UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf
Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Reconocimiento de emociones mediante técnicas de aprendizaje profundo

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Luis Cebrián Chuliá

Tutor: Roberto Paredes Palacios

2015-2016

Resumen

Con la realización de este proyecto se quiere profundizar en el uso de técnicas de aprendizaje automático para el problema de reconocimiento de emociones a partir de rostros. La técnica que emplearemos para abordar el problema es: redes neuronales convolucionales.

Durante el transcurso de este proyecto describiremos en qué consiste esta técnica y cómo funciona. Describiremos cómo la aplicamos para el problema que tenemos entre manos y las herramientas utilizadas para ello. También describiremos todo el proceso que conlleva la resolución de un problema de este tipo: preparación de los datos y arquitectura de la red neuronal, y todo el proceso de aprendizaje y evaluación de resultados. Por último presentaremos los resultados obtenidos en términos de precisión del modelo y también las conclusiones a las que hemos llegado una vez finalizadas las pruebas.

Para la resolución de este problema emplearemos Python como lenguaje de programación, Theano como librería de aprendizaje automático y Blocks como *framework* para trabajar de forma cómoda con Theano.

Este proyecto tiene un doble objetivo. Por un lado, aprender a usar esta clase de técnicas que, pese a tener varias décadas de edad, son posibles hoy en día gracias al rápido avance de la infraestructura hardware. Por otro lado, comparar nuestros resultados con los obtenidos por otros investigadores en la resolución de un mismo problema. Este segundo objetivo es posible ya que emplearemos como conjunto de datos de entrenamiento los proveídos por la página web Kaggle, en una competición celebrada en 2013.

Palabras clave: red neuronal convolucional, reconocimiento de emociones, Theano, aprendizaje profundo, aprendizaje automático

Abstract

Through this project we want to deepen our knowledge about machine learning techniques by finding a solution for recognizing emotions from facial expressions. The technique we will use in order to achieve this is called convolutional neural networks.

During the course of this project we will describe what this technique is about, how it works, how we apply it to the problem we are dealing with and the tools we use to support our solution. We will also explain the process of solving our problem in terms of setting up the data, architecture and learning process of the neural network and giving an evaluation of the found results. Lastly, we will present the results in terms of the model's accuracy and we will give a conclusion based on the collected test results.

In order to solve this problem we will use Python as our coding language, Theano as our machine learning library and Blocks as a framework so we can work comfortably with Theano.

This project has a double objective, the first one being that we learn to use machine learning techniques that, despite being several decades old, they are possible nowadays because of the advancement of hardware. The second objective is to rank our results with the ones obtained by other researchers by solving the same problem. Completing this last objective is possible because we will be using a dataset used in a competition in facial emotion recognition held by the website *Kaggle* in 2013.

Keywords: convolutional neural network, emotion recognition, Theano, deep learning, machine learning.

Resum

Mitjançant la realització d'aquest projecte volem aprofundir en l'ús de tècniques d'aprenentatge automàtic aplicat al problema de reconeixement d'emocions a partir de rostres. La tècnica que utilitzem per a resoldre el problema es: xarxes neuronals convolucionals.

Durant el transcurs d'aquest projecte descriurem en que consisteix aquesta tècnica i com funciona. Descriurem com l'hem aplicat al problema que tenim davant i les ferramentes utilitzades per a fer-ho. També explicarem tot el procés que comporta la resolució d'un problema d'aquest tipus: preparar les dades i l'arquitectura de la xarxa neuronal i, tot el procés d'aprenentatge i avaluació de resultats. Finalment, presentarem els resultats obtinguts en termes de precisió del model i també les conclusions a què hem arribat una vegada finalitzades les probes.

Per a la resolució del projecte utilitzarem Python com a llenguatge de programació, Theano com a llibreria d' aprenentatge automàtic i Blocks com a *framework* per a treballar de manera còmoda amb Theano.

