ayudarme a estudiar esta carrera aunque significase que mi vuelo se iba a retrasar un poco más de lo esperado. A mis tutoras, por darme la oportunidad de adentrarme en un campo tan apasionante y confiar siempre en mí. A mis compañeros de laboratorio y facultad, por acompañarme durante todos estos años. Vuestro apoyo e inestimable ayuda, en ocasiones a distancia, han sido fundamentales a lo largo de este camino, haciéndome sentir como en casa. Y finalmente, a Diego, porque sin él nunca me hubiese planteado comenzar a recorrerlo.

Resumo

En este proyecto se estudian diferentes técnicas de *deep learning* aplicadas a la extracción de características de imágenes que posteriormente serán utilizadas en un sistema de recomendación. Para ello, desarrollaremos un sistema de clasificación binaria que hará uso de las imágenes extraídas de las valoraciones a restaurantes publicadas por los usuarios de TripAdvisor. Se estudiarán y analizarán diferentes aproximaciones para obtener el vector de características que mejor defina una imagen dada, utilizando diferentes modelos de la bibliografía y variantes de los mismos, concretamente redes convolucionales y autoencoders. Además, se aplicarán distintas técnicas de aprendizaje máquina para tratar los problemas de desbalanceo presentes en el conjunto de datos disponible y para evaluar los resultados obtenidos en la predicción de los gustos de los usuarios.

Palabras chave:

- Extracción de características en imágenes
- Recomendación personalizada de restaurantes
- TripAdvisor
- Sistemas recomendadores inteligentes
- Autoencoders convolucionales
- CNN
- Data augmentation
- Transfer learning

Índice Xeral

Índice de Figuras

Índice de Táboas

Capítulo 1

Introducción

En este primer capítulo abordaremos la importancia que las redes sociales y recomendaciones online han ido adquiriendo a lo largo de los últimos años, motivo por el cual, un sistema capaz de detectar características personalizadas y ofrecer opciones en base a ellas, tendría un gran impacto en términos de consumo.

1.1 Motivación y objetivos

Las tecnologías y redes sociales han cambiado nuestras costumbres y modos de vida, millones de personas comparten a diario sus gustos y opiniones en la red y acuden a la misma buscando información y recomendaciones de otros usuarios. La llegada de la revolución digital y el comercio electrónico aumentaron de forma espectacular la cantidad de opciones que un consumidor tenía a su alcance, pudiendo llegar a resultar muy tedioso encontrar aquello que realmente le gusta.

También hizo evidente la necesidad de disponer de información sobre productos o servicios que no se pueden ver directamente, lo que aumenta inevitablemente la desconfianza del consumidor. Por ello, la información suministrada por otros usuarios, en forma de imágenes y opiniones, se ha vuelto indispensable en la red. De hecho, aunque el miedo frene este tipo de compras, es común seguir utilizando esta información para comparar precios y productos, interviniendo directamente en las decisiones de compra futura en el propio establecimiento.

Por último, es importante resaltar que la cultura gastronómica en las redes sociales ha estado en constante aumento en los últimos años. Compartir datos sobre restaurantes se está convirtiendo en algo habitual entre los consumidores de plataformas como Instagram o Facebook, dónde las imágenes adquieren un papel fundamental. Esta tendencia de fotografiar comida y compartir opiniones sobre locales particulares, hace que los usuarios acudan a la red

buscando información sobre los mismos en plataformas como Tripadvisor, elTenedor, FourSquare o JustEat. Aunque este tipo de aplicaciones albergan gran cantidad de datos, no hacen recomedaciones personalizadas. Cabe destacar otras plataformas del sector digital con gran éxito que sí lo hacen, como Youtube o Netflix, que recomiendan vídeos y películas o series respectivamente.

En estos factores radica la importancia de un sistema que pueda analizar y hacer uso eficiente de ese gran flujo de datos. Sin embargo, la recomendación de restaurantes es una tarea subjetiva y la personalización adquiere especial protagonismo. Concretamente, en el caso de las imágenes, es complicado identificar a simple vista cuáles son las características que determinan el gusto de los usuarios. Por ello, el objetivo de este proyecto es estudiar cómo caracterizar las imágenes sobre restaurantes que los usuarios publican en TripAdvisor junto a sus valoraciones, para usarlas posteriormente en un sistema de recomendación. La idea general es resolver el problema utilizando diferentes técnicas de *deep learning*, que permiten aprender la representación de los datos al mismo tiempo que se usan para predicción.

1.2 Viabilidad e impacto

Ya en el año 2016 se hacía patente el gran poder de las redes sociales en el comportamiento de los consumidores. En el Informe Total Retail Survey[1] realizado por PwC ese mismo año, podríamos comprobar como el 78% de los consumidores admitían que las redes sociales influían en mayor o menor medida en sus decisiones de compra online. El factor predominante era leer evaluaciones o comentarios de otros usuarios, como podemos ver en la Figura 1.1 (a).

Dos años después, en el Informe Consumer Insights Survey 2018[2], publicado por la misma empresa, resulta interesante observar que las fuentes online que más inspiraban a los usuarios en su decisión de compra eran las redes sociales, las webs multimarca y las webs de reseñas. Sin embargo, las que menos influencia ejercían eran las webs de ofertas, el email marketing y la prensa digital, entre otras; tal y como podemos ver en la Figura 1.1 (b). Por tanto, podemos extraer que el usuario de hoy en día rechaza acercamientos intrusivos e impersonales y tiende a buscar una recomendación más personalizada.

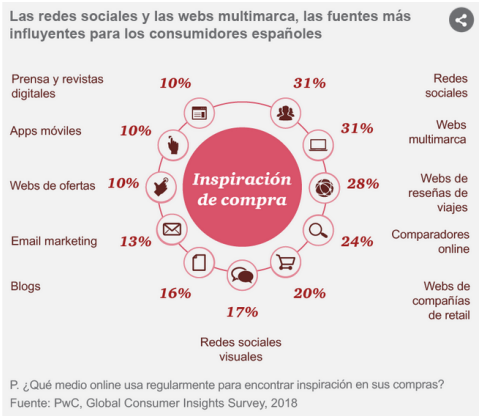
Ya en la décima edición del informe Consumer Insights Survey 2019[3], elaborado también por PwC a partir de la opinión de 21.480 consumidores en 27 países, podemos ver en la Fi-

[1] https://www.pwc.ru/en/publications/totalretail-2016.html

[2] https://www.pwc.es/es/retail-consumo/2018-global-consumer-insights-survey/confianza-del-consumidor.html

[3] https://www.pwc.com/gx/en/consumer-markets/consumer-insights-survey/2019/report.pdf

(a) Influencia en el comportamiento de compra



(b) Inspiración en medios online

(c) Influencia de las redes sociales

Figura 1.1: (a) Aspectos de las redes sociales que influyen en el comportamiento de compra de los consumidores en 2016. (b) Medios online que los consumidores utilizaban como fuente de inspiración en sus compras en 2018. (c) Comportamiento online de los consumidores en relación a las redes sociales en 2019.

gura 1.1 (c) como estas tendencias se mantienen. Además, el factor que más influye en las decisiones de compra de los consumidores, en relación a las redes sociales, siguen siendo las recomendaciones ofrecidas por otros usuarios.

Por tanto, si analizamos las tendencias de compra de los últimos años, podemos ver como las redes sociales y la interacción entre consumidores tienen cada vez más peso en nuestra sociedad. Recordemos que las plataformas de recomendación de restaurantes más populares, mencionadas anteriormente, no hacen recomendaciones personalizadas. El servicio que ofrecen en relación a este aspecto está limitado a promediar calificaciones, sin tener en cuenta el gusto en particular del usuario que formula la búsqueda.

Teniendo en cuenta que Tripadvisor dispone de 760 millones de comentarios y opiniones, 490 millones de visitantes únicos al mes de promedio y 110 millones de fotografías, de acuerdo a su centro de prensa[4], nos podemos hacer una idea del impacto en el consumo que podrían llegar a tener las recomendaciones. Es por ello que para nuestro proyecto hemos utilizado imágenes de críticas de usuarios de Tripadvisor, ya que proporciona un conjunto de datos lo suficientemente rico como para ilustrar el significado de un sistema recomendador, además de ser fácilmente extrapolable a otras plataformas similares.

1.3 Planificación

El tiempo necesario dedicado, de forma parcial, para llevar a cabo la realización de todas las tareas del proyecto fue de 212,5 días, dando comienzo el 3 de diciembre de 2018 y finalizando el 25 de junio de 2019, suponiendo un total de 850 horas de trabajo.

En primer lugar se llevó a cabo una fase de recopilación de información y estudio teórico, donde se estudiaron las principales técnicas de *deep learning* utilizadas en el campo de la extracción de características, así como las investigaciones más recientes relacionadas con nuestro problema. En segundo lugar se procedió al análisis y preprocesamiento del conjunto de datos inicial para obtener los diferentes conjuntos de prueba. En este punto fue necesario utilizar los conocimientos aprendidos en la tarea anterior, ya tuvimos que seguir diferentes estrategias debido a las características de nuestros datos de entrada.

Para las siguientes fases nos decantamos por seguir una metodología de construcción de prototipos en un ciclo de vida en espiral, el motivo es que en cada aproximación se modifica el sistema, ya sea para obtener las características por diferentes vías o para mejorar su rendimiento. Este modelo resulta útil cuando se conocen los objetivos generales pero no iden-

[4]https://tripadvisor.mediaroom.com/us

tificamos en un primer momento los requisitos detallados de procesamiento. En nuestro caso, no hemos encontrado otros estudios en los que se apliquen metodologías estándar en la resolución de este tipo de problemas. Por el mismo motivo, al no estar seguros de la eficacia de los diferentes algoritmos y técnicas de prueba, resulta beneficioso seguir esta metodología, ya que permite ir modificando los modelos e incorporando nuevas estrategias en función de los resultados observados. Para cada una de las aproximaciones abordadas se realizaron las siguientes tareas: diseño e implementación del sistema en función de la estrategia de extracción de características seleccionada, pruebas y validación de los diferentes modelos con las técnicas de evaluación elegidas para medir su rendimiento y finalmente, análisis de resultados, comparando el efecto de las técnicas aplicadas. Por último, fue necesario redactar la documentación del proyecto. La planificación temporal de cada una de las tareas se refleja en el Diagrama de Gantt de la Figura 1.2.

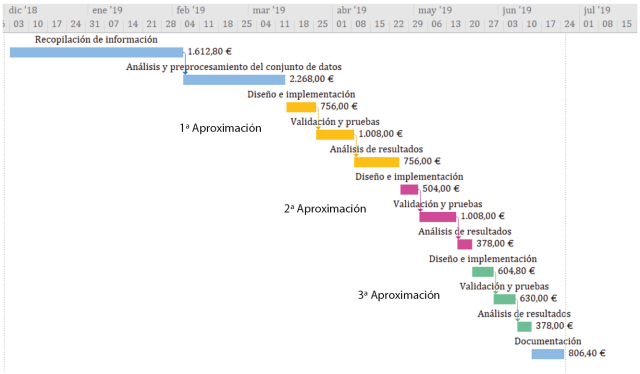


Figura 1.2: Diagrama de Gantt con la planificación de las tareas y costes del proyecto.

Los recursos humanos necesarios para realizar el proyecto consistieron en un analista programador y dos directores. Los recursos técnicos utilizados fueron un ordenador portátil personal y un ordenador de sobremesa, que por adquirirse con anterioridad no supusieron ningún gasto. Las características de ambas máquinas se enumeran en la Tabla 1.1.

Característica	Ordenador Potátil	Ordenador Sobremesa
Procesador	Intel® Core™ i7-6700HQ CPU @ 2.60GHz × 8	Intel® Core™ i7-4790 CPU @ 3.60GHz × 8
Gráficos	Intel® HD Graphics 530 (Skylake GT2)	TITAN Xp
Tipo de SO	64 bits	64 bits
Memoria	15,5 GiB	15,5 GiB

Táboa 1.1: Especificaciones de los ordenadores utilizados.

El sueldo bruto anual de un analista programador según el convenio está fijado a 24.157,27€ a tiempo completo, por lo que el coste por hora es de 12,60€. En la Tabla 1.2 podemos ver el

coste de cada una de las tareas planificadas en función de las horas estimadas. Teniendo en cuenta que no fue necesario acometer ningún gasto debido a hardware o software adicional, para el coste total del proyecto solo es necesario computar el coste del analista programador. Por tanto, como se refleja en la Tabla 1.3, el coste total del proyecto asciende a 10.710,00 €.

Tarea	Duración	Coste
Recopilación de información	128 h.	1.612,80€
Análisis y preprocesamiento del conjunto de datos	180 h.	2.268,00€
Primera Aproximación	200 h.	2.520,00€
Diseño e implementación	60 h.	756,00€
Validación y pruebas	80 h.	1.1008,00€
Análisis de resultados	60 h.	
756,00€		
Segunda Aproximación	150 h.	1.890,00€
Diseño e implementación	40 h.	504,00€
Validación y pruebas	80 h.	1.008,00€
Análisis de resultados	30 h.	378,00€
Tercera Aproximación	128 h.	1.612,80€
Diseño e implementación	48 h.	604,80€
Validación y pruebas	50 h.	630,00€
Análisis de resultados	30 h.	378,00€
Documentación	64 h.	806,40€
Coste total		
10.710,00€		

Táboa 1.2: Duración estimada de las tareas del proyecto y costes.

Recurso	Tiempo dedicado	Coste	Coste Total
Analista programador	850 h.	12,60€/h.	10.710,00€
Directores	48 h.	0€/h.	0€
Total			13.860,00€

Táboa 1.3: Costes y esfuerzos del proyecto.

1.4 Herramientas software

La ventaja de la programación en Python, con liberías como Keras, Tensorflow y Scikit-Learn es la variedad de módulos y algoritmos que proporcionan, facilitando el aprendizaje a través de un rápido desarrollo en en las primeras fases.

Tensorflow

Tensorflow[5] es una plataforma de código abierto utilizada en aplicaciones de aprendizaje máquina (o aprendizaje automático) desarrollada por Google. Originariamente, fue creada con fines internos por el equipo de Google Brain, para el filtrado de spam en Gmail, pero tras su lanzamiento en 2015 se ha consolidado como el framework de programación líder en el sector. Es importante resaltar su gran flexibilidad, ya que puede ser ejecutado en la nube, en aplicaciones y en todo tipo de dispositivos hardware. Además, permite diferentes niveles de abstracción, pudiendo servir tanto a desarrolladores principiantes como a expertos.

Su principal característica es que utiliza grafos dirigidos y tensores para construir y entrenar redes neuronales. Los tensores son matrices multidimensionales que se sitúan en los bordes de los grafos, con nodos entre ellos representando operaciones matemáticas. Por ejemplo, un tensor con dimensión igual a cero es un escalar, si tiene una sola dimensión sería un vector y con dos dimensiones, una matriz. Los cálculos se ejecutan dentro de las sesiones, pero siguiendo una estrategia "evaluación perezosa". Primero se crea el grafo, que determina los tamaños de los tensores y las operaciones que se van a ejecutar, y después se realizan los cálculos pertinentes dentro de una sesión.

Modelar las redes neuronales de esta forma presenta grandes ventajas. Un grafo muestra las dependencias entre operaciones, por lo que al separar la construcción de su ejecución podemos dividirlo en subgrafos. Por tanto, separa la computación en partes más pequeñas, lo que implica un gran ahorro de recursos y permite una computación distribuida. Dado que las operaciones no dependen entre sí, podemos utilizar la CPU junto con la GPU, o incluso múltiples GPU en paralelo, aumentando significativamente la velocidad de procesamiento.

A pesar de no trabajar directamente con Tensorflow durante la realización de este proyecto, podemos beneficiarnos de todas las ventajas que ofrece. Como veremos en el siguiente apartado, Keras es capaz de ejecutarse sobre Tensorflow, y ésta es la librería que utilizaremos.

Keras

Keras[6] proporciona una API de código abierto de alto nivel para trabajar con redes neuronales, escrita en Python y capaz de ejecutarse sobre TensorFlow, Theano o Microsoft Cognitive Toolkit. Está especialmente diseñado para posibilitar la experimentación rápida y prototipado con redes neuronales, en el área del *deep learning*. Sus fuertes se centran en ser intuitivo para el usuario, modular y extensible. Permite, además, ejecutar en CPU y GPU.

[5] https://www.tensorflow.org/learn
[6] https://keras.io/

Existe una gran red de desarrolladores y usuarios detrás de Keras. Su aportación constante y ayuda en la resolución de problemas, no solamente en forma de código, si no explicación, la convierten en una herramienta totalmente recomendable. Otra de las grandes ventajas de utilizar Keras durante este proyecto, es que pone a disposición de los investigadores la implementación de arquitecturas muy populares, pre-entrenadas con grandes conjuntos de datos y fácilmente adaptables a otros problemas.

Scikit Learn

Al igual que las anteriores, Scikit Learn[7] es una librería de software libre para aprendizaje máquina. Está construida sobre SciPy (Scientific Python) e incluye paquetes como NumPy, Pandas o matplotlib. Proporciona multitud de algoritmos, tanto de aprendizaje supervisado como no supervisado, métodos de validación, optimización, conjuntos de datos y herramientas de procesamiento de imágenes, entre otros. Al igual que los anteriores, tiene una gran comunidad de usuarios detrás, facilitando enormemente su uso.

1.5 Estructura de la memoria

A lo largo de esta memoria, profundizaremos en los siguientes aspectos:

- Introducción: En el primer capítulo se explican los motivos por los que decidimos abordar este proyecto, objetivos, viabilidad del mismo y posible impacto. También se incluye la planificación que hemos seguido y se describen brevemente las herramientas utilizadas más relevantes.
- Descripción del problema: En este capítulo se hace una breve introducción sobre los últimos avances en relación a sistemas de recomendación con imágenes. También se introduce la problemática de la clasificación con imágenes en general, así como los diferentes retos que plantea nuestro problema en concreto.
- Metodología: Descripción de la arquitectura general propuesta y de sus diferentes aproximaciones, seguida de la explicación de todas las técnicas que conforman el modelo.
- Conjunto de datos: Análisis del conjunto inicial, procedimiento de selección de las particiones finales y un breve resumen con las características más destacables de las mismas.
- Experimentación: A lo largo de este capítulo se presentarán los diferentes experimentos que hemos realizado, explicando las estrategias utilizadas y los procedimientos de validación, así como las medidas de rendimiento utilizadas para evaluar el correcto funcionamiento del modelo propuesto.

[7] https://scikit-learn.org/stable/

- Resultados: Capítulo dónde se exponen y analizan los resultados obtenidos para cada una de las aproximaciones.
- Conclusiones: Finalmente, se exponen los aspectos más relevantes que pudimos extraer de la realización de este proyecto, así como diversas líneas de trabajo futuro que consideramos interesantes.

Capítulo 2

Descripción del problema

Alo largo de este capítulo se explicará brevemente en qué consiste un sistema de recomendación, cuyo rendimiento pretendemos mejorar a través de la extracción de características en las imágenes de las valoraciones. Además conoceremos las últimas investigaciones relacionadas con el uso de imágenes en recomendaciones. A continuación se hablará sobre la clasificación de imágenes y los retos que plantea en general, motivo por el cual caracterizarlas de este modo es tan importante en este tipo de problemas. Por último, introduciremos brevemente las dificultades a las que nos enfrentamos debido a la propia naturaleza de nuestro problema en concreto.

2.1 Estado del arte

Un sistema de recomendación [6] es un sistema de filtrado de información sobre un determinado tipo de elementos, cuyo objetivo es encontrar recomendaciones para un usuario determinado. Existen diferentes enfoques [7], el basado en contenido, el filtrado colaborativo, el basado en conocimiento y enfoque híbrido. En el primer caso se emplean técnicas de recuperación de información, comparando el contenido a evaluar con el contenido del perfil del usuario. En el segundo, el filtrado colaborativo, se calcula la similitud del usuario con otros perfiles; es decir, no se analizan los elementos a recomendar, sino las similitudes entre usuarios. Se parte de la premisa de que, si lo mismo que le gustó al usuario les gustó a otros usuarios, probablemente tengan las mismas preferencias y también le gusten los nuevos elementos que también les gustaron a ellos. Los sistema de recomendación basados en conocimiento realizan recomendaciones partiendo de la información que el usuario proporciona sobre sus necesidades en conjunto con el conocimiento sobre los productos a recomendar, buscando las mejores coincidencias entre ambos. Por último, los enfoques híbridos realizan recomendaciones combinando información de diferentes vías, como historiales de compra e información sobre características de usuario y productos, beneficiándose de las ventajas de

las estrategias anteriores.

Por tanto, los sistemas de recomendación habituales tratan de codificar las propiedades de los elementos y las preferencias de los usuarios hacia los mismos. Suelen utilizar datos como registros de compra o historiales de navegación, añadiendo en ocasiones información complementaria sobre el producto, normalmente en forma de texto. Además, este tipo de sistemas suele dejar de lado la representación visual de los objetos, en los que se encuentran reflejadas implícitamente ciertas características. En el caso de restaurantes, las fotos de la comida, el ambiente o la vista desde el mismo pueden llegar a ser muy importantes. Cuando un usuario acompaña de imágenes una valoración, suponemos que la intención es reflejar y/o complementar las características que suscitaron esa crítica. Y como se suele decir, una imagen vale más que mil palabras, por lo que en ciertos contextos sería muy interesante poder integrar y analizar toda esta información.

Uno de los primeros intentos de utilizar imágenes en sistemas de recomendación es He et al.[8]. Se trata de un sistema basado en factorización, en el que la descripción de los elementos se facilita a través de las características extraídas por una red convolucional a partir de una imagen. La principal diferencia en relación a nuestro proyecto es que nosotros pretendemos definir tanto el perfil del usuario como el de los restaurantes con las imágenes. En general, nuestro objetivo es encontrar cuáles son las características que le agradan a un usuario, que definen por tanto sus gustos, y si esas características se encuentran reflejadas en las imágenes de un nuevo restaurante de forma representativa. En [9] propusieron un sistema de recomendación de puntos de interés o POI (por sus siglas en inglés) que mejoraba con la inclusión de contenido visual, adaptando una red convolucional para extraer características de las imágenes.

Chu et al.[10] investigaron la efectividad de la información visual en la recomendación de restaurantes en el año 2017. A partir de artículos de blogs, se integraron las características de las imágenes extraídas por una red convolucional junto con el texto que las acompañaba en un recomendador, pero usando un clasificador SVM (*Support Vector Machine*) para separarlas previamente en cuatro categorías: comida, bebida, interior y exterior del local. En este caso, las imágenes no están asociadas a los usuarios, sino a los blogs. Aunque los resultados demuestran que el uso de imágenes mejora el rendimiento hasta cierto punto, no es significativo. Sin embargo, en el apartado de conclusiones, los autores comentan que como trabajo futuro tratarán de explorar más este enfoque, buscando integrar la información visual de manera más efectiva, con mejores medidas de similitud o diferentes pesos en las imágenes, por ejemplo.

También es interesante destacar FoodDist, un modelo de código abierto diseñado por Yang

et al. [11] que analiza imágenes de alimentos para hacer recomendaciones saludables a personas con diversas enfermedades, como diabetes u obesidad. El funcionamiento general se basa en aprender las preferencias de alimentos de los usuarios a partir de comparaciones de imágenes, obteniendo buenos resultados.

Ya más relacionado con nuestro problema, Zhang et al. profundizaron en la recomendación de restaurantes en [12] y [13] utilizando también datos extraídos de TripAdvisor, pero sin utilizar imágenes. Finalmente, Amis [14] utiliza redes convolucionales para seleccionar las fotos más relevantes que TripAdvisor muestra en los resultados de las búsquedas, aunque no de manera personalizada.

Como mencionamos anteriormente, el objetivo de este proyecto es mejorar los resultados de un sistema de recomendación, concretamente un sistema de recomendación de restaurantes. Para ello, crearemos diferentes sistemas de clasificación binaria (*me gusta* o *no me gusta*) para las imágenes de las valoraciones, etiquetadas en función de la puntuación que recibió el restaurante por parte del usuario que publicó la crítica. La idea es que un recomendador final use un modelo equivalente al que estamos desarrollando aquí, con la misma función de pérdida, que representa al fin y al cabo lo que está aprendiendo el modelo. Por tanto, el mejor descriptor de características que encontremos para nuestro sistema de clasificación supondrá también una mejora para el recomendador. Es importante recalcar que no hemos encontrado en la literatura otros estudios para comparar los resultados obtenidos con metodologías estándar para la resolución de este tipo de problemas.

2.2 Dificultades en la clasificación de imágenes

La clasificación de imágenes consiste en, dada una imagen de entrada y un conjunto de categorías prefijado, asignar a dicha imagen su etiqueta correspondiente en función de la categoría a la que pertenezca. Aunque para un ser humano esta tarea es completamente trivial, tengamos en cuenta que la información con la que lidia un ordenador para resolver este tipo de problemas es un array tridimensional (canales de color R,G,B) con el nivel de intensidad de cada uno de los píxeles de la imagen. Tengamos ahora en cuenta los siguientes factores:

- Tamaño: los objetos en el mundo real tienen diferentes tamaños, no es común encontrar un estándar. Además, una fotografía puede ser tomada desde multitud de distancias al objeto de interés.
- Punto de vista: dependiendo de la orientación que tenga el objeto en relación a la cámara pueden obtenerse muchísimos ejemplos diferentes del mismo objeto.

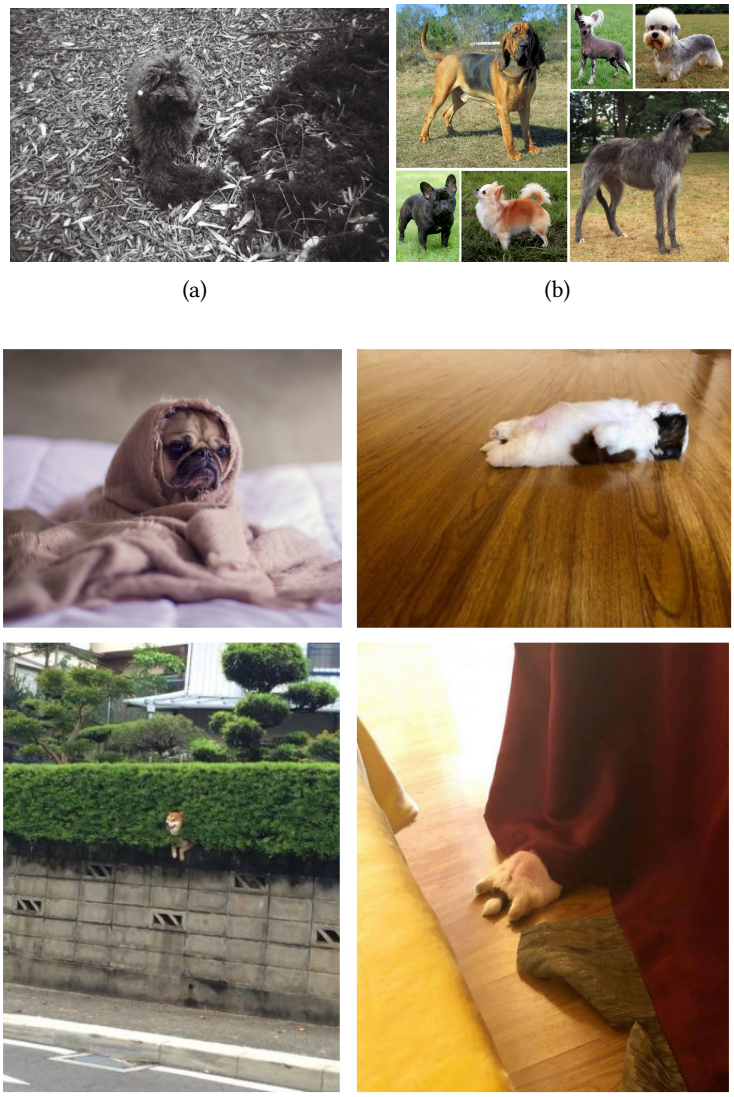
- Iluminación: teniendo en cuenta que la única información que un ordenador tiene sobre los objetos es el nivel de intensidad de los píxeles de la imagen, nos podemos imaginar los infinitos ejemplos del mismo objeto que nos podríamos encontrar con pequeños cambios en la iluminación.
- Fondo: como podemos ver en la Figura 2.1 (a), cuando se trata de imágenes, a veces el fondo puede ser llegar a ser un gran problema.
- Variación intraclase: en el mundo real un mismo objeto puede tener múltiples formas, de hecho, lo común es que así sea, como podemos ver la Figura 2.1 (b). Es decir, que depende de la propia morfología de los elementos de la misma clase.

Por último, observemos la composición de imágenes de la Figura 2.1 (c). En todas ellas podemos identificar a un perro. Y si puede ser complicado para nosotros, imaginemos para un clasificador convencional, que básicamente busca encontrar un patrón que identifique a todos los objetos pertenecientes a cada clase. Por ello, la idea es crear un modelo capaz de extraer características de las imágenes antes de llevar a cabo la fase de clasificación. Esto se consigue con técnicas tradicionales que extraen características de textura, forma o color, por ejemplo, redes convolucionales. Nosotros utilizaremos estas redes y autoencoders para realizar esta tarea, y sobre su funcionamiento y estructura hablaremos en profundidad en la Sección 3.2.

2.3 Dificultades en la recomendación de restaurantes con imágenes

Nuestro problema presenta dificultades adicionales, y es que si hablamos de restaurantes, las características que motivan esta decisión no siempre son tangibles. En nuestros datos de entrada, consistentes en imágenes con una etiqueta de *me gusta* o *no me gusta* asociada, podemos ver información relacionada con la luminosidad del local, cómo es su mobiliario, si la comida está bien presentada, etc. Pero si hablamos de restaurantes, calidad y precio son dos variables que nos vienen rápidamente a la mente. Si nos referimos a la primera, la calidad no se refleja únicamente en el aspecto de la comida, el sabor tiene la mayor parte del peso, y esto no se puede captar en una imagen.

Pongámonos en el caso de que en nuestro conjunto de datos hay un usuario al que le encantan los restaurantes de comida india, pero no si en sus platos usan demasiadas especias. Puede haber dos valoraciones contrarias, a pesar de que el aspecto del local y la comida sea el mismo y por tanto, las imágenes presenten las mismas características, generando una gran confusión en términos de distribución de clases. Si este usuario tiene valoraciones de varios



(a) (b)

(c)

Figura 2.1: Ejemplos de algunos problemas típicos en la clasificación de imágenes: (a) fondo[4], (b) morfología[5], y (c) contexto[6].

restaurantes indios, normalmente positivas, donde sirven comida regional y la decoración presenta una temática similar, el sistema aprende, en la fase de entrenamiento, que las características que sus imágenes proyectan suelen agradar al usuario. Sin embargo, en una fase posterior de validación se podría presentar una imagen con las mismas características, que el sistema recomendaría al usuario, pero el sabor, precio o incluso estado de la comida, fueron absolutamente deficientes, y de ahí su baja valoración. A pesar de que las características fuesen extraídas correctamente de las imágenes y el sistema esté tomando una buena decisión al recomendarle al usuario ese restaurante, por reunir las características que normalmente le gustan, esta predicción se califica como errónea. La ocurrencia de este tipo de situaciones durante la fase de entrenamiento implica un reajuste de pesos. Por tanto, se generan problemas tanto a la hora de entrenar como de evaluar el sistema.

También es importante comentar, que si algo que caracteriza a TripAdvisor en cuestión de restaurantes, es que los usuarios publiquen junto a sus valoraciones imágenes de la cuenta y la carta. Es habitual acudir a esta plataforma buscando este tipo de información, ya que en muchos casos no se podría obtener sin ir directamente al local. Y esto, en nuestro caso, es un problema. La intención es que el modelo propuesto aprenda que este tipo de características no suelen ser distintivas en las decisiones finales, ya que este tipo de imágenes se pueden encontrar tanto en valoraciones favorables como desfavorables. Sin embargo, es probable que en muchos casos la diferencia de los precios de las cartas o tickets sí sea un factor decisivo, y la información que utilizamos no sea suficiente para que el sistema pueda aprender a verlo.

Capítulo 3

Metodología

El objetivo de este proyecto es aprender una función F capaz de extraer un vector de características (descriptor) de las imágenes de valoraciones a restaurantes realizadas por usuarios de Tripadvisor. Para encontrar este descriptor utilizaremos un sistema de clasificación binario, cuya arquitectura general presentaremos a continuación. La idea es que posteriormente un sistema recomendador haga uso de esta representación para detectar aquellas características que definen los gustos de cada usuario en particular. Para encontrar esta función se aplicaron diferentes técnicas de Deep Learning, concretamente redes convolucionales y autoencoders.

3.1 Arquitectura del sistema

El objetivo de nuestro sistema es encontrar el vector de características que mejor describa la imagen, en el contexto de recomendación de restaurantes. Para obtenerlo proponemos una arquitectura general con tres variantes, que explicaremos a continuación, en las que los datos de entrada consistirán en el ID de usuario, ID de restaurante e imágenes extraídas de valoraciones, etiquetadas con su correspondiente puntuación. Las puntuaciones posibles son 10, 20,30,40 y 50, y su uso es análogo al sistema de valoración por clasificación de estrellas. Una estrella correspondería a una puntuación de 10 y cinco estrellas a una de 50, siendo la valoración más alta posible. En nuestro caso reduciremos a *me gusta* todas las valoraciones con una puntuación superior o igual a 40 y a *no me gusta* las que tengan una puntuación inferior a 40. Cada uno de los tres elementos que componen la entrada, identificadores e imagen, son procesados por separado en el sistema, para luego unir el resultado y procesarlo a través de una red neuronal profunda, que actuará como clasificador.

Dado que nuestro objetivo es determinar la representación más adecuada de las imágenes, el vector que nos interesa es el resultado de procesarlas, y en este punto radica la principal

diferencia de las diferentes aproximaciones que vamos a llevar a cabo. La arquitectura general del modelo que vamos a utilizar se ilustra en la Figura 3.1 (a). Si partimos del punto donde las imágenes están representadas por su vector de características, el modelo es el mismo en las tres aproximaciones. Los IDs de restaurante y usuario son procesados de forma independiente por capas de embedding y las características de la imagen por una capa completamente conectada. La salida de estas tres capas, que tiene las mismas dimensiones, es concatenada en un solo vector. Es importante aplicar una capa de *batch normalization* para normalizar el resultado, puesto que las características manifestadas en cada uno de los vectores (usuario+restaurante+imagen) presentan valores totalmente diferentes.

Una vez obtenido el vector que representa finalmente a cada valoración, éste es procesado por una serie de capas, que compactamos en bloques denominados *reduce blocks* para simplificar y cuya estructura se puede observar en la Figura 3.1 (b). El objetivo de estos bloques es reducir el tamaño de esta representación a la mitad, procesando el vector de entrada a través de la superposición de capas completamente conectadas, con inicialización He Normal y ReLU como función de activación, reduciendo su tamaño progresivamente a la mitad. Se intercalan capas de *dropout* cada dos FC con la intención de buscar una correcta generalización. Tras la sucesión de dos bloques de reducción encontramos una capa FC, con la misma dimensión que el vector de la salida del último bloque de reducción y misma función de activación e inicialización, para finalizar con la última capa FC de una neurona, que será la encargada de generar la salida de predicción de nuestro problema de clasificación binario con la función de activación sigmoide.

Los detalles de funcionamiento de cada una de estas capas, así como las diferentes técnicas que utilizaremos a lo largo del proyecto, se explican a continuación. Pero como comentamos anteriormente, a pesar de modelar un sistema de clasificación, lo que realmente nos interesa es mejorar el resultado del procesamiento de la imagen dentro del mismo para obtener el mejor vector de características que represente los gustos del usuario. Cuanto más acierte el sistema en sus predicciones ante nuevos ejemplos, mejor estará extrayendo las características presentes en las imágenes. Por ello, la diferencia en las tres aproximaciones se encuentra en la forma de llevar a cabo el procesamiento necesario para obtener este vector:

- **Primera Aproximación: CNN pre-entrenada como extractor de características** Se obtiene en una primera fase *offline* el vector de características resultado de procesar las imágenes de las valoraciones con una red convolucional neuronal entrenada con el conjunto de datos ImageNet, que explicaremos en detalle más adelante. En primer lugar, almacenamos los vectores de características de las imágenes junto con el resto de variables (ID de usuario, ID de restaurante y etiqueta) para crear las particiones que

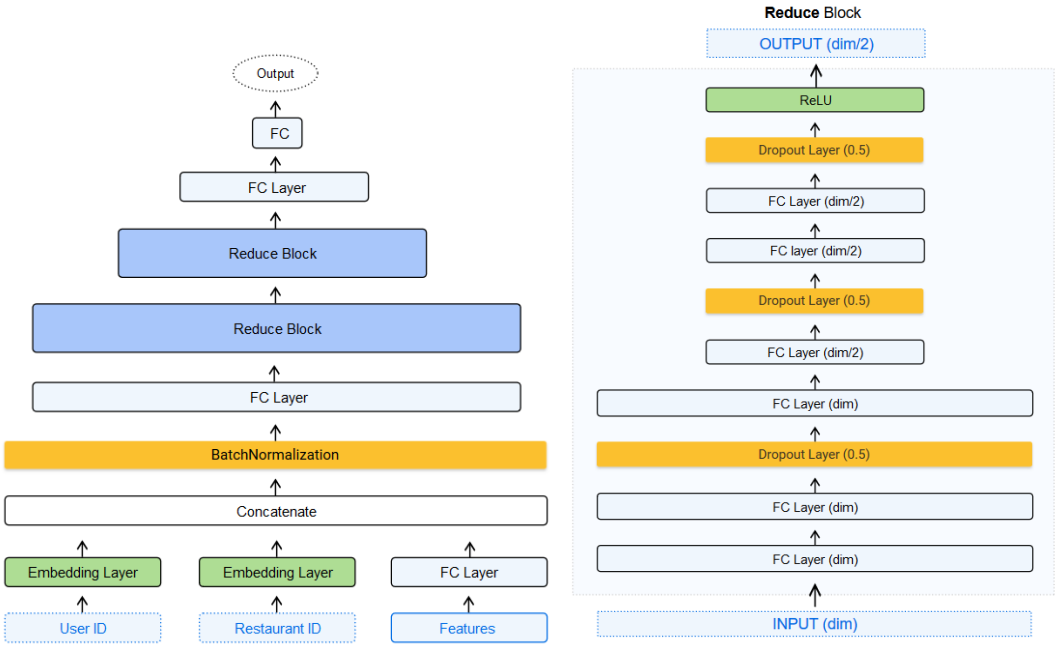


Figura 3.1: (a) Arquitectura general del sistema. (b) Configuración interna de un *reduce block*.

utilizará el sistema como datos de entrada. El procedimiento se refleja en la Figura 3.2 (a), básicamente, el vector de características es la salida de una capa de activación aplicada a la última capa de convolución de la red, que extraemos al obtener el resultado de predicción de las imágenes con la CNN pre-entrenada. Las arquitecturas seleccionadas para ello fueron ResNET50 e InceptionV3, dada su popularidad actual y los buenos resultados obtenidos en numerosos problemas de visión artificial, en las que también profundizaremos a continuación.

- **Segunda Aproximación: *fine-tuning* de una CNN pre-entrenada.** El *fine-tuning* de una red consiste en volver a entrenarla a partir de unos pesos específicos, obtenidos a partir de un entrenamiento previo para resolver un problema diferente. Precisamente por este motivo suele ser necesario reajustar estos pesos volviendo a entrenar con los datos del nuevo problema, pero partiendo de la premisa de que probablemente sea más eficaz continuar a partir de algún tipo de conocimiento que de cero. El proceso es el mismo que en la anterior aproximación, pero en este caso la CNN pre-entrenada forma parte del sistema y es reentrenada durante el entrenamiento. En este caso los datos de entrada están formados por las imágenes de las valoraciones en lugar de su vector de características, y su detección es ajustada en función de nuestros datos de entrada durante la fase de entrenamiento.

- **Tercera Aproximación: *autoencoder*.** Finalmente, decidimos probar el funcionamiento de las redes convolucionales en la extracción de características en comparación a un *autoencoder*. Para ello seguimos un procedimiento similar, como se puede ver en la Figura 3.2 (b), extrayendo la salida de la última capa de parte del *encoder*. Como no disponemos de *autoencoders* pre-entrenados como en el caso de las CNN, fue necesario entrenar uno con las imágenes del conjunto de datos en primer lugar para extraer posteriormente el vector de características que recibe el sistema como entrada.

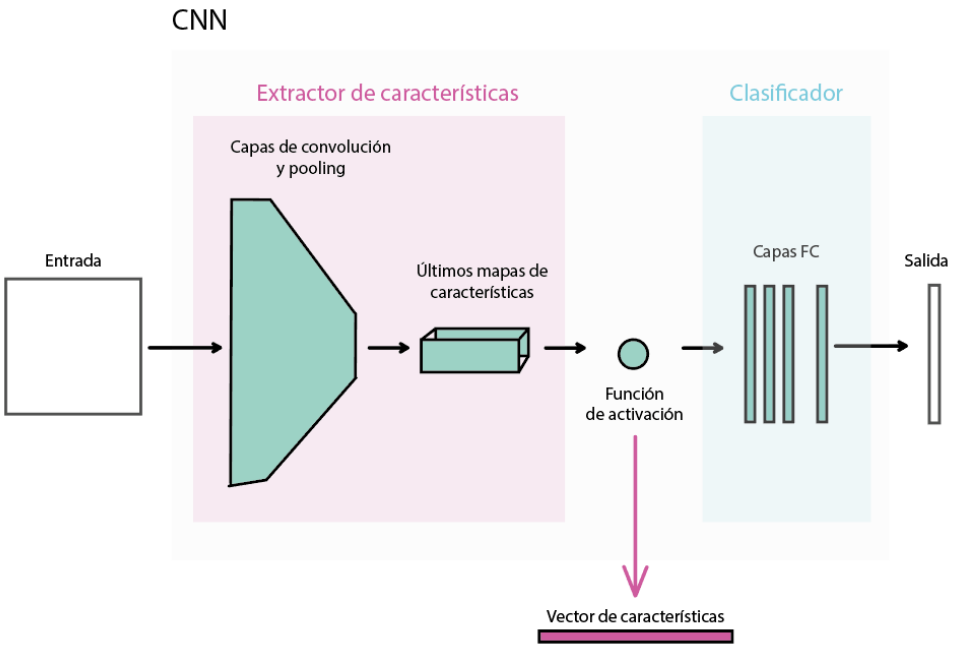
3.2 Redes convolucionales

Una red convolucional o CNN (por sus siglas en inglés) es una red neuronal de aprendizaje supervisado, cuya ventaja principal reside en que presume que las entradas son imágenes, por lo que su arquitectura interna es específica para tratar con este tipo de datos. En las redes neuronales convencionales las neuronas de cada capa se encuentran conectadas a todas las neuronas de la capa anterior. Si tenemos en cuenta que las imágenes tienen tres dimensiones a raíz de los canales de color RGB, podemos ver cómo el número de parámetros crece de manera significativa. La principal diferencia entre las CNNs y otro tipo de redes es que usan conectividad local, por lo que las neuronas en cada capa sólo están conectadas a una pequeña región de la capa anterior.

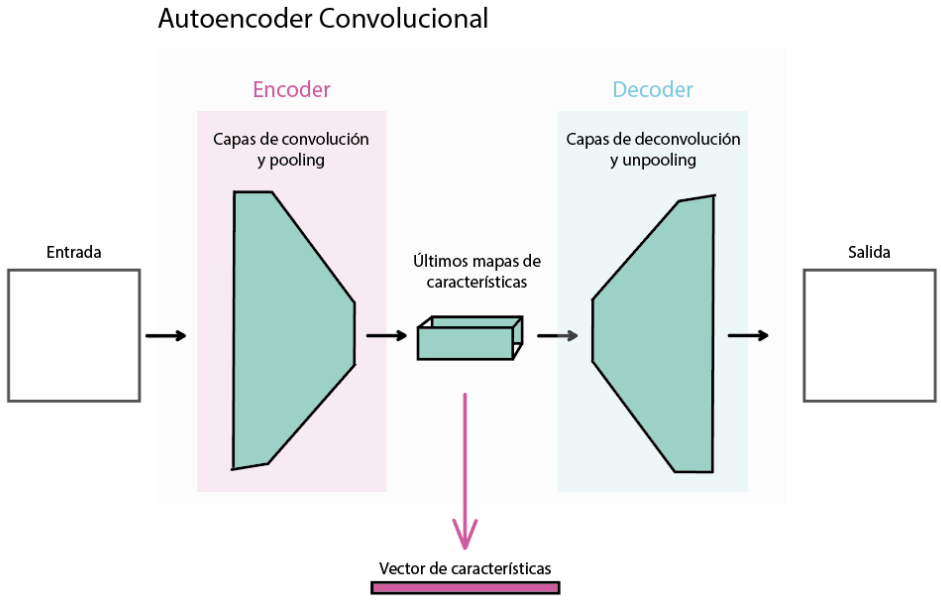
Hoy en día, las CNNs son usadas para resolver todo tipo de problemas, pero para entender el gran éxito y popularidad de estas redes, habría que hablar antes sobre ImageNet. ImageNet[1] es un conjunto de datos etiquetados consistente en más de 14 millones de imágenes categorizadas en alrededor de 21.000 clases. El proyecto fue desarrollado por académicos de las universidades de Princeton y Stanford, entre otras, y su objetivo era promover la investigación en el campo de la visión artificial. En el año 2010 se inaugura el ImageNet Large Scale Visual Recognition Challenge o ILSVRC, cuya finalidad era resolver tareas específicas utilizando como datos de entrada un subconjunto de imágenes procedentes de ImageNet. Entre ellas destacan la clasificación de imágenes, detección de objetos y reconocimiento de objetos. Los conjuntos de datos consistieron en aproximadamente 1 millón de imágenes distribuidas en 1000 clases.