Aquest projecte té com a doble objectiu, per un costat, aprendre a usar aquest tipus de tècniques que, tot i que tenen diverses dècades de antiguitat, són possibles avui en dia gràcies als avanços del *hardware.* Per un altre costat, comparar els nostres resultats amb els obtinguts per investigadors en la resolució del mateix problema. Aquest segon objectiu és possible ja que utilitzarem com a conjunt de dades d'entrenament els proveïts per la pàgina web Kaggle, en una competició celebrada a 2013.

Paraules clau: xarxa neuronal convolucional, reconeixement d'emocions, Theano, aprenentatge profund, aprenentatge automàtic

Agradecimientos

Querría agradecer a las personas que me han ayudado directa o indirectamente en la realización de este proyecto. Este trabajo no habría sido posible sin ellos.

A Alberto Albiol por la gran ayuda prestada en la puesta en marcha e implementación del proyecto así como por haberme prestado la máquina empleada para realizar el proyecto. Sin su ayuda todo habría sido mucho más difícil.

A Roberto Paredes por ofrecerse a ser tutor de este proyecto, aconsejarme y motivarme.

A mis padres por estar siempre apoyándome y hacer que todo esto sea posible.

A mis hermanos. Gracias por servirme de motivación.

A Miriam. Gracias por no fallarme nunca.

Luis Cebrián Chuliá

Valencia, 3 de Julio de 2016

Índice de contenidos

Índice de figuras

Índice de tablas

Abreviaturas y acrónimos

API	*Application programming interface*
AU	*Action Unit*
CIFAR-10	*Canadian Institute for Advanced Research*
CL	*Convolucional Layer* / Capa convolucional
CNN	*Convolutional Neural Network* / Red neuronal convolucional
FACS	*Facial Action Coding System*
GPU	*Graphics Processor Unit* / Unidad de proceso gráfico
HMM	*Hidden Markov Model* / Modelo Oculto de Markov
MIT	*Massachusetts Institute of Technology*
MNIST	*Mixed National Institute of Standards and Technology*
MP	*Max Pooling layer* / Capa max pooling
NB	*Naïve Bayes*
ReLU	*Rectified linear unit*
RGB	*Red Green Blue*
SGD	*Stochastic Gradient Descent* / Descenso por gradiente estocástico
SSS	*Stochastic Structure Search*
SVM	*Support Vector Machine* / Máquina vector de soporte
TAN	*Tree Augmented Naïve Bayes*
UPV	Universidad Politécnica de Valencia

CAPÍTULO 1

Introducción

El objetivo de este trabajo es resolver el problema de reconocimiento de emociones mediante técnicas de aprendizaje profundo (conocido como *Deep Learning).* Emplearemos como técnica de aprendizaje automático las redes neuronales convolucionales. Con ellas trataremos de clasificar emociones (felicidad, enfado, asco, etc.) a partir de imágenes de rostros.

Sin embargo, conviene primero explicar el concepto de aprendizaje automático y el término *deep learning*. Por *aprendizaje,* por parte de una máquina, nos referimos a la tarea que busca encontrar una función que dada una información de entrada (en nuestro caso son imágenes de rostros, pero puede trasladarse también a una señal sonora o cualquier otro tipo de información extraída de un dominio específico), obtengamos la información de salida relevante para nuestro problema (en nuestro caso, la correspondiente emoción que presenta la imagen dada como entrada). Estas tareas buscan aprender una serie de transformaciones no lineales de la información de entrada. Esta serie de transformaciones están divididas en capas, y puesto que el número de capas presentes puede ser potencialmente grande, se le da el nombre de *deep learning*, o aprendizaje profundo.

La primera parte de este capítulo se centra en presentar la motivación que nos ha llevado a la realización de este trabajo. A continuación de esto procederemos a describir el contenido de los capítulos que tiene esta memoria.

1.1 Motivación

Como se ha comentado antes, los algoritmos de aprendizaje automático tienen como objetivo el aprendizaje de una función que transforme la información de entrada en un valor de salida. Con tal de lograr este objetivo (aprender dicha función), estos algoritmos intentan encontrar patrones y relaciones entre los datos de entrada que expliquen lo mejor posible la relación que existe entre la información que les es dada y el concepto abstracto que representan. Con ello se consigue (con más o menos éxito), crear un modelo que generalice lo suficiente lo aprendido con esos datos iniciales para que, ante datos de los que no se sabe el concepto que representan, encontrarlo.

Las técnicas de aprendizaje profundo han sido usadas durante décadas pero no han tenido mayor impacto. Sin embargo, es ahora, durante los últimos años, cuando se han vuelto a popularizar, como se explica en [6], debido al crecimiento del poder

computacional de las máquinas (en concreto, con la utilización de unidades de proceso gráfico (GPU) para la computación numérica), la abundancia de datos con los que trabajar y el abaratamiento para conseguirlos. Todos estos condicionantes junto con los buenos resultados que se han obtenido aplicando estas técnicas a una diversa serie de problemas han hecho que se popularice su uso.

Resulta muy atractivo pensar en los problemas a los que se ha aplicado este tipo de técnicas. Desde la detección del correo basura que recibimos hasta la conducción autónoma de un vehículo. Pero los problemas a los que se pueden aplicar son muchos más y a cada cuál más complejo que el anterior. Ser capaz de reconocer las emociones que muestran las personas en diversas situaciones y analizarlas es una función que tiene mucha utilidad en el diseño de interfaces persona-computador, en el marketing y en diversas situaciones sociales como la política. ¿A quién no le gustaría saber si un político miente?

Otra de las motivaciones que tiene el usar este tipo de técnicas es la similitud que tiene con el funcionamiento del cerebro, en el que las señales sensoriales se propagan a través de una compleja red jerárquica. Además, la estructuración en capas la podemos ver como el aprendizaje a distintos niveles de abstracción que nos ayudan a darle sentido a la gran cantidad de información a procesar (los píxeles de una imagen, o las distintas frecuencias de una señal sonora).

Con todo lo expuesto anteriormente nos parece interesante meterse de lleno en este ámbito y aplicar este tipo de técnicas a la tarea de reconocimiento de emociones a partir de imágenes. Debido a que la elección del conjunto de datos sobre el que trabajaremos es el que se usó en la competición de la página web Kaggle *Challenges in Representation Learning: Facial Expression Recognition Challenge*[1] y debido a que los resultados de esta competición son públicos, comparar nuestros resultados con los participantes de la competición es una motivación extra que nos empuja a querer abordar y resolver este problema.

[1] https://www.kaggle.com/c/challenges-in-representation-learning-facial-expression-recognition-challenge

1.2 Estructura del proyecto

A continuación se presenta una breve descripción de lo que encontrará el lector en este documento:

-**Capítulo 2**: Descripción del problema a resolver: el reconocimiento de emociones. Así como las aplicaciones que tiene en la vida real y las técnicas punteras que tanto en el campo científico como en el profesional se han desarrollado en los últimos años.

-**Capítulo 3:** Descripción en profundidad de la técnica de aprendizaje automático empleada: las redes neuronales convolucionales. Explicamos los principios en los que se basan, cómo funcionan este tipo de redes y cuáles son las arquitecturas más populares.

-Capítulo 4: Comparativa de las herramientas contempladas para la realización de este proyecto, así como los retos y trabajo que ha significado la realización de este proyecto con la herramienta seleccionada.

-Capítulo 5: Explicación de los experimentos llevados a cabo. Se cubren aspectos como: el hardware empleado para hacer los experimentos, el conjunto de datos de partida y los distintos experimentos lanzados para buscar la solución óptima al problema. Así como la visualización de resultados y las conclusiones a las que se ha llegado mediante la visualización de la información extraída en el entrenamiento.

-Capítulo 6: Conclusiones después de la realización del proyecto.