En el año 2012, Krivzhevsky, et al. [1] de la Universidad de Toronto, consiguieron los mejores resultados en las tareas de clasificación de imágenes en el ILSVRC-2010 y el ILSVRC-2012, además de considerarse como una de las publicaciones más influyentes en el campo de la visión artificial. Desarrollaron una red convolucional neuronal, como podemos ver en la Figu-

[1] http://www.image-net.org/



(a)



(b)

Figura 3.2: Procedimiento de obtención del vector de características de la imagen a través de diferentes vías: (a) red convolucional, (b) *autoencoder* convolucional.

ra 3.3, consistente en 5 capas de convolución, capas de maxpooling y 3 capas completamente conectadas, que explicaremos en detalle en la Sección 3.2.1.

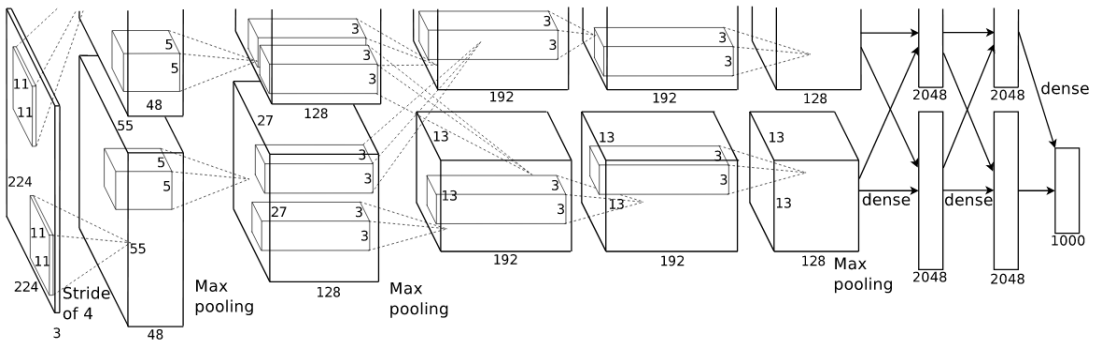


Figura 3.3: Estructura de AlexNet. Ejecutada por dos GPU que solo se comunican en ciertas capas. La entrada de la red es un volumen de 224x224x3 y la salida un vector 1000 dimensional.

Desde ese momento, las CNNs pasaron a considerarse un referente en deep learning, ya que obtenían excelentes resultados en diversas tareas que involucraban la visión artificial, desde recuperación de información hasta detección [15], o incluso segmentación [16]. Los campos de aplicación de las redes convolucionales hoy en día son muy diversos, destacando la conducción autónoma, en la que resolver correctamente este tipo de tareas es fundamental. También son utilizadas en tareas de clasificación de vídeos, imágenes médicas, galaxias o reconocimiento de poses humanas, parte importante del aprendizaje por refuerzo, por ejemplo, en videojuegos. Resaltar, por último, el uso de redes convolucionales en la descripción de escenas, que dada una imagen de entrada son capaces de proporcionar una frase describiendo qué es lo que hay en ella.

3.2.1 Capas de convolución

Las redes convolucionales neuronales procesan sus capas imitando al cortex visual del ojo humano para identificar distintas características en los datos de entrada. Estas características, en definitiva, las que nos permiten identificar objetos. En la corteza visual hay pequeños conjuntos de células sensibles a regiones específicas del campo visual. En un experimento realizado en 1962, Hubel et al. [17] demostraron que algunas células neuronales se disparaban solamente en presencia de bordes de cierta orientación, como horizontales o diagonales, por ejemplo. De ello dedujeron que estas neuronas estaban organizadas de manera jerárquica y juntas producían una percepción visual.

Las CNNs están formadas por varias capas ocultas, especializadas y dispuestas jerárquicamente. En cierta manera, podemos ver similitud en su organización con la del cortex visual que describían Hubel et al.. Concretamente, las primeras capas pueden detectar líneas, curvas o demás formas y características básicas. A medida que se van aumentado las capas, estas se van especializando en base a la información que reciben de las anteriores, para detectar formas más complejas, como gatos, caras o autobuses.

Las capas de convolución son las primeras en procesar la imagen de entrada hasta obtener una salida en una CNN. Para ello, sobre la imagen de entrada se aplica un filtro o kernel, que actúa como el campo receptivo de cada neurona. Podemos hacer una analogía de este campo receptivo con nuestro campo de visión o visión panorámica, ya que a pesar de ver un escenario mucho más grande, nuestro enfoque se centra en un área mucho más pequeña. El área sería el kernel y el escenario la imagen. Matemáticamente, este filtro es una matriz que se desplaza centrándose en cada uno de los píxeles de la imagen de entrada, y sobre su campo receptivo realiza una multiplicación elemento a elemento, como podemos ver en la Figura 3.4. Por eso es necesario que el filtro tenga la misma profundidad que la entrada.

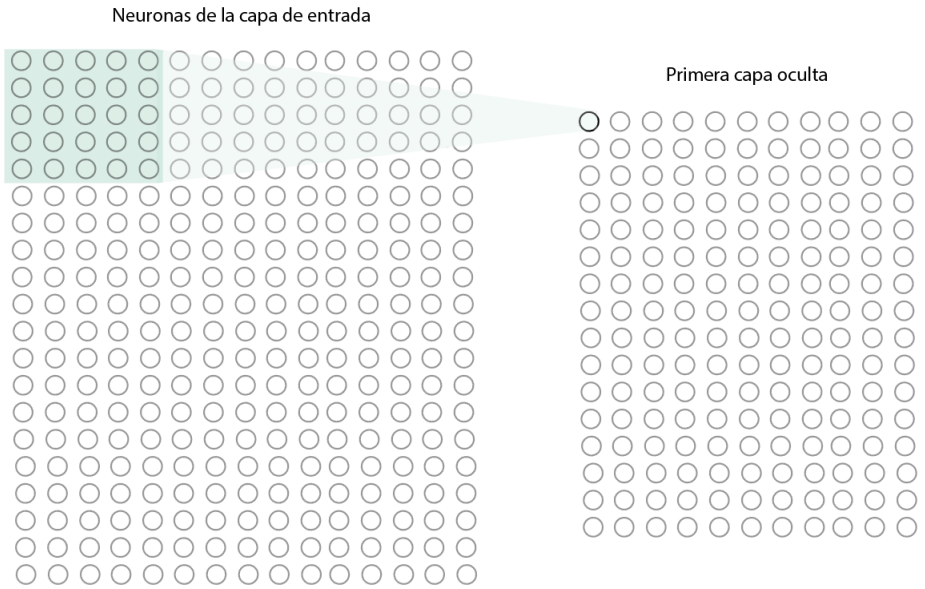


Figura 3.4: Ejemplo de aplicación de convolución con un kernel 5x5 en un volumen de entrada para producir un mapa de activación.

En nuestro problema en concreto, al utilizar imágenes, por ejemplo 224x224 con 3 canales de profundidad para los colores RGB, utilizaremos un filtro de 5x5x3. El resultado de esta multiplicación será el valor de la primera celda de una matriz 220x220x1, ya que no es posible

centrar o "encajar" un filtro de 5x5 alrededor de todos los píxeles de la imagen. Por tanto, lo que perdemos es información en los bordes del volumen de entrada. Esto, y cómo evitarlo con el *padding*, lo veremos más adelante en esta misma sección. El resultado de recorrer toda la imagen formaría el primer mapa de activación o mapa de características. Si decidiésemos aplicar 3 filtros en lugar de uno, tendríamos un volumen de salida de 220x220x3, por tanto, 3 mapas de características.

Si analizamos lo que está ocurriendo realmente con los mapas de activación desde una perspectiva de alto nivel, podemos considerar cada uno de los filtros como identificadores de características. Funcionando del mismo modo que mostraron Hubel y Weisel que se comportaban las células de bajo nivel del córtex visual, detectando bordes, curvas, colores simples y orientaciones, entre otros.

Al situar el filtro sobre diferentes campos receptivos de la imagen de entrada y realizar las multiplicaciones, cuando el patrón reflejado en el filtro coincide en el área observada, obtenemos valores más altos, detectando así esta forma en la imagen. A medida que vamos superponiendo estos filtros y aumentando el número de capas convolucionales, la red es capaz de agrupar estas formas simples para identificar formas más complejas. Por ejemplo, dos líneas rectas unidas por una semicurva podría ser una pata. Esta forma de detección tiene grandes implicaciones en visión artificial, puesto que dos líneas unidas por una curva forman una pata independientemente de la orientación y la localización, y más adelante, unido a otras características, un gato. Concretamente, esto quiere decir que las redes neuronales pueden ser invariantes ante variaciones en tamaño, condiciones de iluminación, puntos de vista o traslación.

Las redes convolucionales van ajustando sus filtros durante el entrenamiento hasta detectar en la entrada las características que conllevan a la salida adecuada. En una imagen tan sencilla como en la del ejemplo, con sólo un canal de profundidad de color, las características son sencillas de detectar. Si pensamos ahora en las imágenes de entrada de nuestro problema, podemos ver que el número de características a identificar para acabar detectando formas tan complejas requiere de una profundidad mucho mayor. En la Figura 3.5 podemos ver un ejemplo de visualización de los filtros de la primera capa convolucional de una CNN.

En una capa de convolución podemos configurar diferentes parámetros, como el número de filtros a aplicar, el *stride* y el *padding*, y de ellos dependerá cuánta información queremos observar o extraer en cada caso.

Stride

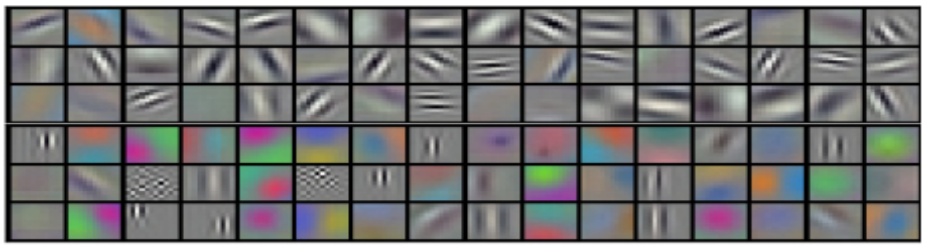


Figura 3.5: Visualización de los 96 filtros de convolución de dimensiones 11x11x3 aprendidos por la primera capa de AlexNet [1], con imágenes de entrada de 224x224x3.

El *stride* [18] es el número de píxeles que avanzamos al recorrer la imagen con el filtro. En los ejemplos anteriores sólo avanzábamos un píxel, recorriendo así toda la imagen de entrada, pero este número es dependiente del problema. Es importante tener en cuenta que cuanto más grande es el stride, más pequeño será el volumen de salida. En la Figura 3.6 podemos ver un ejemplo aplicación de convolución con un filtro 3x3 y stride 2. La primera posición la marca la neurona de color rojo, desplazándola dos casillas a la derecha obtenemos la segunda posición, representada por la neurona amarilla, y así sucesivamente.

Figura 3.6: Ejemplo de aplicación de un filtro 3x3 con stride de 2 en un volumen de entrada de 7x7.

Padding

En el ejemplo del que hablamos al inicio de esta sección, al convolucionar una imagen de 224x224 obteníamos mapas de características de 220x220. El motivo es que no podemos centrar el filtro de 5x5 sobre el primer píxel de la imagen. Si observamos el ejemplo de la

Figura 3.7, con un volumen de entrada más pequeño, vemos como para encajar el filtro en la primera posición necesitaríamos 16 píxeles más alrededor para poder realizar la multiplicación elemento a elemento.

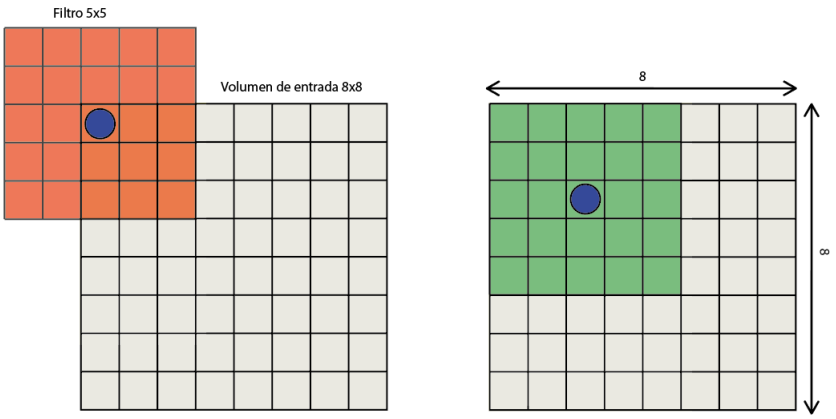


Figura 3.7: Ejemplo de aplicación de convolución con un filtro de 5x5. A la izquierda podemos ver como no es posible centrar el filtro en la primera neurona de la imagen, puesto que no hay datos para realizar la multiplicación elemento a elemento. A la derecha vemos cuál sería la primera posición posible para llevar a cabo la convolución.

Por tanto nos movemos a la primera posición en la que podemos encajar, perdiendo la información resultante de aplicar el filtro en esa celda. Si nos fijamos en la Figura 3.8, podemos ver a la izquierda un volumen de entrada de 8x8, con la primera posición desde la que podrá "observar" una neurona marcada en azul. Las celdas en rojo representan todos esos lugares desde los que no será posible obtener información, y por tanto, menor número de neuronas. Para evitarlo añadimos el número de casillas que permitan realizar la multiplicación elemento a elemento. Esta operación se denomina *padding* o relleno [18] (en este caso *zero padding* al rellenar con ceros, que no afectan de ninguna manera en la aplicación del filtro), es decir, es el volumen que tenemos que aumentar alrededor de la imagen de entrada para no perder información en la convolución.

Tengamos en cuenta que sin usar *padding*, con las sucesivas capas de convolución que apliquemos en la CNN, los mapas de características serán cada vez más pequeños, aunque no sea de manera significativa. En las primeras capas de la red, lo que nos interesa conservar información toda la información posible para poder extraer características de bajo nivel. Para calcular el volumen de salida de una capa convolucional, podemos seguir la siguiente fórmula,

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

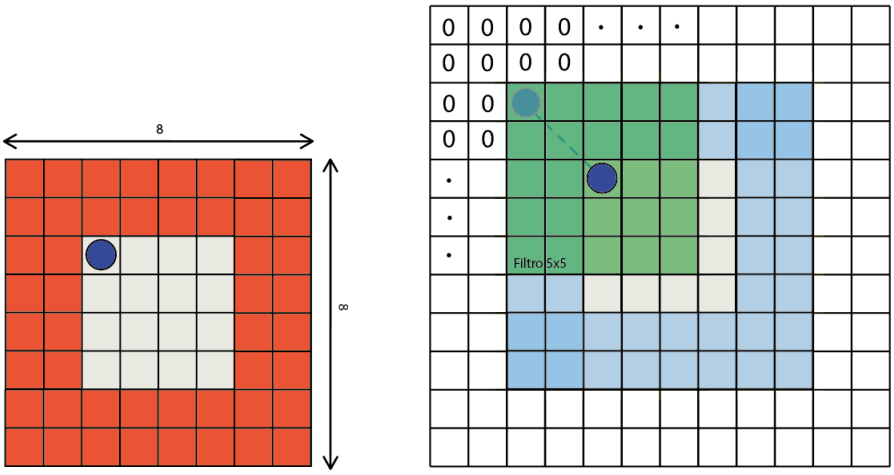


Figura 3.8: Ejemplo de aplicación de *padding*. A la izquierda, región superior izquierda de un volumen de entrada de 8x8 sin *padding*, con la primera posición desde la que podrá "observar" una neurona marcada en azul. Las celdas en rojo representan todos esos lugares desde los que no será posible obtener información, por tanto, menor número de neuronas en la capa de salida. En la imagen de la derecha podemos ver en esa misma región cómo aplicando un *padding* de 2 alrededor de toda la imagen de entrada podremos encajar el filtro de 5x5 en la primera posición para realizar una convolución, obteniendo un volumen de salida con las mismas dimensiones que la entrada original.

dónde O representa el alto/ancho del volumen de salida, W el ancho/alto del volumen de entrada, K el tamaño del filtro, P el *padding* y S el *stride*. Típicamente se configuran estos valores para garantizar que el volumen de salida resultante de realizar la convolución conserve las mismas dimensiones espaciales que el volumen de entrada.

3.2.2 Capas de pooling

Una vez que se detecta una característica, debido a un alto nivel de activación, no nos importa su ubicación exacta en la imagen, sino en relación a otras características. Es por esto que podemos reducir la dimensión espacial del volumen de entrada, para lo que se utilizan diferentes tipos de capas de *pooling* [18]. La operación que se aplica para llevar a cabo esta reducción reducción puede ser calcular la media o la norma L2 o el máximo (*maxpooling*), siendo estas últimas las más comunes. Podemos ver un ejemplo en la Figura 3.9, donde el valor de las neuronas de la siguiente capa es el mayor valor del campo receptivo del volumen de entrada.

En primer lugar, esto implica que la cantidad de pesos o parámetros a aprender se reduce

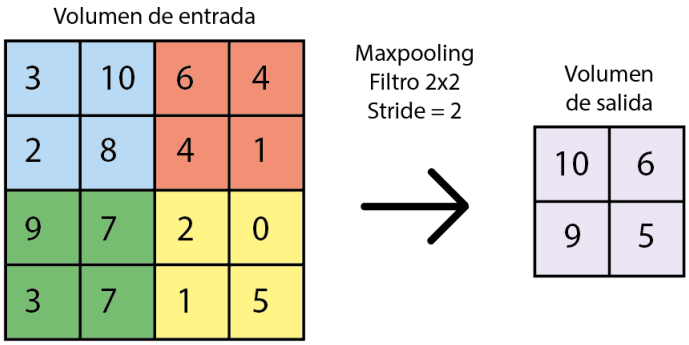


Figura 3.9: Ejemplo de aplicación de *maxpooling* con un filtro de 2x2 y stride igual a 2.

enormemente, lo que lleva a disminuir el coste de cálculo y por tanto, el tiempo de ejecución. Y en segundo lugar, estas capas ayudan a controlar el sobreajuste, ya que de esta manera nos vamos abstrayendo de las formas más simples a medida que avanzamos. De esta forma se motiva el desarrollo de la habilidad de generalización de la red, indispensable para funcionar correctamente ante nuevos ejemplos.

3.2.3 Capas completamente conectadas

En las últimas capas convolucionales de la CNN se encuentran las características generales que, después de un entrenamiento correcto, permiten caracterizar correctamente nuevos ejemplos en las diferentes clases. Esto quiere decir que el sistema fue capaz de extraerlas correctamente de los ejemplos durante el entrenamiento. En relación a nuestro problema, en el que los datos de entrada son imágenes de valoraciones de restaurantes en TripAdvisor, podrían estar resaltadas ciertas características en relación a decoración del local, por ejemplo, si abundan las plantas, el mobiliario es oscuro o refleja ciertos atributos de color o forma. Y en relación a la comida, a través de los diferentes colores, texturas o tamaños que presenta, pudiendo detectar preferencias a ciertos alimentos en concreto.

Después de las capas convolucionales se encuentran, entre otras, las capas completamente conectadas (*fully connected* layers o FC), que actúan como clasificador. Concretamente, una capa completamente conectada recibe un vector derivado de la salida de la capa anterior, y genera un vector N dimensional. Todas las neuronas de una capa FC se encuentran conectadas a todas las neuronas de la capa anterior, a diferencia de las capas de convolución, que como vimos anteriormente hacen uso de una conectividad local. Cuantas más capas FC apilemos, más capacidad de aprendizaje obtendremos al poder emplear más neuronas y pesos para fundamentar la decisión final de clasificación. Sin embargo, es importante tener cuidado con

el sobreajuste y evitar que la red "aprenda" exactamente las características de los ejemplos de entrenamiento, perdiendo la habilidad de generalizar y funcionar correctamente ante nuevos ejemplos de entrada, que es su objetivo.

Si añadimos una capa FC al final, de tamaño igual al número de clases, lo que conseguimos es determinar qué características se corresponden con cada clase en particular obteniendo un valor de pertenencia final, en función de las activaciones presentes en los mapas de características. Si queremos clasificar una imagen en 3 categorías, por ejemplo gato, pájaro o barco, la salida resultante de la CNN será un vector de 3 elementos, generado por la última capa completamente conectada de la red, que se corresponde con el valor de predicción de la imagen para cada clase. Si este vector resultante fuese [0,80, 0,11, 0,09], querría decir que con una probablidad del 80% esa imagen corresponde a la de un gato. En un problema de clasificación binario como el nuestro, *me gusta* o *no me gusta*, el enfoque es diferente, pero el significado es el mismo, como veremos más adelante al introducir las funciones de activación (3.2.7).

3.2.4 Capas de dropout

Si recordamos el ejemplo de la Figura 2.1 (c), dónde hablábamos sobre la dificultad de distinguir a un perro por la multitud de situaciones en las que se nos puede presentar en el mundo real, podemos ver cómo es importante detectar una pata en un gran escenario, no necesariamente las cuatro. Por eso es importante que la red aprenda a generalizar a la hora de realizar correctamente sus predicciones.

El objetivo de las capas de dropout [2] es desechar un conjunto aleatorio de activaciones en una capa, esto es, establecer sus valores a 0. Podemos ver un ejemplo en la Figura 3.10. Las capas de dropout, sólo activas durante la fase de entrenamiento, cuentan con un parámetro p para fijar el % de neuronas que se "desactivan" en cada *epoch*, dejando de propagar su valor a la siguiente capa, que como mencionamos anteriormente son seleccionadas de forma aleatoria. La red aprende que hay ocasiones en que se activan más características que otras, pero si algunas de ellas están presentes, sigue siendo un perro. Esto es muy importante, puesto que, como comentábamos, es inabarcable entrenar recreando todas las situaciones posibles, así como las múltiples combinaciones entre los propios atributos del objeto en cuestión.

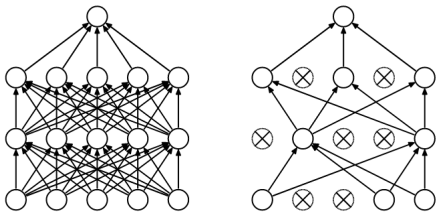


Figura 3.10: Comparación entre una red neuronal regular con 2 capas ocultas (a) y la misma red neuronal aplicando dropout (b) [2].

3.2.5 Batch normalization

Para evitar el problema de sobreajuste que acabamos de mencionar, las capas de *batch normalization* también ayudan, eliminando en algunos casos la necesidad de utilizar capas de *dropout* [19]. Lo que hacen es transformar los datos de entrada a una distribución gaussiana. Normalmente este tipo de capas son utilizadas después de una capa convolucional o completamente conectada y antes de las capas no lineales.

Algoritmo 1 Batch normalization

Entrada: valores de x en cada mini-batch B={x$_1, \ldots, _m$ },γ y β.
Salida: $\{y_i = BN_\gamma, \beta(x_i)\}$

1:
$$\mu B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$$
(media del mini-batch)

2:
$$\sigma^2 B \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu B)^2$$
(varianza del mini-batch)

3:
$$\hat{x_i} \leftarrow \frac{x_i - \mu B}{\sqrt{\sigma^2 B + E}}$$
(normalización)

4:
$$y_i \leftarrow \gamma \hat{x_i} + \beta \equiv BN_\gamma, \beta(x_i)$$
(escalado y desplazamiento)

El algoritmo 1 incluye los pasos seguidos en esta técnica, con la que se pretende forzar a los datos de entrada a convertirse en una unidad gaussiana, por tanto, normaliza las entradas

a cada capa. La media y varianza obtenidas para cada mini-batch no se recalculan para los conjuntos de test, si no que se estiman a raíz de los valores obtenidos en el entrenamiento.

Las ventajas de usar esta técnica residen en que mejora el flujo de gradientes a través de la red, por lo que reduce la fuerte dependencia de la inicialización y permite tasas de aprendizaje más altas. Actúa como una forma de regularización, reduciendo en algunos casos la necesidad de capas de *dropout*.

3.2.6 Capas de embedding

El *embedding* [20] es una técnica para mapear variables categóricas discretas a un vector de números continuos. En el contexto de las redes neuronales, los *embeddings* son representaciones vectoriales continuas de baja dimensión, aprendidas por la red a partir de variables discretas.

Toda la información introducida en una red neuronal debe ser numérica y en determinados casos, este valor no tiene ninguna implicación. Por ejemplo, en nuestro caso queremos representar usuarios a través de su ID, por lo que los numeramos secuencialmente para que la entrada sea un valor numérico. Si tenemos cinco usuarios, pasaríamos a tener a los usuarios 1, 2, 3, 4 y 5, pero estos números no tendrían ninguna relación entre ellos, como en el caso de otro tipo de entradas. Es decir, que el ID de un usuario sea 3 y el de otro 1 no implica que un ID sea mayor que otro, simplemente distinto, a diferencia de si trabajamos con otros conceptos como edad o coste.

Una técnica comúnmente utilizada para representar estas variables es la codificación *one-hot* [18], que consiste en representar cada valor con un vector n-dimensional, siendo n el número de valores únicos en el conjunto. En cada muestra se mapea la variable discreta a un vector, en el que cada una de las posiciones identifica cada uno de los posibles valores que la variable puede adoptar. Todas las celdas del vector contienen un 0 a excepción de la que representa la categoría específica a la que pertenece el ejemplo, que contendrá un 1. En el ejemplo anterior, el usuario 1 se representaría como [1,0,0,0,0] y el usuario 3 como [0,0,1,0,0]. Si pensamos en nuestro problema, con miles de usuarios y restaurantes que representar, vemos que el espacio necesario para representar tan poca información es demasiado grande. Utilizar una capa de *embedding* en la red que reciba el el número de usuario es análogo a utilizar una capa completamente conectada que reciba la codificación *one-hot* del usuario, pero sin tener que preocuparse por el proceso de codificación y ahorrando espacio de almacenamiento.

Los *embeddings* son aprendidos por la red, inicializando sus pesos aleatoriamente y ajus-

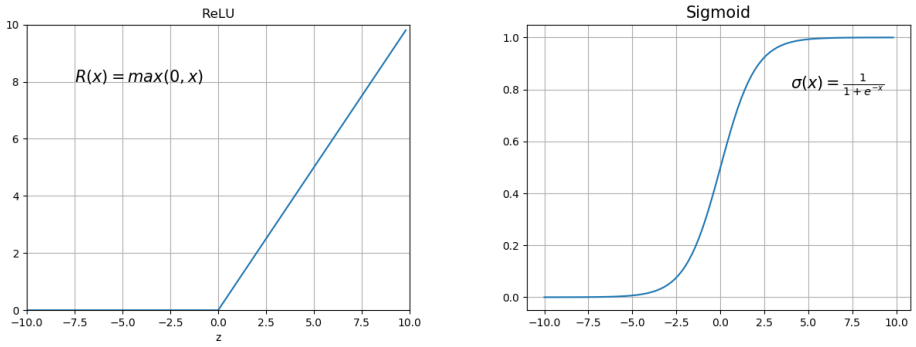
tándolos posteriormente durante el entrenamiento. Si seguimos con el ejemplo anterior, eligiendo una dimensión de la capa de *embedding* de 3, una posible representación para el usuario 1 tras procesar su número sería el vector [0.22,0.01,0.9]. Las capas de *embedding* no sólo nos permiten representar variables categóricas utilizando unos poco números, sino que también sitúan categorías similares cerca unas de otras en el espacio de *embedding*, muy útil para visualizar relaciones entre categorías y agrupar elementos cercanos de acuerdo a algún criterio, por lo general distancia o similitud.

3.2.7 Funciones de activación

Las funciones de activación [18], también llamadas funciones de transferencia, son funciones matemáticas que determinan la salida de una capa de una red neuronal. Son usadas por cada una de las neuronas que conforman la red para determinar si deben de activarse o no según si la información que reciben es relevante para la predicción del modelo. En redes neuronales, datos numéricos, llamados inputs, son recibidos por las neuronas de la primera capa, en nuestro caso píxeles. Cada neurona tiene un peso, y multiplicando ese peso por el input de esa neurona obtenemos la salida de la misma, que es transferida a la siguiente capa. Por tanto, podemos ver estas funciones como una especie de puerta matemática entre lo que una neurona recibe y lo que devuelve, que puede ser tan simple como activarse o no en función de si este valor es superior a 0, como veremos más adelante.

Pueden ser lineales, como la función identidad, o no lineales, que son las más usadas en redes neuronales por proporcionar una mayor capacidad de aprendizaje, esencial para trabajar con datos complejos. Una característica fundamental que deben presentar las funciones de activación es que deben ser computacionalmente eficientes, ya que estos resultados son computados en millones neuronas. Además, en aprendizaje máquina, se usa una técnica llamada *backpropagation* [21] para actualizar los pesos de red durante el entrenamiento, un método de cálculo del gradiente para propagar los errores hacia atrás. Esto es, cuando el sistema recibe una imagen de entrada al entrenar, ésta atraviesa todas las capas del modelo hasta generar una salida. Esta salida se compara con su etiqueta, la salida deseada, y el error generado por la señal se propaga hacia todas las neuronas que contribuyeron directamente a dicha salida. Sin embargo, esta señal no se propaga de forma íntegra a todas las neuronas del modelo, si no que la fracción que le llega a cada neurona es en función de la contribución relativa que esa neurona haya hecho a la salida. De ahí la importancia de que las funciones de activación sean eficientes, porque no sólo es necesario calcular la salida para cada neurona con ellas, sino también la influencia que ésta ejerció en la salida y modificar sus pesos en consecuencia.

Podemos diferenciar entre dos tipos de funciones de activación, las que se utilizan en la última capa completamente conectada para obtener la salida y las que se utilizan en capas intermedias para introducir la no linealidad del sistema. En este proyecto se utilizan dos tipos de funciones de activación que explicaremos a continuación: la función ReLU, usada a lo largo de todo el modelo, y la función sigmoide, utilizada en la última capa.



(a) (b)

Figura 3.11: Funciones de activación (a) ReLU, (b) Sigmoide.

Rectified linear units (ReLU)

Como acabamos de ver, las funciones de activación deben ser computacionalmente eficientes, y las capas ReLU [22] aplican la función

$$f(x) = max(0, x)$$

a todas los valores del volumen de entrada, por lo que solo cambian todas las activaciones negativas a 0 (Figura 3.11 (a)). Estas capas aumentan las propiedades no lineales del modelo sin afectar a los campos receptivos de las capas convolucionales, por lo que son comúnmente usadas con imágenes. También permiten a la red entrenarse más rápido, y cuando se trabaja con gran cantidad de datos, este factor adquiere especial importancia.

Sigmoide

La función sigmoide [23] normaliza los números de entrada en un rango [0,1], como podemos ver en la Figura 3.11 (b), aplicando la función

$$\phi(z) = \frac{1}{1 + e^{-x}}$$

Si los valores de entrada son muy altos la salida será cercana a 1 y si son muy negativos cercana

a 0. Puede considerarse como la tasa de activación de una neurona, pudiendo no activarse (0) o activarse en su máxima frecuencia (1).

Sin embargo el uso de esta función podría traer graves problemas de saturación de neuronas. Si la entrada es igual a 10 o -10, debido a la zona plana en la función, el cómputo del gradiente local será 0. Al hacer backpropagation multiplicando éste por el gradiente de flujo, el resultado también será 0, por lo que no habrá flujo de señal de la neurona a sus pesos y recursivamente a sus datos. Por tanto, es necesario prestar especial atención al inicializar los pesos de las neuronas sigmoide, ya que si los pesos son muy altos la mayoría de las neuronas se saturarían y la red apenas aprendería. En nuestro caso, usaremos esta función solamente en la última capa de la red, con una única neurona como resultado de predicción de clase, concretamente para la clase *me gusta*. Valores muy altos en la salida implican una alta probabilidad de pertenencia a esta clase, por tanto, durante el entrenamiento se busca que el valor que obtienen las imágenes de esta clase sea suficientemente mayor que el que obtienen las que no lo son.

3.2.8 Inicialización

Como vimos anteriormente, *backpropagation* es la forma en la que las redes neuronales ajustan los pesos de la red para aprender correctamente en función de la salida obtenida y la salida deseada o etiqueta. Recordemos que valores muy altos o bajos usando la función sigmoide podrían desembocar en una saturación de las neuronas y no habría aprendizaje al convertir automáticamente toda señal recibida en 0. Este y otro tipo de problemas puede presentarse con la inicialización de los pesos de las neuronas.

Por ejemplo, si iniciamos todos los valores a 0 todas las neuronas acabarían haciendo lo mismo. Dado que todos los pesos son 0, cada neurona acabaría teniendo básicamente la misma operación, y como todas van a dar la misma salida, también tendrán el mismo gradiente. Por eso acabarían por actualizarse de la misma manera y por tanto, todas serían exactamente iguales. Dado que nos interesa que las neuronas aprendan a distinguir diferentes cosas para sacar conclusiones después con toda la información en conjunto, esto no sería buena idea.

Si nos encontramos con valores de inicialización excesivamente bajos, al multiplicarlos constantemente entre sí y estar tan próximos a 0, al propagarlos por la red acabarían tendiendo a 0, por lo que tampoco habría aprendizaje. Si por el contrario decidimos inicializar los pesos con valores excesivamente altos, casi todas las neuronas estarían completamente saturadas a causa de la función de activación y el cálculo del gradiente sería 0, por lo que los pesos no se

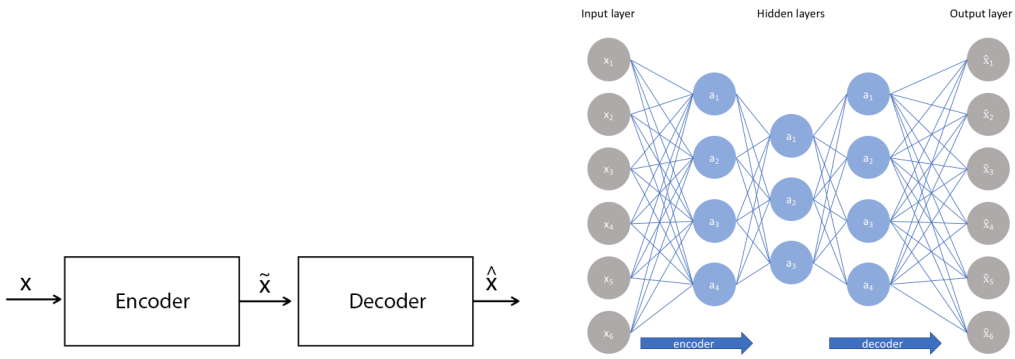
actualizarían. Para intentar evitar este tipo de problemas, lo que intentamos es que la varianza de los valores de entrada sea igual que la varianza de los valores de salida. En este proyecto hemos optado por la inicialización propuesta por He et al. [24], diseñada para mantener la escala de los gradientes aproximadamente igual en todas las capas. Los pesos se inicializan teniendo en cuenta el tamaño de la capa anterior, lo que ayuda a alcanzar un mínimo global de la función de coste de manera más rápida y eficiente. La inicialización de los pesos sigue siendo aleatoria, pero difieren en el rango dependiendo del tamaño de la capa anterior de las neuronas. Esto proporciona una inicialización controlada, por lo tanto, el descenso de gradiente más rápido y más eficiente.

3.3 Autoencoders

Un *autoencoder* es una red neuronal que trata de recrear la entrada en la salida [18]. Básicamente, trata de aprender una aproximación de la función identidad al entrenar para aprender a reconstruir el *input* original. Presenta una estructura simétrica compuesta por tres partes principales: un *encoder*, que comprime la entrada hasta reduciendo su dimensión, esta misma representación reducida producida por el *encoder* llamado *code* y un *decoder*, entrenado para reconstruir la entrada a partir del *code*, donde se encuentran las características extraídas por el *encoder*. Su representación se puede ver en la Figura 3.12 (a), así como un ejemplo de arquitectura simple de un *autoencoder* para tarabajar con datos estructurados en la Figura 3.12 (b). Concretamente, en el modelo del ejemplo podemos ver 3 capas ocultas completamente conectadas. La primera capa oculta actúa como *encoder*, la siguiente capa oculta contiene la representación de bajas dimensiones de la entrada llamada *code* y la última capa actúa como *decoder*, obteniendo así la representación de la entrada original en la salida.

Los *autoencoders* son comúnmente utilizados en diversas tareas relacionadas con imágenes, como compresión [25], reducción de dimensionalidad [26], reconstrucción [27], reducción de ruido [28] y extracción de características [29] [30], motivo por el cual decidimos utilizarlos para este proyecto. Al trabajar con imágenes, necesitamos utilizar un *autoencoder* que procese volúmenes de entrada en lugar de datos simples como en el ejemplo de la Figura 3.12 (b), y como vimos en secciones anteriores, las capas convolucionales son una buena opción pata detectar características en imágenes. A diferencia de los *autoencoders* convencionales, formados por capas completamente conectadas, los *autoencoders* convolucionales utilizan capas de convolución, por lo que podemos aprovecharnos de sus ventajas en la extracción de características.

La estructura que utilizaremos en este proyecto, reflejada en la Figura 3.13, es una adap-



(a) Representación (b) Arquitectura interna

Figura 3.12: Representación y ejemplo de arquitectura simple de un autoencoder. La salida $\hat{x}$, corresponde a la reconstrucción de la entrada, x, generada a partir de la representación de baja dimensión de la misma, $\tilde{x}$.

tación de la arquitectura propuesta por Chollet [31], compuesta por una serie de capas de convolución y *pooling* hasta obtener el vector de características como salida del *encoder*. A continuación se suceden de nuevo diversas capas de convolución y *pooling*, cuyo efecto es exactamente el contrario al de las capas que conforman el *encoder*, para ir reconstruyendo el volumen de entrada hasta obtener un volumen de salida con las mismas dimensiones que el original.

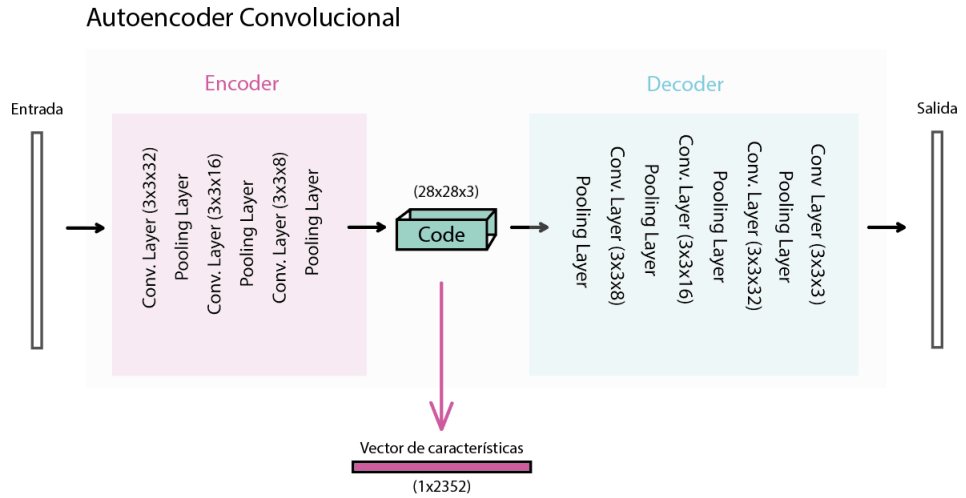


Figura 3.13: Arquitectura del *autoencoder* convolucional utilizado en la tercera aproximación.

Por tanto, los *autoencoders* convolucionales son un tipo de CNNs, cuya principal diferencia radica en que éstos son entrenados para aprender filtros capaces de extraer de forma

óptima aquellas características que puedan utilizarse para reconstruir la entrada original, es decir, que minimicen el error de reconstrucción. El *encoder* está formado por una serie de capas convolucionales, a partir de las cuales obtenemos los mapas de activación que el *decoder* utilizará como entrada para reconstruir la imagen original a partir de una representación más reducida. La idea es utilizar un *autoencoder* convolucional en la parte del sistema encargada del procesamiento de la imagen para llevar a cabo la extracción de características, utilizando posteriormente esa representación de bajas dimensiones.

3.4 Funciones de pérdida

La función de pérdida [18] es la encargada de cuantificar cuán bueno es nuestro clasificador. Dado un conjunto de datos:

$$D = \{(x_i, y_i)\}_{i=1}^{N},$$

dónde x_i es una imagen e y_i una etiqueta, la pérdida de información sobre D es la suma de la pérdida de todos los ejemplos, obtenida al comparar la etiqueta real con la que predice el sistema:

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i(f(x_i, W), y_i)$$

La elección de la función de pérdida dependerá, principalmente, del tipo de problema a resolver, en función de si de trata de un problema de clasificación o regresión. Por ejemplo, un problema de regresión implica predecir una cantidad de un valor real, sin embargo, un problema de clasificación implica asignar a los ejemplos una etiqueta de entre los posibles. Por tanto, el error cometido en ambos modelos no puede calcularse de la misma manera.

En nuestro proyecto, por tratarse de un problema de clasificación binaria, haremos uso de la entropía binaria cruzada o *binary cross entropy* [32]. Los valores de salida de la función de pérdida se encuentran en el conjunto {0,1}, siendo 0 el resultado deseado. Al tener solo dos clases posibles para cada ejemplo, en vez de hallar la probabilidad de que el ejemplo sea de cada clase y elegir la clase con el valor más alto, podemos simplificarlo desde el punto de vista de pertenencia a una sola clase. Concretamente, la función de pérdida que vamos a utilizar tiene la siguiente forma:

$$H_p(q) = -\frac{1}{N} \sum_{i=1}^{N} y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i))$$

Los valores se encuentran en el conjunto {0,1}, y_i es la etiqueta y $p(y_i)$ la probabilidad de que el ejemplo pertenezca a la clase. Por tanto, si el ejemplo pertenece efectivamente a dicha clase, cuanto más alta sea esta probabilidad p, más bajo será el resultado de la función de pérdida.

3.5 Algoritmos de optimización

Si la función de pérdida nos permite cuantificar la calidad de un conjunto de pesos W, el objetivo de la optimización es encontrar la W que minimice la función de pérdida. La idea general es alcanzar el punto más bajo de una pendiente. Si tenemos lo ojos vendados, la forma más intuitiva sería ir desplazándonos por el terreno y sentir la pendiente en diferentes puntos, en función de la inclinación iríamos avanzando hasta que no pudiésemos bajar más. La derivada de una función en un punto determinado nos da el valor de la pendiente. Por tanto, usamos el valor de las derivadas de f_x en x, cualquier punto, para alcanzar el mínimo.

Se llama gradiente al vector de derivadas parciales ∇f_x , por lo que tendrá la misma dimensión que x y cada elemento del gradiente nos dirá cuál es la pendiente de la función f_x si nos movemos en esa dirección de coordenadas. El gradiente apunta a la dirección del mayor incremento de pendiente de la función, y correspondientemente, mirando la dirección negativa del gradiente se obtendrá la dirección del mayor decremento de esta. La pendiente de la función en cualquier dirección es el producto del punto con el gradiente y la dirección del descenso más inclinado es el gradiente negativo. En la práctica, gran parte del trabajo en *deep learning* está en calcular los gradientes de las funciones y usar esos gradientes para actualizar el vector de parámetros iterativamente. Esto es debido a que el entrenamiento de las redes neuronales es, en gran medida, un problema de optimización. Definimos una función de pérdida con valores asociados a diversas clases para cuantificar cuán bueno es nuestro sistema con respecto a sus predicciones. De lo que se trata, en definitiva, es de encontrar esa configuración de pesos de W que minimice el resultado de la función de pérdida, esperando que funcione bien ante nuevos ejemplos y no se hayan producido problemas de sobreajuste, por ejemplo.