CAPÍTULO 2

Reconocimiento de emociones

Con el incremento del poder computacional, la mejora de las cámaras de video y la cada vez más presente robotización de los entornos trabajo, la habilidad de procesar imágenes para diversas tareas se ha vuelto uno de los temas candentes en áreas de estudio como el procesamiento de señales, la inteligencia artificial, la física y la neurobiología. Y es que se le encuentran numerosas aplicaciones tanto en el entorno empresarial como en el entorno social. Algunas de estas aplicaciones son: navegación (por vehículos autónomos), detección de eventos (en cámaras de seguridad), control de procesos (en la industria robótica) o análisis de resultados médicos (procesamiento de imágenes para encontrar malformaciones, tumores, fracturas, infecciones, etc.).

En este capítulo describimos el problema central en la que se basa el proyecto: el reconocimiento de emociones a partir de rostros. Describimos en qué consiste, las aplicaciones que tiene y cuáles son las técnicas usadas en la actualidad (estado del arte).

2.1 Introducción

La interacción inteligente humano-máquina es un área que siempre ha despertado interés en la sociedad (tanto que, podemos ver muchos ejemplos de este tema, aunque muy perfeccionados, en la ciencia-ficción). Sin embargo, es en los últimos años cuando se ha convertido en algo asequible. Es por ello que se ha convertido en una de las áreas emergentes envolviendo a disciplinas como la psicología, neurociencia y la ingeniería informática.

Los estudios realizados en este tema se centran tanto en la mejora de las interfaces para comunicarse con los ordenadores, como en la mejora de las acciones que realizan estos ordenadores con la información que provee el usuario. Información que, tradicionalmente, el usuario proveía con teclado y ratón y que sin embargo, hoy en día es posible proveer a través de cámaras y micrófonos y hacer que las máquinas “vean” y “escuchen”, procesando esa información para actuar acordemente.

Se cree que para alcanzar una comunicación verdaderamente inteligente, es necesario que el ordenador exhiba comportamiento humano, y para ello, una de las habilidades que necesita desarrollar es la de entender el estado emocional de la persona con la

que se comunica. En la próxima sección explicamos las formas en las que se puede analizar este estado emocional y las aplicaciones que tiene este tipo de tecnología en la industria con ejemplos concretos.

2.2 Reconocimiento de emociones y sus aplicaciones

Los humanos interaccionan principalmente por medio de la voz, sin embargo, el lenguaje corporal es también una fuente de información que apoya al discurso oral. Por medio de gestos (voluntarios o involuntarios) y por medio de emociones transmitimos sentimientos y sensaciones que los humanos somos capaces de procesar e introducir en el contexto de la comunicación de la que toman parte. Por tanto, la evidencia de que el reconocimiento de emociones se pueda catalogar como "inteligencia" es bastante fuerte si tenemos en cuenta que se cataloga como enfermedad la no percepción de las mismas (como el síndrome de Asperger).

Para estudiar las emociones humanas hay que tener en cuenta las dos formas principales en las que se exhiben y transmiten dichas emociones; a través del discurso oral y la expresión facial. Estas dos han sido fuentes de estudio desde el punto de vista ingenieril para tratar de identificar emociones en audio y/o video. Analizando tanto video como audio por separado se ha conseguido identificar emociones en variedad de ocasiones (como es el caso de [5], [8] y [14]) de forma satisfactoria, pero como se explica en [2], el empleo de estos dos canales en conjunto para identificar emociones ha mostrado ser mucho más preciso (97%) que el estudio de los dos canales de comunicación por separado (75% en el caso de audio y 70% en el caso de video). Estos resultados satisfactorios no han dejado a la industria indiferente y podemos encontrar aplicaciones que toman ventaja de analizar las emociones de las personas en beneficio propio.

Una de estas aplicaciones tiene el nombre de *Affectiva,* tecnología desarrollada en el *Massachusetts Institute of Technology* (MIT), la cual es usada por marcas como Coca-Cola para evaluar sus anuncios. El *software* capta imágenes del sujeto evaluado a través de una webcam y analiza sus niveles de sorpresa, diversión o confusión y las compara con otras personas de distinta demografía con el fin de intentar vislumbrar el impacto que tendrá el anuncio. Esta tecnología también se emplea en el terreno político para evaluar la respuesta de la gente ante un debate de esta índole.