El *learning rate* o tasa de aprendizaje es un hiperparámetro que controla en qué medida ajustar los pesos de nuestra red con respecto a la pérdida del gradiente. Básicamente se sigue la siguiente relación:

$$nuevo\_peso = peso\_actual - learning\_rate * gradiente$$

Cuanto más bajo es el valor de *learning rate*, más lento nos moveremos a lo largo de la pendiente descedente de la función de pérdida. Si utilizamos valores bajos podemos asegurarnos

de no perder ningún mínimo local, pero puede llevar mucho tiempo converger.

Uno de los algoritmos de optimización más simples es el SGD (Stochastic Gradient Descent) [33]. Lo que hace es evaluar la función de pérdida en el gradiente actualizando el vector de parámetros en la dirección negativa del gradiente, porque esto da la dirección de mayor disminución de la función de pérdida. Este proceso se repite iterativamente hasta llegar a converger en la zona de mayor descenso de gradiente. Entre otros problemas, este algoritmo es sensible a mínimos locales, quedándose atascado, ya que en esas zonas el gradiente sería 0 y no podríamos avanzar en la dirección negativa del gradiente. Otro aspecto a tener en cuenta es la existencia de zonas en las que los cambios se suceden muy lentamente y la diferencia de gradientes es relativamente baja. Si además consideramos los millones de parámetros de una red convolucional, teniendo en cuenta sus volúmenes de entrada de altas dimensiones, podemos imaginar como esta situación de ligero descenso ocurre a menudo. Por último, si tenemos en cuenta que el SGD calcula la función de pérdida para cada ejemplo, nos podemos hacer una idea del coste computacional que requiere. Además, al no usar una estimación, es muy sensible al ruido, pudiendo llevar mucho más tiempo alcanzar el óptimo.

Con el *momentum* [34] se introducen nuevas variables en el enfoque SGD, la velocidad en relación al tiempo y la fricción. En cada paso se reduce la velocidad actual en función de la fricción (constante) y se añade al gradiente, moviéndose finalmente en la dirección que marca el vector de velocidad y no solo el vector de gradientes. Según se va cogiendo velocidad al avanzar en una dirección de gran descenso de gradiente, aunque se alcance un punto de gradiente 0, todavía habrá velocidad.

En este proyecto se han utilizado diferentes optimizadores, RMSProp [35] y ADAM [36]. El RMSProp, como el SGD+Momentum, también ajusta la velocidad de aprendizaje automáticamente, eligiendo además una tasa de aprendizaje diferente para cada parámetro. Lo que hace es dividir la tasa de aprendizaje por un promedio decreciente exponencial de gradientes al cuadrado. Por ello, se comporta mejor ante oscilaciones. El algoritmo ADAM combina características del RMSProp y el *momentum*, obteniendo ventajas de ambos. Calcula las tasas de aprendizaje para cada parámetro, además de almacenar un promedio decreciente exponencial de gradientes cuadrados pasados. En resumidas cuentas, a la hora de tomar una decisión tiene en cuenta tanto la velocidad como la diferencia de gradientes al cuadrado.

El gradiente es la generalización vectorial de la derivada, es un vector de tantas dimensiones como la función y cada dimensión contiene la derivada parcial en dicha dimensión:

$$\nabla f = \left[\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots \frac{\partial f}{\partial x_n} \right]$$

El gradiente ∇f_x es el vector que contiene la información de cuanto crece la función en un punto especifico x por cada dimensión de nuestra función de forma independiente.

3.6 Transfer learning

Para crear modelos efectivos en *deep learning* suele ser necesario disponer de una gran cantidad de datos, ya que son una parte crítica de la red al extraer directamente las características de los mismos. El *transfer learning* [37], o transferencia de conocimiento, ayuda a solventar este problema. Consiste en obtener los pesos y parámetros de una red que ha sido entrenada previamente en un conjunto de datos de gran tamaño y "ajustar" el modelo con otro conjunto de datos. La idea es que mejore el aprendizaje en la nueva tarea con los nuevos datos a través de la transferencia de conocimiento de una tarea relacionada que ya se ha aprendido.

Utilizando estos pesos obtenemos por tanto un modelo pre-entrenado. Si obtenemos la última capa de la red y añadimos un nuevo clasificador, conseguimos adaptar lo aprendido a nuestro problema en concreto. Existen varias formas de aplicar esta técnica, que usaremos a lo largo de este proyecto. Una de ellas es congelar los pesos de la red del modelo pre-entrenado, esto es, no modificar los pesos durante el entrenamiento, como una especie de extractor de características fijo para el nuevo conjunto de datos. Otra estrategia, consiste en reemplazar y volver a entrenar el clasificador, con la diferencia de que en este caso sí que se ajustan los pesos de la red previamente entrenada para ajustar las características a nuestro problema en particular sin perder las ventajas de lo ya aprendido. Es posible congelar los pesos de algunas de las capas del modelo, como por ejemplo, las primeras. En ellas se encontrarán filtros detectores de características más genéricas, como bordes o manchas de color, útiles para muchas tareas. Las capas posteriores son más específicos para detectar detalles de clases, por lo que es posible que nos interese adaptarlas para trabajar con nuestro problema en concreto.

Hemos seleccionado dos arquitecturas de redes convolucionales muy populares para utilizar en este trabajo, ResNET50 e InceptionV3, que se definen a continuación.

3.6.1 ResNET50

La arquitectura ResNET [3] fue introducida por Microsoft, ganando la competición ILSVRC, mencionada en la Sección 3.2, en el año 2015. Aquí se introdujeron por primera vez los conceptos de aprendizaje residual y *skip connection* o conexión de salto. El aprendizaje residual consiste en conectar la salida de capas anteriores a la salida de las nuevas capas a través de las conexiones de salto, que son conexiones adicionales entre neuronas en diferentes capas

de una red que omiten una o más capas de procesamiento no lineal.

Como toda CNN, la ResNET está formada por una sucesión de capas convolucionales. La novedad está en agregar la entrada original a la salida del bloque de convolución, con los llamados bloques residuales (ilustrados en la Figura 3.14). De esta forma, permiten al

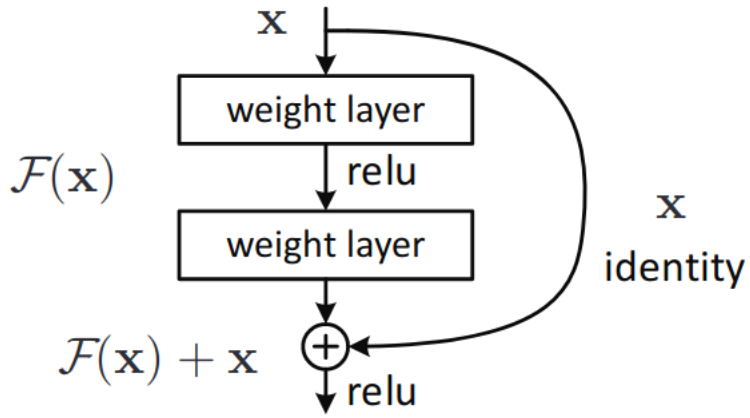


Figura 3.14: Bloque residual [3].

modelo aprender una función de identidad que garantiza que la capa siguiente tendrá un rendimiento al menos tan bueno como la capa anterior, y no peor. De hecho, lo que estos bloques están haciendo, es aprender la diferencia entre la entrada y la salida, partiendo de la entrada inicial. Consideremos $H(x)$ como el resultado de la transformación que realiza un bloque de una red neuronal, al recibir un volumen de entrada x y generar una salida. Ahora vamos a denominar $F(x)$ a la diferencia entre esa salida $H(x)$ y la entrada x, también conocida como diferencia residual, $F(x) = Salida - Entrada = H(x) - x$. Si lo reorganizamos, obtenemos lo siguiente, $H(x) = F(x) - x$.

La principal ventaja de usar un bloque residual es que tenemos una conexión directa a x a través de la función identidad, por tanto, lo que las capas están tratando de aprender es esta diferencia $F(x)$, ya que x se añade directamente antes de dar una salida. La idea es que es más fácil para las capas tener que aprender $F(x)$ y no $H(x)$, y también más fácil de optimizar. De forma adicional, conseguimos mitigar el problema de desaparición de gradientes al permitir una ruta de acceso directo alternativa para fluir a través. Es mucho más fácil empezar con $F(x)$ a 0 y devolver simplemente x, por lo que la ResNET proporciona a las capas un punto de referencia x desde el cual aprender, en lugar de comenzar desde cero. Podemos ver un ejemplo más concreto de la estructura general de sus diversas configuraciones en la Figura 3.15 y de uno de sus bloques residuales en la Figura 3.16.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix} \times 3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix} \times 8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix} \times 23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix} \times 36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix} \times 3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9

Figura 3.15: Arquitectura de las diferentes configuraciones de ResNET para ImageNet [3].

La innovación introducida por ResNET permitía construir redes de innumerables capas con alto grado de precisión, hito no logrado hasta ahora a causa de los problemas de gradientes ya mencionados.

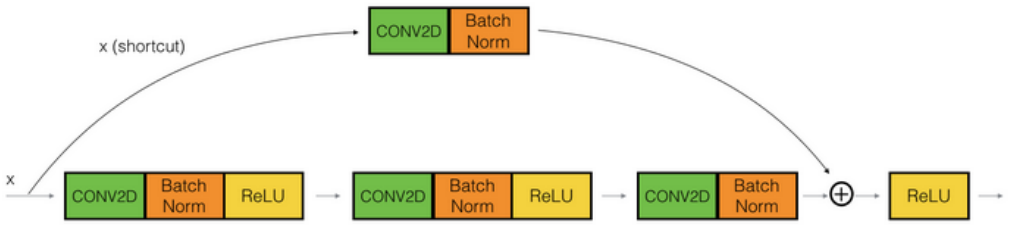


Figura 3.16: Bloque residual de la red ResNET [3]. Para igualar las dimensiones del volumen de entrada con el volumen de salida obtenido tras la sucesión de transformaciones aplicadas en la otra rama, se aplica a la entrada una capa convolucional seguida de otra de batch normalization.

3.6.2 InceptionV3

El ganador del ILSVRC-2014 fue Google, con su modelo denominado Inception, presentado por Szegedy et al. [4] ese mismo año. Podemos ver detalladamente su estructura en la Figura 3.17.

Su arquitectura está compuesta por una sucesión de capas de convolución al inicio, in-

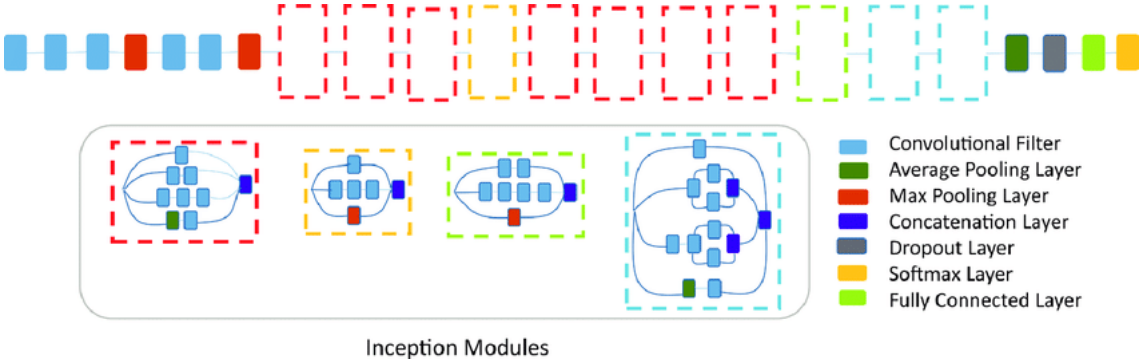


Figura 3.17: Arquitectura de la red Inception v3 [4].

tercalando con *maxpooling*, seguida por los denominados módulos *inception*. Lo que estos módulos hacen es aplicar diferentes transformaciones a la mismos datos de entrada, uniendo los resultados en una sola salida. Si pensamos en una capa convolucional convencional, cada neurona está vinculada a un campo receptivo, la salida de esta pasa a otra neurona con campo receptivo menor o igual, y así sucesivamente. La idea detrás de estos bloques es que diversos filtros de diferentes tamaños puedan acceder a ese mismo campo receptivo a la vez, cada uno de ellos extrayendo sus propias características, y vincular toda esa información a una sola salida. Podemos ver un módulo *inception* en la Figura 3.18 (a).

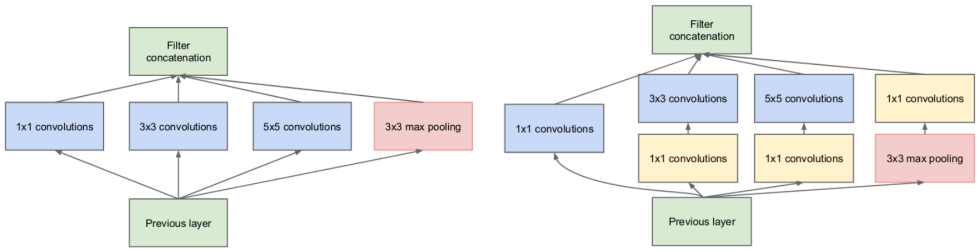


Figura 3.18: Módulos *inception*: : (a) diagrama de la primera versión, (b) versión con reducción de la dimensionalidad [4].

El principal problema que presentaba esta arquitectura era el gran aumento de capacidad computacional que requería. La superposición de diferentes filtros aumenta el número de mapas de características por capa. Para solventarlo, los desarrolladores de Inception usaron convoluciones con filtros 1x1. Aunque en un primer momento no se aprecie su utilidad, por el tamaño de su campo receptivo espacial, estas convoluciones abarcan una profundidad N, que se refiere al número de filtros aplicados por capa. Por tanto, aunque tienen en cuenta un valor

a la vez, es a través de múltiples canales, por lo que pueden extraer información espacial y comprimirla en una dimensión más pequeña. Reduciendo el número de mapas de entrada es posible realizar la superposición de las diferentes transformaciones en paralelo, permitiendo desarrollar redes amplias (operaciones en paralelo) y profundas (número de capas).

En el modelo que nosotros usaremos en este proyecto, InceptionV3 [5], la convolución 5×5 fue reemplazada por dos convoluciones consecutivas de 3×3 (Figura 3.19).

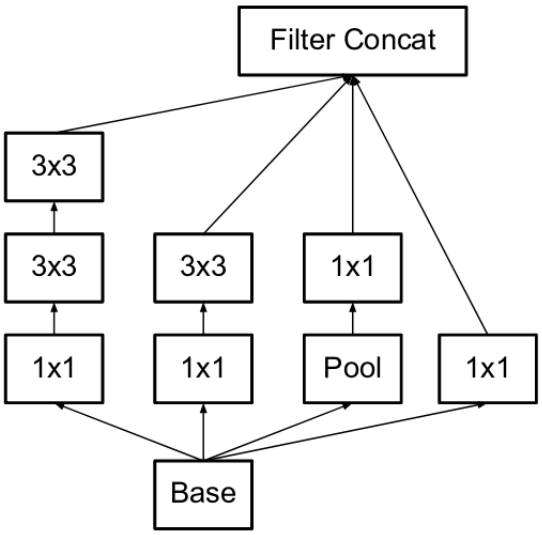


Figura 3.19: Módulos InceptionV3. En relación a su versión original (Figura 3.18), las convoluciones 5x5 fueron reemplazas por 2 convoluciones 3x3 [5]

Capítulo 4

Experimentación

Para llevar a cabo los diferentes experimentos se aplicaron diferentes técnicas de aprendizaje máquina, como *grid search* o *data augmentation*. También fue necesario utilizar técnicas de validación 4.3 y medidas de rendimiento 4.4 de los distintos modelos. Todo esto lo veremos con más nivel de detalle a lo largo de este capítulo.

4.1 Búsqueda de hiperparámetros

Un parámetro en un modelo de aprendizaje automático es una variable de configuración interna al modelo cuyo valor puede aprenderse a partir de los datos, como por ejemplo, los pesos de las neuronas de una red. Al contrario, un hiperparámetro es una variable de configuración externa, y por tanto, su valor no puede ser estimado a partir de los datos. Algunos de los hiperparámetros de nuestros modelos son el *learning rate* o el número de filtros de una capa convolucional.

Como los modelos pueden tener múltiples hiperparámetros, existen diferentes estrategias de búsqueda para determinar los valores óptimos para un modelo dado. En este proyecto se utilizará una de las más simples, *grid search* [38], que consiste en evaluar el rendimiento del modelo con cada una de las diferentes combinaciones de hiperparámetros. En nuestro caso, para cada aproximación se lleva a cabo una fase de búsqueda, analizando el comportamiento del modelo con las diferentes combinaciones de hiperparámetros en la partición de validación tras cada entrenamiento. Posteriormente, se reentrena el modelo con mejores resultados y se evalúa su rendimiento con la partición de test. En el Capítulo 6 explicaremos en detalle que hiperparámetros fueron seleccionados en cada uno de los experimentos.

4.2 Pesos asociados a las clases

Para tratar de suavizar el problema de debalanceo de clases se utilizaron pesos asociados a cada ejemplo a la hora de computar la pérdida durante el entrenamiento en función de su etiqueta. Concretamente, las imágenes de la clase mayoritaria tienen asociado un peso de 1 y para calcular el peso de las imágenes de la clase minoritaria utilizamos la siguiente fórmula:

$$w_i = \frac{M}{N},$$

dónde M es el número de ejemplos pertenecientes a la clase mayoritaria y N a la clase minoritaria. Como acabamos de ver, este valor variará en función del dataset que utilicemos.

4.3 Técnicas de validación

El criterio que se suele emplear para estimar la bondad de un clasificador es su exactitud, es decir, el porcentaje de muestras no vistas durante el entrenamiento que el sistema es capaz de clasificar correctamente. Como en la gran mayoría de dominios, analizar la respuesta del modelo frente a todos los posibles ejemplos que todavía no ha visto es imposible de analizar. Concretamente, en nuestro problema, es imposible predecir el valor que el clasificador asignará a todos los restaurantes que vayan a gustar o disgustar al usuario en el futuro. Por ello, es común estimar esta exactitud analizando la respuesta del modelo frente a un conjunto de datos no usados para el entrenamiento.

Validación Simple

Para validar el comportamiento de nuestro modelo, optaremos por la validación simple. Esto es, dividir el conjunto de datos disponible en dos particiones, una será el conjunto de entrenamiento y otra el conjunto de test, como podemos ver en la Figura 4.1. El conjunto de entrenamiento se usa para construir el modelo y, una vez entrenado, se evalúa su comportamiento con las predicciones que realiza sobre el conjunto de test. Concretamente, el conjunto de entrenamiento se divide en dos subconjuntos, a los que nos referiremos a partir de ahora como *train* y *val*. El conjunto de *train* se usa para ajustar los pesos durante el entrenamiento, es decir, para aprender. El conjunto de *val* es usado después de cada fase dentro del entrenamiento para ver cómo va evolucionando el modelo, y nos otorga cierto control durante esta fase. Por ejemplo, permite detener el entrenamiento si vemos que llegado cierto punto su comportamiento va empeorando progresivamente. Así mismo, es el que utilizamos para ajustar los hiperparámetros del modelo en la fase de optimización. El conjunto de *test* es el que

comprueba realmente la corrección del sistema. Si éste clasifica correctamente C ejemplos de los N que forman el conjunto de test, la exactitud del modelo será $p = C/N$.

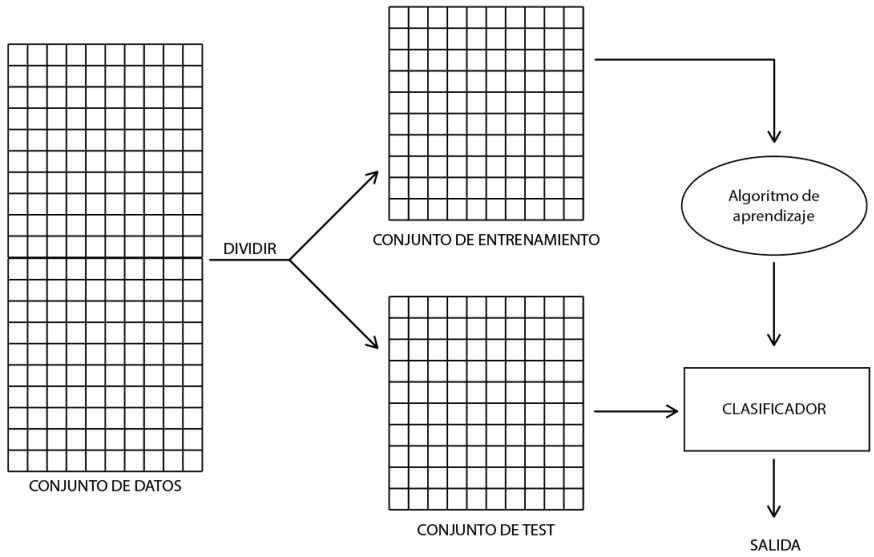


Figura 4.1: División del conjunto de datos inicial en entrenamiento y test.

Al estimar este valor en una pequeña muestra del conjunto de datos que nos gustaría evaluar, el conjunto de test frente a todos los posibles ejemplos no vistos en entrenamiento, necesitamos un método estadístico para encontrar un rango de valores dentro del cual se encuentra el valor de exactitud con una probabilidad o nivel de confianza dado. Para este fin utilizamos en error estándar asociado al valor estimado p, que se calcula como:

$$E = \sqrt{p(1-p)/N}$$

Resumiendo, si nuestro conjunto de test contiene T muestras y clasificamos correctamente C de ellas, la exactitud del modelo sería $p = C/T$ y el error $E = \sqrt{p(1-p)/T}$. Suele ser habitual realizar una división 90% vs 10% o 70% vs 30%, por ejemplo, pero debido a las particularidades de nuestros datos decidimos implementar un algoritmo *ad-hoc*, como veremos en profundidad en el Capítulo 5.

4.4 Medidas de rendimiento

El rendimiento de un sistema se calcula como la comparación entre lo que este hace y lo que debería hacer. En nuestro proyecto, al tratarse de aprendizaje supervisado, podemos

computar la diferencia entre lo que esperábamos obtener y lo que realmente obtuvimos, concretamente, con una matriz de confusión.

Matriz de confusión

A pesar de que la validación simple muestra información importante sobre la exactitud que obtiene el sistema evaluando cómo se comporta frente a un conjunto de muestras no vistas durante el entrenamiento, esta no es suficiente para probar la bondad del modelo, especialmente en problemas con clases desbalanceadas. Por ejemplo, nuestro conjunto de test contiene una distribución de clases aproximada de 80:20, 80% de muestras pertenecientes a la clase A frente al 20% pertenecientes a la clase B (más detalles en el Capítulo 5). Aunque el sistema clasifique incorrectamente todas las instancias de la clase B, obtendría una exactitud del 80%, lo que a primera vista parece un resultado bastante bueno. Sin embargo, si observamos la matriz de confusión comprobaríamos que realmente el sistema no habría aprendido nada.

La matriz de confusión nos dará una idea de cómo está clasificando el sistema, si hay confusión entre clases y en qué medida. Concretamente, se tratará de una matriz de confusión de dos clases, en este caso + (*me gusta*) y - (*no me gusta*), con la siguiente estructura:

		Predicción	
		+	-
Realidad	+	TP	FN
	-	FP	TN

Cada columna de la matriz representa el número de predicciones para cada clase realizadas por el modelo, y cada fila los valores reales por cada clase. Con lo cual las predicciones quedan divididas en 4 clases, TP, FN, FP y TN, que significan lo siguiente:

- TP (*true positives*): Número verdaderos positivos, es decir, de predicciones correctas para la clase positiva.
- FN (*false negatives*): Número de falsos negativos, es decir, la predicción es negativa cuando realmente el valor tendría que ser positivo.
- FP (*false positives*): Número de falsos positivos, es decir, la predicción es positiva cuando realmente el valor tendría que ser negativo.
- TN (*true negatives*): Número de verdaderos negativos, es decir, de predicciones correctas para la clase negativa.

Con estas mediciones podremos calcular métricas más elaboradas que nos ofrecen una visión más general de cómo está funcionando el clasificador, entre las que destacan las siguientes:

- Sensibilidad: tasa de verdaderos positivos. Nos da la probabilidad de que el sistema clasifique correctamente una muestra de la clase positiva.

$$\text{Sensibilidad} = \frac{\text{TP}}{\text{TP + FN}}$$

- Especificidad: tasa de verdaderos negativos. Nos da la probabilidad de que el sistema clasifique correctamente una muestra de la clase negativa.

$$\text{Especificidad} = \frac{\text{TN}}{\text{TN + FP}}$$

- Exactitud: tasa de ejemplos correctamente clasificados frente al total de ejemplos.

$$\text{Exactitud} = \frac{\text{TP + TN}}{\text{TP + FP + TN + FN}}$$

Nuestro objetivo final es extraer correctamente las características presentes en las imágenes. Es igualmente importante clasificar correctamente tanto ejemplos positivos como negativos, lo que implica que las características extraídas durante el proceso de entrenamiento contienen la información necesaria para que posteriormente un clasificador, o recomendador, hago uso de ellas. Por tanto, preferimos un sistema con una exactitud general más baja pero un mayor acierto a la hora de clasificar ejemplos negativos, es decir, nuestro objetivo es encontrar la tasa de clasificación más alta con el mejor balance posible entre sensibilidad y especificidad. La métrica que se suele utilizar es el F1-score, calculada a partir de la siguiente fórmula, en la que entendemos precisión como el ratio de positivos correctamente clasificados entre el total de ejemplos clasificados como positivos:

$$\text{F1-score} = 2 * \frac{\text{sensibilidad * precisión}}{\text{sensibilidad + precisión}}$$

Sin embargo, ésta no tiene en cuenta los verdaderos negativos en el cálculo, por lo que sería escogido un sistema con un 91% de sensibilidad y un 0% de especificidad frente a otro con un resultado de sensibilidad más bajo pero equilibrado, como se refleja en los ejemplos de la Tabla 4.1.

Por este motivo decidimos utilizar una nueva métrica, a la que nos referiremos como B-score (balanced score), que tiene en cuenta ambos factores. Se trata de una variación del

F1-Score, en el que se cambia precisión por especificidad, quedando de la siguiente manera:

$$\text{B-score} = 2 * \frac{\text{sensibilidad * especificidad}}{\text{sensibilidad + especificidad}}$$

Por tanto, tratando de maximizar este valor, estamos buscando un sistema que consiga clasificar el mayor número de ejemplos tanto positivos como negativos, buscando alcanzar una media harmónica entre ambos. De esta manera, si no se consigue clasificar ningún ejemplo de la clase minoritaria este valor será directamente 0.

En el ejemplo de la Tabla 4.1 podemos ver tres modelos con diferentes matrices de confusión, que ilustran los problemas de utilizar las métricas de evaluación más habituales en determinadas situaciones, bastante probables en un problema con nuestras características como es el desbalanceo. Partiendo de un conjunto de datos con 100 muestras positivas y 80 negativas, con una relación 4:1, veamos los resultados obtenidos con las siguientes métricas: exactitud, F1-score y B-score. El primer modelo, M1, no aprendió nada durante la fase de entrenamiento, puesto que clasifica todos los ejemplos como positivos, y sin embargo obtendría unos valores de exactitud y F1-score bastante elevados, incluso más que M2, que obtiene unos resultados mucho mejores a simple vista. Si clasificamos los ejemplos en función de las diferentes métricas, veríamos que solo con el valor de B-score obtendríamos la ordenación adecuada en relación a la matriz de confusión de los modelos. Por eso es tan importante utilizar esta métrica en ciertas fases, como por ejemplo, en la de búsqueda de hiperparámetros a la hora de seleccionar el mejor modelo.

	M1	**M2**	**M3**
VP	100	70	97
FN	0	30	3
VN	0	4	17
FP	20	16	3
Sensibilidad	1,0000	0,7000	0,9700
Especificidad	0,0000	0,2000	0,8500
Exactitud	0.8333	0.6167	0.9500
F1-score	0,9091	0,7527	0,9700
B-score	0,0000	0,3111	0,9060

Táboa 4.1: Ejemplo de cálculo de medidas de rendimiento para tres sistemas diferentes.

Capítulo 5

Conjunto de datos

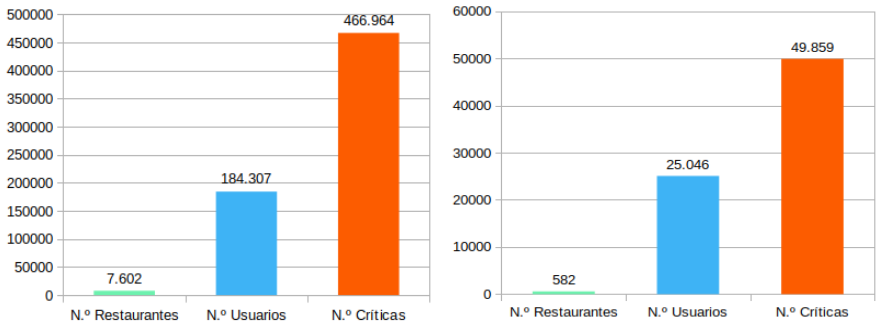
Una de las plataformas digitales de valoraciones más populares es TripAdvisor, Inc., un portal web estadounidense que proporciona críticas de hoteles y restaurantes, reservas de alojamiento, reseñas y otros contenidos relacionados con viajes. Como mencionamos anteriormente, cuenta con 760 millones de comentarios y opiniones, una media de 490 millones de visitantes únicos mensuales y 110 millones de fotografías. En este proyecto haremos uso de las imágenes obtenidas de críticas de TripAdvisor, realizadas a restaurantes en diferentes ciudades españolas.

5.1 Conjunto de datos disponible

Los datos de los que disponemos inicialmente para este proyecto han sido descargados de las valoraciones obtenidas en TripAdvisor por restaurantes de dos ciudades españolas: Barcelona y Oviedo. Estos datos, almacenados en ficheros de tipo .pkl, contienen diferentes diccionarios con información sobre usuarios (users.pkl), restaurantes (restaurants.pkl) y valoraciones (reviews.pkl), así como una carpeta (images) en la que se encuentran organizadas las imágenes asociadas a cada crítica en función del ID de la valoración. Concretamente, cada crítica está representada por fecha, url de las imágenes, índice, idioma, puntuación (de 1 a 5 estrellas), ID de restaurante, ID de usuario, texto, título y URL. Así mismo, para cada usuario disponemos de ID, nombre y localización, y de cada restaurante también tendremos su ID, nombre, ciudad, intervalo de precio (de a 5 €), URL y valoración general (de 1 a 5 estrellas). Dado que nos centraremos en la extracción de características de las imágenes, solo utilizaremos la información del ID para identificar a usuarios y restaurantes y la valoración, disponibles en el archivo reviews.pkl, y las imágenes asociadas.

La base de datos completa de la que disponemos tiene un total de 510.591 críticas, realizadas por 209.763 usuarios a 8.115 restaurantes. Podemos ver la información desglosada por

ciudades en la Figura 5.1. Para simplificar el problema decidimos considerar los datos de cada ciudad de forma independiente. Cabe destacar que, en proporción, ambas ciudades siguen una distribución similar en cuanto a la distribución del número de valoraciones, como se puede ver en la Figura 5.2. De forma general, podemos resaltar tres puntos principales a extraer del análisis:



(a) Barcelona (b) Oviedo

Figura 5.1: Número de usuarios, restaurantes y valoraciones de los datos descargados de TripAdvisor para cada ciudad.

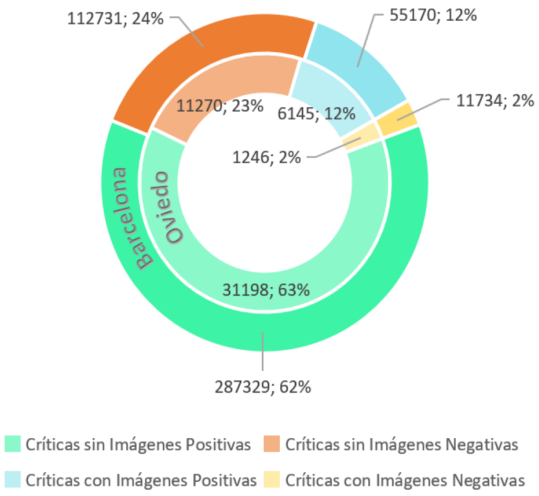


Figura 5.2: Distribución del número de críticas descargadas de TripAdvisor en las dos ciudades.

1. Predominancia de críticas sin imágenes: En el caso de Barcelona, sólo el 14,33% de las críticas disponen de imágenes, contando con un total de 153.707 imágenes. De forma similar, únicamente el 14,82% de las críticas en Oviedo se apoyan con imágenes, reuniendo un total de 17.697. También es importante resaltar la baja media de críticas

por usuario, que en Barcelona es del 1,99% y en Oviedo de 1.55%, así como el promedio de imágenes por crítica, siendo de 2.3% y 2.39% respectivamente.

2. Mayoría de críticas positivas: Con respecto al total de críticas, el 73,35% de las muestras de Barcelona y el 47,90% de las de Oviedo son positivas, y si nos centramos en el caso de aquellas que contienen imágenes, nos situamos en un 82% de críticas positivas en Barcelona y en un 83% en Oviedo. Contamos, por tanto, con una base de datos completamente desbalanceada, aproximadamente con una relación 5:1.

3. Muchos ejemplos de usuarios con una única valoración: Es importante resaltar la gran cantidad de usuarios con sólo una crítica en su historial, representando el 71,50% de los usuarios que apoyan sus críticas con imágenes en el caso de Barcelona y el 77,89% en el de Oviedo. Es decir, no tenemos información completa sobre sus gustos como para poder extraer esas características representativas que motivan sus decisiones finales a la hora de valorar lo que les gusta o no les gusta.

5.2 Conjunto de datos seleccionado

Dado que nuestro proyecto se centra en las imágenes, hemos seleccionado aquellas críticas de usuarios que las contienen, descartando las restantes. Los resultados que presentaremos a lo largo de la memoria han sido obtenidos al realizar experimentos principalmente sobre la partición de Barcelona, probando las configuraciones finales con los datos de Oviedo para observar el rendimiento del sistema en una ciudad más pequeña. Las características presentes en ambos conjuntos de datos se presentan a continuación.

5.2.1 Conjuntos de entrenamiento, validación y test

A pesar de haber analizado y valorado diferentes criterios de selección, como escoger las críticas de los usuarios que tuviesen al menos tres críticas positivas con imágenes o una negativa y otra positiva, decidimos utilizar todas las imágenes sin filtrar ninguna crítica o usuario. Para crear las particiones seleccionadas obtuvimos en primer lugar el conjunto de todas las valoraciones que contenían imágenes. En segundo lugar, procedimos a dividirlas en los subconjuntos de TRAIN y D_test, para lo que fue necesario agrupar todas las valoraciones por usuario, procesándolas teniendo en cuenta las siguientes consideraciones:

1. En Tripadvisor es posible que los usuarios valoren al mismo restaurante más de una vez. Por ello es importante controlar que todas las imágenes procedentes de este tipo de valoraciones estén en el conjunto de entrenamiento. Imaginemos que el usuario X valora

al restaurante Y con 5 estrellas y después con 2. Podría tratarse de una disminución de calidad en el restaurante Y, en cuyo caso ambas críticas reflejarían los gustos del usuario X. Sin embargo, también existe la posibilidad de que el usuario X simplemente haya cambiado sus gustos. En este último caso deberíamos de tratar ambas valoraciones como independientes, ya que no se debería comprobar si el sistema extrae correctamente las características que determinan que al usuario X le guste o no el restaurante Y con valoraciones contradictorias. Por tanto, todas las críticas del mismo par (usuario, restaurante) son destinadas directamente al conjunto de TRAIN.

2. Si solamente tenemos una valoración positiva por usuario, se destina a la partición de TRAIN, procediendo del mismo modo con las valoraciones negativas. Como el problema depende del usuario, es importante que los ejemplos de imágenes que usamos en test provengan de los mismos usuarios que utilizamos durante el entrenamiento. Por tanto, a la hora de crear las particiones, solo se destinarán a TEST aquellas imágenes procedentes de usuarios con más de una valoración con imágenes, concretamente una por usuario. De esta forma garantizamos que en D_test no haya imágenes procedentes de usuarios que no se hayan visto en previamente en TRAIN.

3. Por último, teniendo en cuenta que las críticas se agrupan por ID de usuario y no de restaurante, tenemos que controlar que no haya en la partición de test ningún restaurante que no se encuentre también en la de entrenamiento. Por tanto, extraemos de D_test estas valoraciones y las almacenamos en TRAIN.

Dividiendo los datos de esta forma no perdemos información en el proceso, ya que no descartamos ninguna valoración. Una vez obtenidas las particiones D_test y TRAIN, procedemos a repetir el mismo proceso, pero partiendo del conjunto de TRAIN en lugar del conjunto de todas las valoraciones con imágenes. De esta forma obtenemos los conjuntos D_train, utilizado durante el entrenamiento, y D_val, para monitorizar el proceso de entrenamiento y validar los diferentes modelos en la fase de búsqueda de hiperparámetros, como comentamos en la Sección 4.3.

Se contemplaron y analizaron otras particiones, como por ejemplo, escoger a aquellos usuarios con al menos una crítica positiva y otra negativa, o a los usuarios que tuviesen al menos 3 críticas positivas. Sin embargo, debido a la importancia del tamaño del conjunto de datos en *deep learning*, preferimos utilizar toda la información disponible.

Por último, entre los ejemplos hay imágenes que provienen de usuarios con una sola crítica que resulta ser también la única crítica de un restaurante. Sin embargo esta situación no afecta a la evaluación, ya que siempre son destinadas al conjunto de entrenamiento. De hecho puede

ayudar a mejorar el modelo actuando como regularizador.

5.2.2 Características

Teniendo en cuenta que no se discriminó finalmente ninguna valoración, el conjunto de datos seleccionado presenta los mismos problemas que el conjunto de datos disponible, como podemos ver en las gráficas que conforman la Figura 5.3. En distribución de las muestras para cada partición podemos ver un gran desbalanceo, siendo las críticas positivas la clase dominante. Si nos centramos en las imágenes que incluyen esas críticas, que al fin y al cabo serán las entradas a nuestro sistema, el desbalanceo aumenta ligeramente. Para solucionarlo implementaremos diferentes técnicas, que se explicaran en el Capítulo 4.

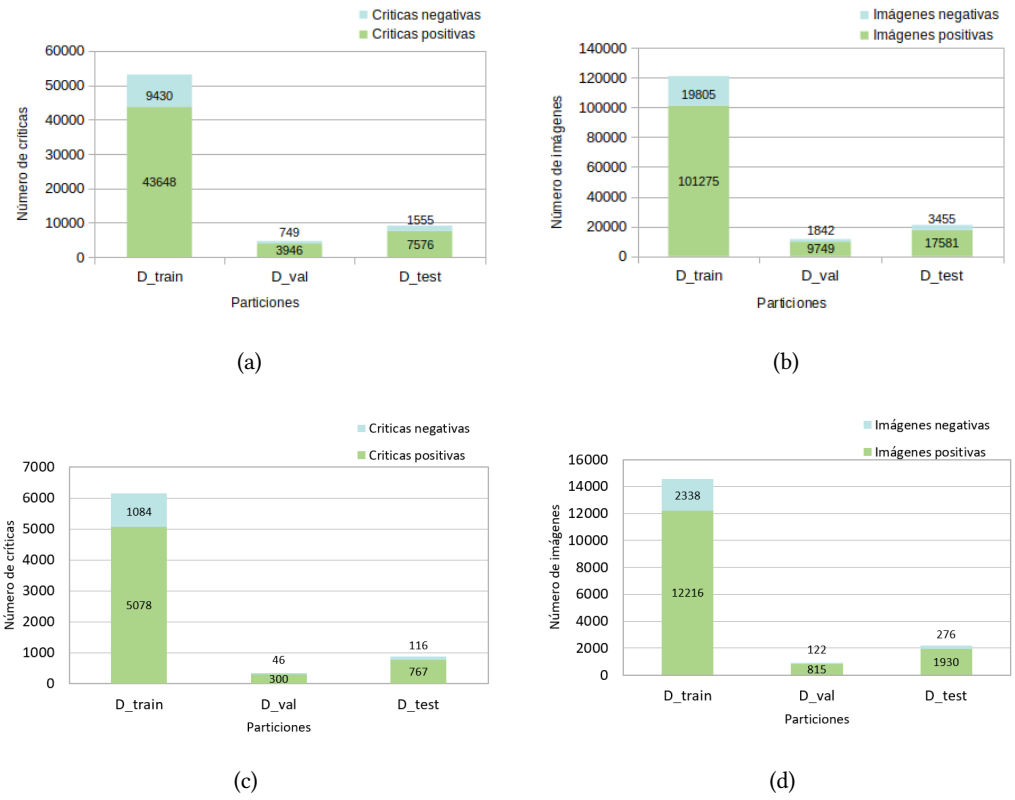


Figura 5.3: Análisis de los conjuntos de datos pertenecientes a las ciudades de Barcelona y Oviedo: (a) número de críticas de Barcelona, (b) número de imágenes de Barcelona, (c) número de críticas de Oviedo, (d) número de imágenes de Oviedo.

Además, si nos fijamos en la distribución de usuarios por número de críticas en las figuras 5.4 (b) y 5.4 (d), podemos ver cómo disponemos de muy poca información para aprender una representación personalizada, resultando en una media de 1,58 críticas/usuario en Barcelona

y 1,3 críticas/usuario en Oviedo. Como se ilustra en la Tabla **??**, en cada valoración hay 2,28 imágenes/crítica, por lo que el número promedio de ejemplos por usuario es de 3,6 en el conjunto de entrenamiento de Barcelona y 2,36 imágenes/crítica en Oviedo, resultando en un promedio de 3 en el conjunto de Oviedo.

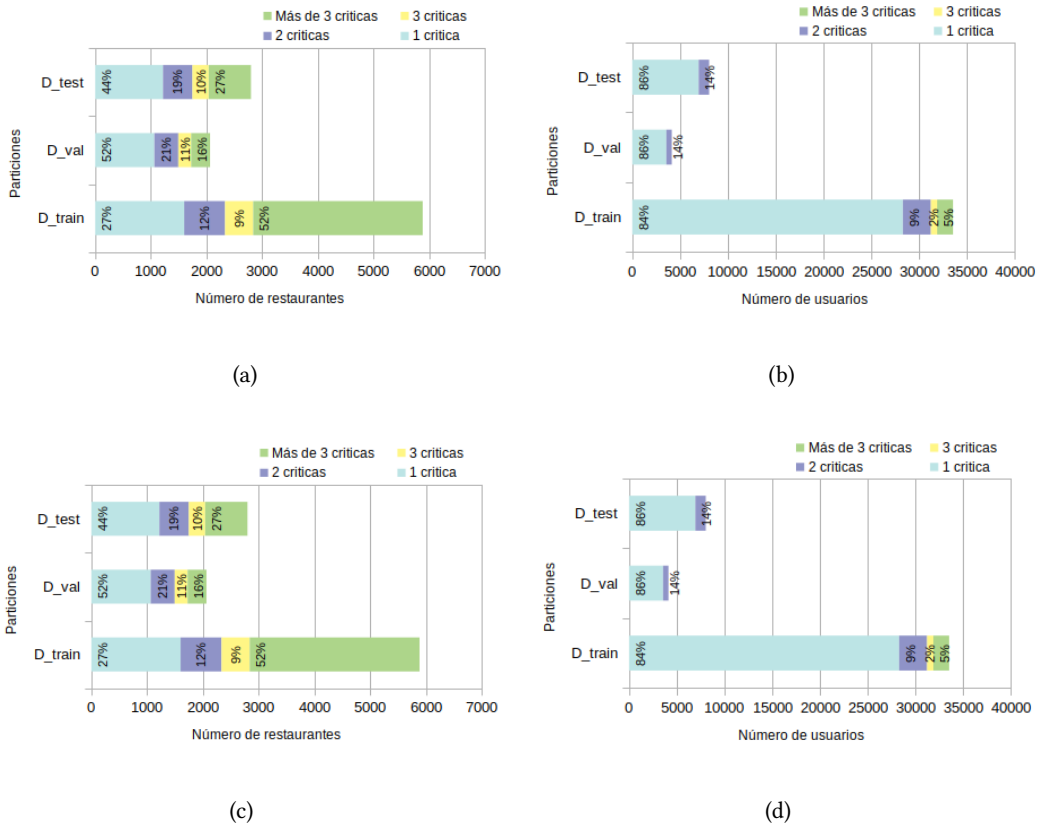


Figura 5.4: Análisis de los conjuntos de datos pertenecientes a las ciudades de Barcelona y Oviedo: (a) distribución de restaurantes en función del número de críticas de Barcelona, (b) distribución de usuarios en función del número de críticas de Barcelona, (c) distribución de restaurantes en función del número de críticas de Oviedo, (d) distribución de usuarios en función del número de críticas de Oviedo.