Figura 2.1: *Software* Affectiva en acción

Otros ejemplos son, el caso de Eyeris, empresa que trabaja con compañías que producen sistemas empotrados (como empresas automovilísticas o robóticas) y que incorporan su *software* de análisis facial y reconocimiento de emociones, o el caso de nViso, el cual provee reconocimiento de emociones en tiempo real para web y móvil a través de su API.

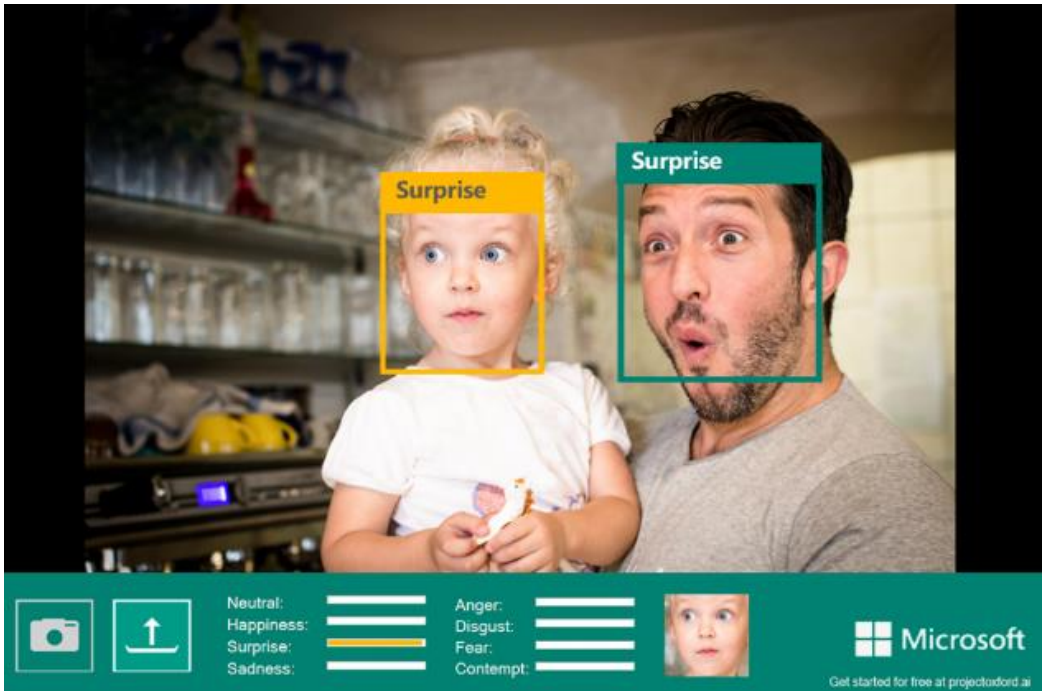

Figura 2.2: Ejemplo de uso de la herramienta *Microsoft Project Oxford Tools* en aprendizaje automático.

En el siguiente apartado describimos las técnicas que usan este tipo de aplicaciones para funcionar de manera eficaz.

2.3 Estado del arte

Como ya hemos comentado anteriormente, la tarea del reconocimiento de emociones, se puede hacer analizando el discurso oral o mediante el reconocimiento de expresiones faciales; debido a que nuestro trabajo se centra en lo segundo, desestimamos el estado del arte y los avances hechos en el campo del reconocimiento de voz.