En el conjunto de datos nos encontramos con imágenes que provienen de usuarios con una sola crítica, que resulta ser también la única crítica de un restaurante, pudiendo generar un efecto de "ruido". Estas imágenes no se extraen del conjunto, ya que en el mundo real esta situación podría darse fácilmente y es necesario que el sistema sea robusto. A causa de la forma de hacer las particiones, en concreto la restricción de no almacenar en el conjunto de test ningún ejemplo procedente de usuarios que no se encuentren también en la partición de entrenamiento, todas las valoraciones que podrían generar ese "ruido" se encuentran en el

conjunto D_train, representando un 0.97% del total. En la Tabla 5.1 se encuentra un resumen más completo sobre la información contenida en las valoraciones.

5.3 Data augmentation

Un aspecto fundamental del Deep Learning es que las características se extraen directamente de los datos, por lo que la cantidad de datos disponibles durante el entrenamiento es fundamental. Existen diferentes técnicas para aumentar de forma artificial las muestras disponibles inicialmente. El *data augmentation* consiste en incrementar el número de ejemplos del conjunto de datos, en nuestro caso el número de imágenes, a partir de los ya disponibles. Esto se consigue realizando pequeños ajustes o modificaciones en la imagen original, como rotaciones, escalados, o traslaciones, entre otras. Estas trasformaciones solo se aplican a las imágenes que se encuentran en la partición de *train*.

Si desplazáramos todos los píxeles de una imagen un píxel a la derecha, para el ojo humano sería una modificación imperceptible y seguiría tratándose de la misma imagen. Por tanto, seguiría transmitiendo exactamente el mismo contenido y no se habría producido ninguna pérdida de información en el proceso. Sin embargo, para una red neuronal es una imagen totalmente diferente, ya que todos los elementos que la conforman, el valor de los píxeles, son diferentes entre sí. Llevando a cabo estas ligeras modificaciones podemos aumentar el número de ejemplos con seguridad por dos motivos: al ser inapreciables para nosotros siguen siendo igual de relevantes para resolver el problema en cuestión y al tratarse de ejemplos totalmente nuevos para el sistema evitamos problemas de sobre-ajuste y mejoramos su capacidad de generalización. No obstante, es importante que la información que estamos introduciendo sea relevante, pensar qué tipo de modificaciones siguen teniendo sentido en relación al contexto que se va a encontrar el sistema. Por ejemplo, si queremos una aplicación que aprenda a distinguir entre caras para gestionar la entrada a un edificio, no tendría ningún sentido aplicar rotaciones de 180º a las imágenes.

Pero el objetivo de esta técnica no es simplemente aumentar la cantidad de datos disponible, si no también la calidad de los mismos. En un escenario real, las imágenes con las que contamos para entrenar han sido obtenidas bajo una serie de condiciones: punto de vista, iluminación, brillo, orientación, tamaño, etc. Una vez entrenado el sistema, éste también va a encontrarse con nuevas imágenes obtenidas bajo diferentes circunstancias. Como no es posible recrear todas estas combinaciones y poder partir de un conjunto de datos lo suficientemente diverso y rico, añadimos los mismos datos con pequeñas alteraciones.

A lo largo de las diferentes aproximaciones se realizaron pruebas con tres datasets dife-

Valoraciones de Barcelona	**D_train**	**D_val**	**D_test**
Críticas positivas	43648	3946	7576
Críticas negativas	9430	749	1555
Número total de críticas	53078	4695	9131
Imágenes de críticas positivas	101275	9749	17581
Imágenes de críticas negativas	19805	1842	3455
Número total de imágenes	121080	11591	21036
Promedio de imágenes por restaurante	20,59	5,62	7,52
Promedio de críticas por restaurante	9,02	2,28	3,26
Restaurantes con 1 crítica	1599	1063	1218
Restaurantes con 2 críticas	733	430	528
Restaurantes con 3 críticas	506	232	293
Restaurantes con más de 3 críticas	3043	337	758
Número total de restaurantes	5881	2062	2797
Promedio de críticas por usuario	1,58	1,14	1,13
Promedio de imágenes por crítica	2,28	2,47	2,3
Usuarios con 1 crítica	28301	3561	6945
Usuarios con 2 críticas	2926	567	1093
Usuarios con 3 críticas	660	0	0
Usuarios con más de 3 críticas	1650	0	0
Número total de usuarios	33537	4128	8038
Valoraciones de Oviedo	**D_train**	**D_val**	**D_test**
Críticas positivas	5078	300	767
Críticas negativas	1084	46	116
Número total de críticas	6162	346	883
Imágenes de críticas positivas	12216	815	1930
Imágenes de críticas negativas	2338	122	276
Número total de imágenes	14554	937	2206
Promedio de imágenes por restaurante	30,64	5,68	9,47
Promedio de críticas por restaurante	12,97	2,1	3,79
Restaurantes con 1 crítica	120	90	93
Restaurantes con 2 críticas	54	37	46
Restaurantes con 3 críticas	39	16	19
Restaurantes con más de 3 críticas	262	22	75
Número total de restaurantes	475	165	233
Promedio de críticas por usuario	1,3	1,08	1,09
Promedio de imágenes por crítica	2,36	2,71	2,5
Usuarios con 1 crítica	4235	296	743
Usuarios con 2 críticas	340	25	70
Usuarios con 3 críticas	70	0	0
Usuarios con más de 3 críticas	108	0	0
Número total de usuarios	4753	321	813

Táboa 5.1: Resumen de la información extraída del análisis de las valoraciones de TripAdvisor de las ciudades de Barcelona y Oviedo.

rentes, obtenidos a raíz del conjunto de datos de inicial:

- C1: conjunto de datos original.
- C2: conjunto de datos en el que para cada imagen de entrada perteneciente a la clase minoritaira se generan 4 imágenes adicionales, para conseguir una relación aproximada de 1:1. Podemos ver un ejemplo en la Figura 5.5.

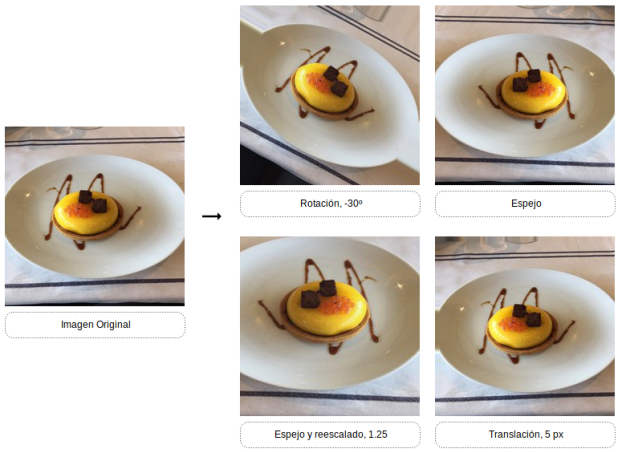


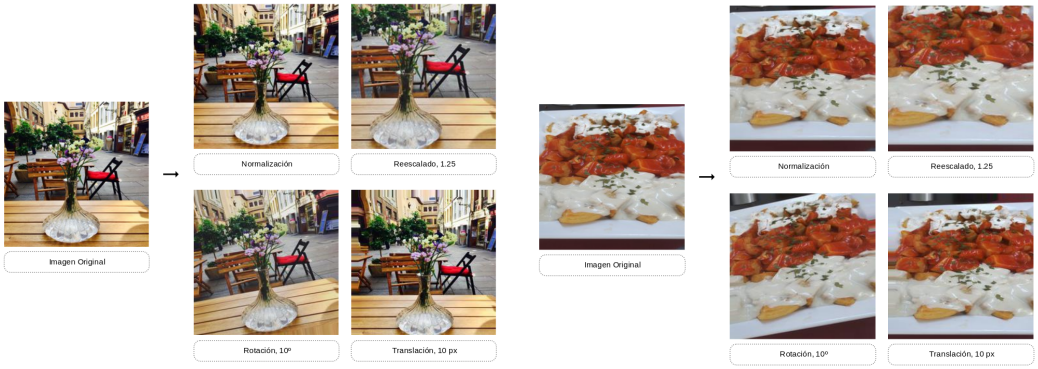
Figura 5.5: *Data augmentation* para C2, con transformaciones únicamente para los ejemplos de la clase minoritaria.

- C3: conjunto de datos en el que, para cada imagen de entrada, se obtienen 3 transformaciones adicionales: reescalado, rotación y traslación, como podemos ver en la Figura 5.6.

En la Tabla 5.2 se muestra el numero final de muestras para cada clase, así como la relación que presentan.

	C1	**C2**	**C3**	**C4**
Ejemplos de la clase positiva	101.275	405.100	405.100	101.275
Ejemplos de la clase negativa	19.805	79.220	158.440	99.025
Relación	5,11:1	5,11:1	2,55:1	1,02:1

Táboa 5.2: Número de ejemplos en cada unos de los conjuntos de datos después de aplicar *data augmentation*.



(a) Local (b) Comida

Figura 5.6: *Data augmentation* para C3, con las mismas transformaciones para los ejemplos de ambas clases.

Capítulo 6

Resultados

A lo largo de este capítulo se presentarán los resultados obtenidos en los diferentes experimentos para cada una de las tres aproximaciones, explicando cada una de las técnicas que fueron utilizadas y los cambios y mejoras que se fueron incorporando en el proceso.

6.1 Primera Aproximación

En la primera aproximación utilizamos la ResNET50 pre-entrenada como extractor de características de todas las imágenes de entrada, almacenándolas posteriormente junto con sus respectivos IDs de usuario y restaurante para cada uno de los tres conjuntos de datos. La arquitectura del modelo puede verse en la Figura 3.1 (a) y tanto ésta como el procedimiento de obtención del vector de características de la imagen están explicados en la Sección 3.1.

El primer experimento que llevamos a cabo fue con el objetivo de establecer los mejores valores para la tasa de aprendizaje del optimizador, la dimensión de los *embeddings* y de la capa completamente conectada, que procesan los identificadores y vector de características respectivamente de los datos de entrada, y el mejor conjunto de datos. Las condiciones de entrenamiento se enumeran a continuación y los resultados se presentan en la Tabla 6.1.

- *Batch size*: 32
- Número de *epochs*: 100
- Optimizador: Adam
- Tasas de Aprendizaje: [0.0001, 0.00001, 0.000001]
- Función de pérdida: entropía binaria cruzada
- Paciencia *early stopping*: 6
- Conjuntos de datos: C1, C2, C3
- Pesos de clases: No aplicado

A pesar de realizar pruebas con diferentes tasas de aprendizaje, concretamente [0,0001, 0.00001, 0.000001], mostramos los resultados obtenidos con el mejor valor de ellos por simplificar. Al utilizar los pesos de una red pre-entrenada para extraer las características y no ajustarlos durante el entrenamiento con nuestros propios datos, probamos con unas tasas de aprendizaje bajas. Evaluamos el rendimiento de los diferentes modelos con la partición de validación, monitorizando el valor de la función de pérdida entre *epochs*. También la utilizaremos para controlar la evolución del entrenamiento y restaurar al modelo a su mejor configuración de pesos con la técnica de *early stopping*, que utilizaremos a lo largo de todos los experimentos para ahorrar tiempo de ejecución.

Aunque la diferencia entre los mejores modelos los mejores valores de dimensión y el mejor conjunto de datos no es significativa, sí al valorar el rendimiento con el resto de conjuntos, siendo 512 el valor con el que se obtienen unos resultados más estables en todos los conjuntos. Reentrenando el mejor modelo con la partición de test y una paciencia fijada en 12 en el *early stopping* obtuvimos un B-score de 0,6524, con una sensibilidad y especifididad de 0,7585 y 0,5987 respectivamente.

Tasa de aprendizaje	0.00001								
Dim. (emb., FC)	128			256			512		
Conjuntos	C1	C2	C3	C1	C2	C3	C1	C2	C3
Sensibilidad	1,0000	0,9247,	0,7296	1,0000	0,9910	0,7327	0,7526	0,9805	0,7148
Especificidad	0,0000	0,0662	0,5613	0,0000	0,0684	0,5928	0,5689	0,1194	0,6162
B-Score	0,0000	0,1236	0,6345	0,0000	0,1280	**0,6554**	0,6480	0,2129	0,6619

Táboa 6.1: Primera aproximación, experimento 1: resultados de clasificación de para los diferentes conjuntos de datos, tasas de aprendizaje y dimensiones de los *embeddings* y capas FC que procesan los datos de entrada al sistema.

Los siguientes experimentos consistieron en utilizar la mejor configuración para el sistema obtenida en el experimento anterior para contrastar su rendimiento utilizando otro optimizador diferente, el RMSprop, y utilizando otro modelo de CNN como extractor de características, InceptionV3. Los modelos fueron evaluados utilizando la partición de test, con una paciencia en el *early stopping* marcada a 12. Como podemos ver en la Tabla 6.2, la mejor red para resolver nuestro problema es ResNET50 y el optimizador que mejores resultados presenta en entrenamiento es Adam. En la Figura 6.1 podemos ver la matriz de confusión obtenida por el mejor modelo, dónde se refleja el número de muestras de las clases *me gusta* y *no me gusta* que son clasificados correctamente.

El último experimento de la aproximación consistió en reentrenar el mejor modelo con

Tasa de aprendizaje	0.00001		
Conjunto	C3		
CNN	ResNET50		InceptionV3
Optimizador	Adam	RSMprop	Adam
Sensibilidad	0,7285	0,8032	0,9488
Especificidad	0,5907	0,2703	0,0692
B-Score	**0,6524**	0,4045	0,1289

Táboa 6.2: Primera aproximación, experimentos 2 y 3: comparación de resultados con diferentes optimizadores, Adam y RMSprop, y de redes, ResNET e InceptionV3.

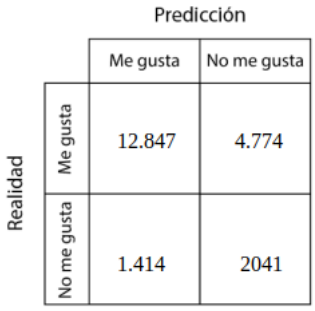


Figura 6.1: Matriz de confusión del mejor modelo de la primera aproximación.

el conjunto de datos de Oviedo para contrastar su comportamiento con muestras representativas más pequeñas durante el entrenamiento, contrastando así su robustez. Los resultados se pueden observar en la Tabla 6.3, presentando un valor de de B-score de 0. Por tanto, si utilizamos un extractor fijo de características que no fue entrenado con nuestros datos, los datos de la partición de Oviedo no son suficientes para que el sistema pueda aprender.

Tasa de aprendizaje	0.0001
Conjunto	C1
Sensibilidad	1,0000
Especificidad	0,0000
B-Score	0,0000

Táboa 6.3: Primera aproximación, experimento 4: resultados obtenidos con el mejor modelo reentrenado con el conjunto de datos de Oviedo.

Por tanto, las conclusiones que pudimos extraer de esta primera aproximación fueron que el mejor valor de *embedding* y primera FC para caracterizar las valoraciones utilizando la ResNET50 como extractor fijo de características es de 512. Así mismo, el conjunto de datos con el que alcanzamos un B-score más elevado es C3, por lo que aplicar en este caso *data augmentation* para solventar el problema de desbalanceo de clases no mejora el resultado,

pero sí aumentar el conjunto de datos inicial con la misma, aumentando en gran medida la especificidad del modelo.

6.2 Segunda Aproximación

En la segunda aproximación el proceso de extracción extracción de características es exactamente el mismo que el utilizado durante la anterior, misma arquitectura del sistema y mismos pesos de entrenamiento en la red convolucional, procedentes del conjunto de datos de ImageNet. La gran diferencia radica en que en esta ocasión los filtros de la red ajustan sus pesos a los ejemplos de los datos de nuestro problema durante el entrenamiento. Por tanto, ahora la CNN forma parte del sistema a entrenar y los datos de entrada consisten en ID de usuario y restaurante e imagen, en lugar de vector de características.

El primer experimento consistió en comprobar si en este caso, ajustando los pesos del extractor de características, aplicar los pesos de clase mejoraban los valores de clasificación del sistema. La red seleccionada fue ResNet50, elegida sobre InceptionV3 por los resultados alcanzados en la anterior aproximación, con una dimensión en *embeddings* y FC inicial de 512, usando Adam como optimizador. La evaluación de los sistemas se llevó a cabo con la partición de test. A continuación se listan los parámetros empleados durante el entrenamiento:

- *Batch size*: 32
- Número de *epochs*: 100
- Optimizador: Adam
- Tasas de Aprendizaje: [0,00001]
- Función de pérdida: entropía binaria cruzada
- Paciencia *early stopping*: 12
- Conjuntos de datos: [C1, C2, C3]

Con los resultados, reflejados en la Tabla 6.4, podemos ver cómo en este caso sí que se consigue un B-score más elevado aplicando pesos a la clase minoritaria. Podemos ver que si utilizamos aumento de datos con pesos de clases la especificidad y sensibilidad disminuyen con respecto al conjunto inicial en el mejor modelo, al comparar los resultados obtenidos con C1 y C3, y sin embargo ocurre lo contrario sin utilizar pesos de clase.

Tasa de aprendizaje	0.00001					
Conjuntos	C1	C2	C3	C1	C2	C3
Peso de la clase *no me gusta*	-	-	-	5.11	1.02	5.11
Sensibilidad	0,9344	0,9364,	0,9278	0,9196	0,9382	0,9315
Especificidad	0,1049	0,2078	0,2391	0,3123	0,2588	0,2946
B-Score	0,1886	0,3401	0,3802	**0,4663**	0,4056	0,4477

Táboa 6.4: Segunda aproximación, experimento 1: comparación de resultados con la mejor tasa de aprendizaje del experimento anterior para los diferentes conjuntos de datos aplicando pesos de clase y sin aplicarlos.

El segundo experimento consistió en comprobar si al cambiar el valor con el que monitorizamos el entrenamiento, que utilizamos para detener su evolución y restaurar los pesos del modelo a los alcanzados con la mejor *epoch*, obteníamos un mejor rendimiento en la clasificación. El valor seleccionado para comparar resultados con el utilizado hasta el momento, el valor de la función de pérdida del conjunto de validación, fue el B-score, y en este caso el conjunto de datos seleccionado en la búsqueda fue el original. En este caso mantuvimos las condiciones de entrenamiento del experimento anterior, pero utilizando pesos de clase en todos los modelos. Por los resultados obtenidos, Tabla 6.5, resulta mucho más efectivo monitorizar el entrenamiento con el valor de B-score, consiguiendo unos valores de especificidad mucho más altos de esta forma.

Conjunto	C1					
Función de monitorización	Función de pérdida			B-score		
Tasa de aprendizaje	0,00001	0,0001	0,001	0,00001	0,0001	0,001
Sensibilidad	0,9191	0,9287	0,8987	0,6268	0,8209	0,5372
Especificidad	0,2448	0,2796	0,3192	0,3664	0,4631	0,6189
B-Score	0,3867	0,4298	0,4711	0,4625	**0,5921**	0,5752

Táboa 6.5: Segunda aproximación, experimento 2: comparación de resultados monitorizando el entrenamiento con valores diferentes: valor de función de pérdida o valor de B-score .

En tercer lugar realizamos una búsqueda de hiperparámetros para encontrar, una vez fijada la mejor forma de monitorizar el entrenamiento y aplicando pesos a la clase minoritaria a raíz de los resultados obtenidos en los experimentos anteriores, la mejor configuración de modelo en cuanto a tasas de aprendizaje y conjuntos de datos. Al igual que en la primera aproximación, en este experimento utilizamos la partición de validación monitorizando el valor de B-score con una paciencia de 6 en el *early stopping* y reentrenamos posteriormente al

mejor modelo para evaluarlo con la partición de test, cambiando por una paciencia de 12. El resto de condiciones y parámetros se mantienen con respecto al anterior experimento. En la Tabla 6.6 podemos ver los resultados.

Tasas de aprendizaje	0.001			0.0001			0.00001			0.000001		
Conjuntos	C1	C2	C3	C1	C2	C3	C1	C2	C3	C1	C2	C3
Sensibilidad	0,5372	0,8862	0,8740	0,8209	0,93333	0,8643	0,5629	0,9690	0,8473	0,6268	0,1800	0,0007
Especificidad	0,6189	0,3990	0,3610	0,4631	0,2671	0,4001	0,4691	0,2041	0,4039	0,3664	0,8154	0,9995
B-Score	0,5752	0,5503	0,5110	**0,5921**	0,4153	0,5470	0,5117	0,3372	0,5470	0,4625	0,2949	0,0014

Táboa 6.6: Segunda aproximación, experimento 3: resultados de clasificación de para los diferentes conjuntos de datos y tasas de aprendizaje.

Igual que en la anterior aproximación, contrastamos el rendimiento de las dos redes convolucionales, comparando los resultados obtenidos utilizando InceptionV3 en comparación a los alcanzados hasta el momento con el sistema con ResNET50. Los modelos fueron evaluados en este caso con la partición de test, con una paciencia de 12 en el *early stoping*, y de nuevo obtenemos unos resultados de clasificación mucho más elevados con ResNET, como podemos ver en la Tabla 6.7. La matriz de confusión para este modelo se refleja en la Figura 6.2

Tasa de aprendizaje	0.0001	
Conjunto	C1	
CNN	ResNET50	InceptionV3
Sensibilidad	0,8412	0,2800
Especificidad	0,4654	0,7520
B-Score	**0,5993**	0,4081

Táboa 6.7: Segunda aproximación, experimento 4: comparación de resultados utilizando diferentes redes convolucionales, ResNET e InceptionV3.

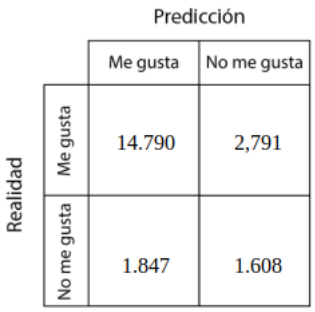


Figura 6.2: Matriz de confusión del mejor modelo de la segunda aproximación.

El último experimento consistió en reentrenar el mejor modelo obtenido hasta el momento bajo las mismas condiciones de entrenamiento con los datos de la ciudad de Oviedo, evaluando su rendimiento con la partición de test, cuyos resultados se muestran en la Tabla 6.8. En este caso, adaptando los pesos del extractor de características con nuestros datos, el modelo sí es capaz de aprender. Al disponer de menos datos de entrada para realizar el entrenamiento el valor de B-score disminuye, así como la clasificación de ejemplos de la clase *me gusta*, pero aumenta la de los ejemplos de la clase *no me gusta*.