Sin embargo, antes de entrar de lleno en las técnicas empleadas, cabe hacer una breve descripción de los problemas que uno tiene que enfrentar y solucionar para desempeñar esta tarea correctamente. Las características que dotan de utilidad a un sistema de reconocimiento de emociones a partir de expresiones faciales son: ser capaz de trabajar tanto con videos como con imágenes, trabajar en tiempo real, ser robusto frente a diferentes condiciones lumínicas, ser independiente de la persona (no debería atender al pelo, etnia o género de la persona), ser capaz de trabajar con rostros desde diferentes ángulos y un largo etcétera que podríamos añadir a esta lista para modelar nuestro sistema de reconocimiento ideal. Como podemos ver, la larga lista de requisitos para ser un sistema robusto exige que a medida que se sofistican las técnicas para resolver los problemas antes mencionados se produzca una modularidad del proceso para que cada parte se centre en una tarea específica. Es por ello que este tipo de sistemas se dividen en tres módulos principales: (1) detección y seguimiento del rostro, (2) extracción de características y (3) clasificación de expresiones. Hablaremos en profundidad del apartado 2 y 3. El apartado 1 queda fuera del alcance de este proyecto.

Para que un sistema sea capaz de reconocer y clasificar cualquier cosa, este requiere de datos de entrada que representen el estado del mundo que se quiere estudiar. Estos datos (imágenes en este caso) se filtran para obtener aquellos realmente relevantes. Los sistemas sofisticados utilizan, antes de realizar la tarea de reconocimiento de la emoción, un software de reconocimiento facial que detecta el rostro y traza una serie de puntos de control a lo largo del mismo que posiciona elementos clave como, ojos, labios, cejas, mejillas, etc. En la fig. 2.3 podemos ver un ejemplo de este *software.* Este *software* desarrollado por Tao and Huang [22] genera un volumen 3D del rostro dividiéndolo en 16 volúmenes Bezier. De este tipo de

software se extrae las características que se creen que serán relevantes para la tarea de reconocimiento en cuestión.

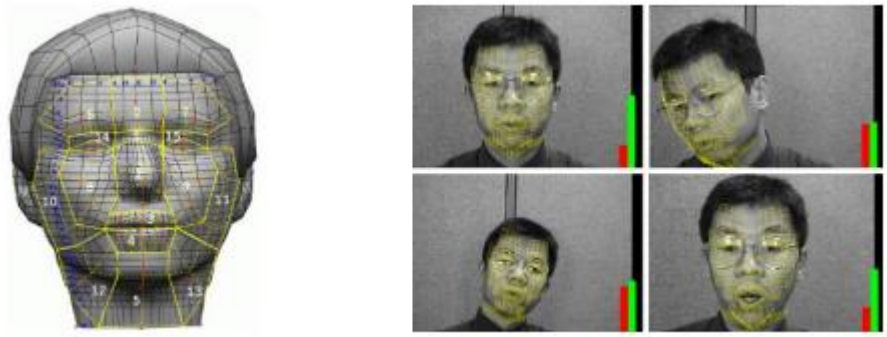

Figura 2.3: Los 16 volúmenes Bezier calculados en tiempo real. **Fuente**: *Face Expression Recognition and Analysis: The State of the Art, Vinay Bettadapura, 2012*

Los movimientos de estos diferentes volúmenes o regiones son cuantificados y asociados (para ver cómo se lleva a cabo consultar [4]) a movimientos de los músculos de la cara. Estudios se han realizado desde el punto de vista anatómico para tratar de catalogar expresiones en función de los movimientos de los músculos del rostro como el llevado a cabo por Paul Ekman[7], creador del Sistema de codificación facial (FACS, *Facial Action Coding System*). Así pues, este sistema utiliza AU (*Action Units*) que son movimientos de un músculo o músculos que producen un cambio en el rostro. Un AU es, por ejemplo, abrir la mandíbula. Sin embargo, no todos se guían con este sistema de codificación (como se da en [4]) debido a sus limitaciones (los movimientos pueden ser representados por un conjunto de parámetros los cuales son suficientes para determinar la pose sin considerar la expresión facial como conjunto).