Tasa de aprendizaje	0.0001
Conjunto	C1
Sensibilidad	0,5813
Especificidad	0,5108
B-Score	0,5438

Táboa 6.8: Segunda aproximación, experimento 5: resultados obtenidos con el mejor modelo reentrenado con el conjunto de datos de Oviedo.

En general, comparando los resultados de esta aproximación con la anterior, vemos como obtenemos mejores resultados utilizando el vector de características de la red pre-entrenada ajustando posteriormente los pesos del sistema al realizar la clasificación, aumentando los ejemplos de entrada con aumento de datos para reforzar el aprendizaje. Si ajustamos los filtros que intervienen en la extracción de la red con nuestros ejemplos durante el entrenamiento, como hacemos en esta aproximación, los resultados finales de clasificación empeoran y, en consecuencia, la extracción de características que la propició. Sobre todo si nos fijamos en los valores de especificidad, relacionada con la clasificación de los ejemplos de la clase minoritaria.

6.3 Tercera Aproximación

Al igual que en la primera aproximación, en este caso utilizaremos un extractor de características, con la gran diferencia de que en este caso sí que fue entrenado a partir de nuestros datos. Por tanto, aunque no se adapte a los ejemplos de entrenamiento a partir de sus etiquetas como en la segunda aproximación, todas las características que el extractor aprendió a reconocer forman parte de nuestro contexto.

El primer paso que tuvimos que llevar a cabo fue entrenar un *autoencoder* convolucional, cuya arquitectura se ilustra en la Sección 3.13. Para llevar a cabo el entrenamiento fue utilizado el conjunto de datos original, C1. Tanto los datos de entrada como las etiquetas consistieron en las imágenes de la partición de entrenamiento, la partición de validación para

monitorizar el valor de B-score entre *epochs* y la de test para validar el rendimiento final. Los hiperparámetros fijados para el entrenamiento fueron los siguientes:

- *Batch size*: 32
- Número de *epochs*: 100
- Optimizador: Adam
- Tasa de Aprendizaje: 0.001
- Función de pérdida: entropía binaria cruzada
- Paciencia *early stopping*: 6
- Conjuntos de datos: C1

En la Figura 6.3 podemos ver un ejemplo de salida del *autoencoder* en las imágenes de la segunda fila, recibiendo como entrada las imágenes pertenecientes a la partición de test de C1. El resultado de evaluar la función de pérdida con el conjunto de test fue de 0.5199. Una vez entrenado el autoencoder, se obtuvieron los nuevos conjuntos C1, C2 y C3 siguiendo el procedimiento ilustrado en la Figura 3.2 (b) y explicado con anterioridad.

Figura 6.3: Ejemplo de salida del *autoencoder* (segunda fila) proporcionando como entrada imágenes de la partición de test (primera fila) del conjunto C1.

El primer experimento que llevamos a cabo consistió en una búsqueda de hiperparámetros para encontrar el valor de tasa de aprendizaje y conjunto con los que mejores resultados obtiene el sistema. El objetivo es el mismo que en anteriores aproximaciones, reducir el número de pruebas necesarias en futuros experimentos y obtener el modelo con el rendimiento más alto posible. La arquitectura del sistema es la inicial, eligiendo en un primer momento un tamaño de salida de las capas de *embedding* y primera FC de 512, por ser el que mejor rendimiento proporcionó en anteriores aproximaciones. Al tratarse de una primera búsqueda, la evaluación fue realizada con la partición de validación y las características que marcaron el entrenamiento fueron las siguientes:

- *Batch size*: 32
- Número de *epochs*: 100
- Optimizador: Adam
- Tasas de Aprendizaje: [0.001, 0.0001, 0.00001, 0.000001]
- Función de pérdida: entropía binaria cruzada
- Paciencia *early stopping*: 6
- Conjuntos de datos: C1, C2, C3
- Pesos de clases: No aplicado

Los resultados se presentan en la Tabla 6.9. En este caso el conjunto de datos con el que mejor B-score alcanza el modelo es C2, en el que se utiliza *data augmentation* para balancear los ejemplos entre clases. No se aprecian diferencias significativas si lo comparamos con el valor de B-score obtenido con el conjunto original (C1) para la misma tasa de aprendizaje, pero igualando la relación de clases con esta técnica aumentamos el valor de especificidad, al mismo tiempo que disminuimos el de sensibilidad.

Tasas de aprendizaje	0.001			0.0001			0.00001			0.000001		
Conjuntos	C1	C2	C3	C1	C2	C3	C1	C2	C3	C1	C2	C3
Sensibilidad	0.7051	0.8400	0.6684	0.7386	0.6280	0.7882	1.0000	0.0752	0.6706	0.9871	0.4301	1.0000
Especificidad	0.5673	0.4696	0.5505	0.6303	0.7671	0.5559	0.0000	0.9387	0.6574	0.0130	0.5809	0.0000
B-Score	0.6287	0.6024	0.6037	0.6802	**0.6906**	0.6520	0.0000	0.1392	0.6640	0.0257	0.4942	0.0000

Táboa 6.9: Tercera aproximación, experimento 1: resultados de clasificación de para los diferentes conjuntos de datos y tasas de aprendizaje.

De hecho, si con las mismas condiciones de entrenamiento probamos la tasa de aprendizaje seleccionada para los tres conjuntos de datos y aplicamos los pesos de clases siguiendo la Fórmula 4.2, vemos en la Tabla 6.10 cómo los resultados en el mejor modelo se mantienen al presentar una relación 1,02:1, implicando un peso aplicado a la clase minoritaria de 1,02, pero con el resto de conjuntos los resultados empeoran. Si validamos el B-score del modelo con y sin pesos para el mejor conjunto de datos con el conjunto de test, sigue obteniendo un mayor rendimiento el modelo que no los usa, como podemos ver en la Tabla 6.11 . Por tanto, en un problema desbalanceado como el nuestro, con las particularidades descritas en la Sección 2.3, es preferible aplicar *data augmentation* y no seguir la estrategia de penalizar la pérdida obtenida con los ejemplos de la case minoritaria durante el entrenamiento. La aplicación conjunta de técnicas, como podemos ver con los resultados obtenidos con C3 aplicando pesos de clase, tampoco es recomendable.

Tasas de aprendizaje	0.0001			0.0001		
Conjuntos	C1	C2	C3	C1	C2	C3
Peso de la clase *no me gusta*	-	-	-	5.11	1.02	5.11
Sensibilidad	0.7386	0.6274	0.7882	0.8889	0.6280	0.8041
Especificidad	0.6303	0.7861	0.5559	0.3469	0.7671	0.5043
B-Score	0.6802	**0.6979**	0.6520	0.4991	0.6906	0.6199

Táboa 6.10: Tercera aproximación, experimento 2: comparación de resultados con la mejor tasa de aprendizaje del experimento anterior para los diferentes conjuntos de datos aplicando pesos de clase y sin aplicarlos.

Tasa de aprendizaje	0.0001	
Conjunto	C2	
Optimizador	Adam	RSMprop
Sensibilidad	0.6906	0.5423
Especificidad	0.7716	0.5213
B-Score	**0.7289**	0.5316

Táboa 6.11: Tercera aproximación, experimento 3: comparación de resultados aplicando pesos de clase evaluando con la partición de test.

Si volvemos a los resultados del primer experimento, en la Tabla 6.9, y comparamos el valor de B-score obtenido con el conjunto C1 en comparación con el obtenido con C3, vemos cómo si seguimos aumentando los datos pero mantenemos las características de conjunto inicial, empeoramos el problema en relación a la clasificación de ejemplos de la clase *no me gusta* en lugar de ayudar a solucionarlo. En relación a este aspecto, la especificidad, sí que se observa mejoría en el rendimiento del sistema, con nuestros datos en particular, utilizando *data augmentation* para solventar el desbalanceo de clases, pasando de un 63% de ejemplos de la clase *no me gusta* con el conjunto original a un 77% siguiendo esta estrategia.

El siguiente experimento consistió en una nueva búsqueda de hiperparámetros, en este caso para establecer el mejor valor para las dimensiones de salida de las capas de *embedding* y de la capa completamente conectada que procesan las entradas del sistema, ID de usuario y restaurante y vector de características de la imagen respectivamente. Para ello elegimos los hiperparámetros seleccionados con los anteriores experimentos y replicamos las mismas condiciones de entrenamiento, eligiendo los valores [128, 256, 512] para los *embeddings* y primera FC, pero en este caso el rendimiento del sistema fue evaluado con la partición de test. Los resultados se muestran en la Tabla 6.12, manifestando una clara mejoría con el valor 512 al igual que en el resto de aproximaciones con la CNN.

Tasa de aprendizaje	0.0001		
Conjunto	C2		
Dimensión de *embedding*	128	256	512
Sensibilidad	0,4933	0,4300	0,6906
Especificidad	0,5560	0,6310	0,7716
B-Score	0,5228	0,5114	**0,7289**

Táboa 6.12: Tercera aproximación, experimento 3: comparación de resultados con la mejor tasa de aprendizaje y conjunto de datos del experimento anterior para diferentes dimensiones de *embedding*.

A continuación, probamos el funcionamiento del optimizador RMSprop en comparación con Adam con el mejor modelo anterior, para lo que replicamos las mismas condiciones de entrenamiento e hiperparámetros, pero cambiando optimizador. En la Tabla 6.13 podemos ver cómo si comparamos con los resultados del experimento anteriores, seguimos obteniendo mejores resultados usando Adam con esta técnica, al igual que en anteriores comparaciones con CNNs, sobre todo en términos de especificidad, como se refleja en los valores de la matriz de confusión del modelo de la Figura 6.4.

Tasa de aprendizaje	0.0001	
Conjunto	C2	
Optimizador	Adam	RSMprop
Sensibilidad	0.6906	0.5423
Especificidad	0.7716	0.5213
B-Score	**0.7289**	0.5316

Táboa 6.13: Tercera aproximación, experimento 4: comparación de optimizadores, Adam vs RMSprop.

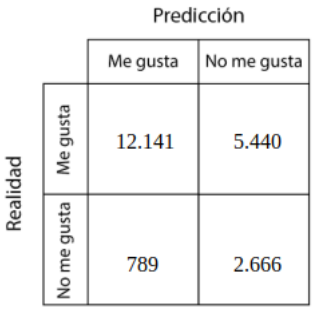


Figura 6.4: Matriz de confusión del mejor modelo de la tercera aproximación.

Al igual que en las dos anteriores aproximaciones, el último experimento consistió en

reentrenar el mejor modelo obtenido hasta el momento con los datos de la ciudad de Oviedo, cuyos resultados se muestran en la Tabla 6.14. Con este experimento podemos ver como el rendimiento del extractor de características depende en gran medida de los datos con los que llevó a cabo el entrenamiento. En este caso los resultados también empeoran, principalmente el valor de *especificidad*, probablemente porque el sistema no dispone de los datos suficientes para poder llevar a cabo una correcta predicción.

Tasa de aprendizaje	0,0001
Conjunto	C1
Sensibilidad	0,6176
Especificidad	0,3956
B-Score	0,4845

Táboa 6.14: Tercera aproximación, experimento 5: resultados obtenidos con el mejor modelo reentrenado con el conjunto de datos de Oviedo.

Capítulo 7

Conclusiones

Como comentábamos en el primer capítulo, a pesar del gran éxito que la recomendación personalizada está mostrando en algunas plataformas digitales, como Netflix o Youtube, no existe en el mercado una aplicación que realice recomendaciones personalizadas de restaurantes. Si recordamos el alto flujo de visitantes mensuales de TripAdvisor y el impacto que las redes sociales y las recomendaciones ejercen en la intención de compra de los consumidores, nos hacemos a la idea del impacto que un sistema recomendador personalizado, utilizando la información contenida en este portal, podría tener en términos económicos.

El objetivo principal de este trabajo era contrastar el uso de diferentes técnicas de extracción de características aplicadas a la recomendación de restaurantes, utilizando para ello los datos extraídos de las valoraciones realizadas pos usuarios de TripAdvisor. Modelamos un sistema de clasificación, con la premisa de que si predice correctamente las valoraciones ante nuevos ejemplos, está extrayendo correctamente de las imágenes las características necesarias que motivan esa decisión en el usuario.

Se llevaron a cabo tres aproximaciones para probar diferentes técnicas, concretamente evaluamos el rendimiento como extractores de características de las redes convolucionales, primera y segunda aproximación, frente a los *autoencoders*, tercera aproximación. También analizamos el uso de *data augmentation*, tanto para tratar de reducir los problemas de desbalanceo de clases que caracteriza nuestro problema, conjunto C2, como para mejorar el aprendizaje en general, conjunto C3. Realizamos diferentes experimentos con la intención de contrastar la eficacia del *transfer learning*, utilizando los pesos de una CNN pre-entrenada con otro conjunto de datos en un contexto diferente al nuestro, así como del *fine-tunning* de parámetros, utilizando los pesos de la CNN pre-entrenda sin modificar para obtener el vector de características durante la primera aproximación o reajustándolos al integrar la CNN durante el entrenamiento del sistema en la segunda. Para evaluar los diferentes modelos propusimos

una nueva métrica, B-score, utilizada para promediar la especificidad y sensibilidad alcanzadas por el sistema. Está especialmente recomendada en nuestro caso, ya que al tratarse de un sistema de clasificación binaria y ser la clase positiva la mayoritaria con diferencia, penaliza aquellos casos en los que la clasificación de ejemplos de la clase *no me gusta* es muy baja a pesar de que la de los ejemplos de la clase *me gusta* sea elevada. El uso de otras métricas más populares, al no tener en cuenta la especificidad, nos impiden seleccionar al modelo que mayor rendimiento proporciona realmente.

Si observamos los resultados obtenidos con los mejores modelos de las tres aproximaciones, comprobamos como utilizar un extractor fijo de características para obtener el vector proporciona un mejor rendimiento con nuestros datos de entrada. El modelo que obtiene un mejor B-score es el que utiliza un *autoencoder* (0,7289), tercera aproximación, seguido por el modelo de la primera aproximación (0,6524), que utiliza los pesos de la CNN pre-entrenada sin modificar durante el entrenamiento. Además, utilizar un extractor fijo pre-entrenado con nuestros datos también funciona mejor que otro con pesos obtenidos por otra vía, al realizar una mejor clasificación utilizando el vector de características obtenido a través del *autoencoder* de la primera aproximación entrenado con nuestras imágenes frente a la CNN con los pesos de ImageNet de la CNN de la segunda, una base de datos mucho más grande que la nuestra. Por tanto, podemos concluir que el contexto es muy importante a la hora de aprender a detectar este tipo de características.

Comparando el rendimiento de las dos primeras aproximaciones, resulta curioso ver como ajustando los filtros detectores de características de la red con los nuestros datos de entrada durante el entrenamiento se consiguen peores resultados. Lo mismo ocurre cuando el sistema recibe una peor evaluación usando *data augmentation* en lugar del conjunto de datos original, cuando en problemas de *deep learning* proceder al aumento de ejemplos de entrenamiento suele mejorar los resultados.

A causa de estos dos motivos, unidos a los problemas de validación que se pueden presentar por la propia naturaleza de los datos explicados en la Sección 2.3, es posible que la extracción de características no dependa finalmente del usuario, aunque sí lo haga la recomendación posterior. Las características son intrínsecas al restaurante, no al usuario. Es decir, existen imágenes del mismo restaurante con las mismas características que presentan valoraciones contrarias para usuarios diferentes. Al disponer de tan pocos datos para cada usuario y presentarse un desbalanceo de clases tan altamente marcado, la separación interclase podría no estar suficientemente definida con nuestros datos de entrada.

Por tanto, como trabajo futuro, se propone reentrenar los modelos que obtuvieron el valor de B-score más elevado en las diferentes aproximaciones, y por tanto, mayor sensibilidad y es-

pecificidad, pero agrupando las valoraciones en función del restaurante en lugar del usuario, etiquetándolas con su puntuación general. El objetivo es comprobar si el sistema aprende a detectar mejor las características presentes en las imágenes, que representan los gustos de los usuarios, utilizando un extractor que aprendió a detectarlas buscando patrones comunes a los gustos del público en general. De esta forma, al utilizar la valoración general del restaurante como etiqueta en lugar de la que otorga cada usuario, es más fácil para el sistema aprender a clasificar los ejemplos en función de los gustos de la mayoría. Tras obtener el vector de características generado por un extractor de características que ha sido entrenado con nuestros datos organizados y etiquetados de esta manera, pudiendo utilizar después una medida de similitud que compare los ejemplos pertenecientes al usuario con los nuevos restaurantes propuestos para tomar una decisión. La idea es aprender a detectar características relevantes utilizando el gusto general, para luego aplicarlo de forma personalizada a cada usuario en particular. Así mismo, cuando la extracción de características esté lista, el siguiente paso será integrarla en un sistema de recomendación.

Apéndices

Apéndice A

Glosario de acrónimos

SO *Sistema Operativo.*

GiB *Gibibyte.*

CNN *Convolutional neural network.*

ILSVRC *ImageNet Large Scale Visual Recognition Challenge.*

Conv. *Gibibyte.*

FC *Fully Connected Layer.*

ReLU *Rectified Linear Unit.*

Apéndice B

Glosario de termos

Epoch Ciclo de entrenamiento con todos los ejemplos del conjunto de entrenamiento.

Batch Hiperparámetro que controla el número de ejemplos de entrenamiento tras los cuáles los pesos de la red son actualizados.

Bibliografía

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[2] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting," *The journal of machine learning research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[3] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[4] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[5] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.

[6] P. Resnick and H. R. Varian, "Recommender systems," *Communications of the ACM*, vol. 40, no. 3, pp. 56–59, 1997.

[7] D. Jannach, M. Zanker, A. Felfernig, and G. Friedrich, *Recommender systems: an introduction*. Cambridge University Press, 2010.

[8] R. He and J. McAuley, "Vbpr: visual bayesian personalized ranking from implicit feedback," in *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[9] S. Wang, Y. Wang, J. Tang, K. Shu, S. Ranganath, and H. Liu, "What your images reveal: Exploiting visual contents for point-of-interest recommendation," in *Proceedings of*

the 26th International Conference on World Wide Web. International World Wide Web Conferences Steering Committee, 2017, pp. 391–400.

[10] W.-T. Chu and Y.-L. Tsai, "A hybrid recommendation system considering visual information for predicting favorite restaurants," *World Wide Web*, vol. 20, no. 6, pp. 1313–1331, 2017.

[11] L. Yang, C.-K. Hsieh, H. Yang, J. P. Pollak, N. Dell, S. Belongie, C. Cole, and D. Estrin, "Yum-me: a personalized nutrient-based meal recommender system," *ACM Transactions on Information Systems (TOIS)*, vol. 36, no. 1, p. 7, 2017.

[12] H.-y. Zhang, P. Ji, J.-q. Wang, and X.-h. Chen, "A novel decision support model for satisfactory restaurants utilizing social information: A case study of tripadvisor. com," *Tourism Management*, vol. 59, pp. 281–297, 2017.

[13] C. Zhang, H. Zhang, and J. Wang, "Personalized restaurant recommendation method combining group correlations and customer preferences," *Information Sciences*, vol. 454, pp. 128–143, 2018.

[14] G. Amis. (2017) Improving tripadvisor photo selection with deep learning. [Online]. Available: https://www.tripadvisor.com/engineering/improving-tripadvisor-photo-selection-deep-learning/

[15] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Advances in neural information processing systems*, 2015, pp. 91–99.

[16] C. Farabet, C. Couprie, L. Najman, and Y. LeCun, "Learning hierarchical features for scene labeling," *IEEE transactions on pattern analysis and machine intelligence*, vol. 35, no. 8, pp. 1915–1929, 2012.

[17] D. H. Hubel and T. N. Wiesel, "Receptive fields, binocular interaction and functional architecture in the cat's visual cortex," *The Journal of physiology*, vol. 160, no. 1, pp. 106–154, 1962.

[18] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. MIT press, 2016.

[19] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[20] C. Guo and F. Berkhahn, "Entity embeddings of categorical variables. arxiv 2016," *arXiv preprint arXiv:1604.06737*.

[21] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," California Univ San Diego La Jolla Inst for Cognitive Science, Tech. Rep., 1985.

[22] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.

[23] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, 1989.

[24] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1026–1034.

[25] L. Theis, W. Shi, A. Cunningham, and F. Huszár, "Lossy image compression with compressive autoencoders," *arXiv preprint arXiv:1703.00395*, 2017.

[26] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *science*, vol. 313, no. 5786, pp. 504–507, 2006.

[27] C. R. A. Chaitanya, A. S. Kaplanyan, C. Schied, M. Salvi, A. Lefohn, D. Nowrouzezahrai, and T. Aila, "Interactive reconstruction of monte carlo image sequences using a recurrent denoising autoencoder," *ACM Transactions on Graphics (TOG)*, vol. 36, no. 4, p. 98, 2017.

[28] J. Xie, L. Xu, and E. Chen, "Image denoising and inpainting with deep neural networks," in *Advances in neural information processing systems*, 2012, pp. 341–349.

[29] C. Hong, J. Yu, J. Wan, D. Tao, and M. Wang, "Multimodal deep autoencoder for human pose recovery," *IEEE Transactions on Image Processing*, vol. 24, no. 12, pp. 5659–5670, 2015.

[30] J. Masci, U. Meier, D. Cireşan, and J. Schmidhuber, "Stacked convolutional auto-encoders for hierarchical feature extraction," in *International Conference on Artificial Neural Networks*. Springer, 2011, pp. 52–59.

[31] F. Chollet. (2016) Building autoencoders in keras. [Online]. Available: https://blog.keras.io/building-autoencoders-in-keras.html

[32] P.-T. De Boer, D. P. Kroese, S. Mannor, and R. Y. Rubinstein, "A tutorial on the cross-entropy method," *Annals of operations research*, vol. 134, no. 1, pp. 19–67, 2005.

[33] J. Kiefer, J. Wolfowitz *et al.*, "Stochastic estimation of the maximum of a regression function," *The Annals of Mathematical Statistics*, vol. 23, no. 3, pp. 462–466, 1952.

[34] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *International conference on machine learning*, 2013, pp. 1139–1147.

[35] G. Hinton, N. Srivastava, and K. Swersky, "Overview of mini-batch gradient descent," *Neural Networks for Machine Learning*, vol. 575, 2012.

[36] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[37] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?" in *Advances in neural information processing systems*, 2014, pp. 3320–3328.

[38] J. S. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *Advances in neural information processing systems*, 2011, pp. 2546–2554.