Una vez extraídos estos parámetros, se introducen en un clasificador para determinar la emoción que representan. Gran parte del esfuerzo se ha puesto en la evaluación de cómo diferentes clasificadores desempeñan esta tarea. Cohen et al. en [4] hace un estudio de clasificadores estáticos como *Naïve Bayes* (NB), *Tree Augmented Naïve Bayes* (TAN) y *Stochastic Structure Search* (SSS) y dinámicos como Modelos Ocultos de Markov (*Hidden Markov Model*, HMM). Por estáticos nos referimos a que las características que toman para realizar la clasificación son de *frames* individuales mientras que los dinámicos toman en cuenta la secuencia de características extraídas de diversos *frames* a lo largo del tiempo.

Los resultados de Cohen et al. muestran que los clasificadores NB daban mejores resultados a pesar de que estos hacen la asunción de que no existe una dependencia entre las diferentes características extraídas de los rostros. Esto no es cierto en muchos problemas del mundo real, y en el caso de las expresiones faciales se da el

mismo hecho (existe una gran correlación entre los movimientos faciales y la muestra de emociones).

Otros clasificadores empleados por la comunidad científica en la tarea y ligeras modificaciones de los mismos son las máquinas de vectores de soporte (*Support Vector Machines*, SVM), HMM y redes neuronales. También es usual combinar varios de estos clasificadores y realizar un proceso de votación; la emoción que más se vote por los distintos clasificadores para un dato de entrada es la que dará como resultado el sistema.

En este proyecto utilizamos para afrontar el problema de reconocimiento de emociones un tipo de redes neuronales llamado redes neuronales convolucionales. No requerimos de un *software* de reconocimiento facial ya que los datos con los que trabajamos son caras correctamente centradas y recortadas. Lo interesante, además, es que no necesitamos confiar en el diseño de la extracción de características relevantes del rostro. Dejamos al modelo que las aprenda automáticamente. Una de las preguntas que se intenta averiguar es si este tipo de técnica rivaliza con las técnicas que confían en la extracción manual de características.

CAPÍTULO 3

Redes neuronales convolucionales

En este capítulo explicamos en detalle la técnica de aprendizaje automático usada en este proyecto. Para poner al lector en contexto, primero describimos los principios en los que se basan las redes neuronales, y a continuación explicamos el modelo completo.

Hay que destacar que el tipo de aprendizaje automático que describimos y empleamos es de tipo supervisado. Esto significa que el valor deseado de salida, o dicho de otra forma, la idea que representa cada una de las muestras de un conjunto de datos de entrada se sabe con antelación. Por tanto, durante la fase de entrenamiento suplimos a nuestro modelo con pares: imagen de entrada y etiqueta correspondiente. El modelo tratará de inferir una función que pueda ser usada sobre imágenes sobre las que no se sabe la etiqueta para hallarla correctamente.

3.1 Redes neuronales

Una red neuronal es una estructura de neuronas interconectadas las cuales transfieren información de unas a otras. Las neuronas, en lugar de estar dispuestas de forma arbitraria, están dispuestas en capas formando un grafo acíclico. Esto significa que las salidas de unas neuronas serán las entradas de las neuronas con las que se encuentran conectada. Sin embargo, como se ha puntualizado anteriormente, no se permiten ciclos porque implicaría un bucle infinito en el recorrido hacia adelante de la red (explicaremos este concepto más adelante).

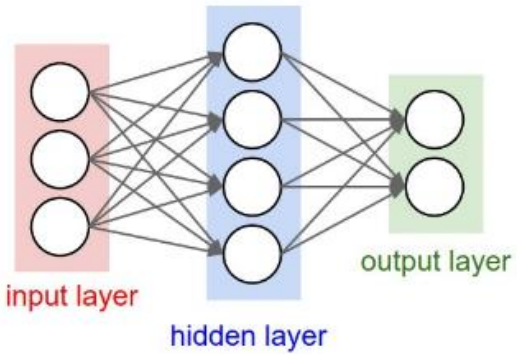

Figura 3.1: Ejemplo de red neuronal con 2 capas. Capa de entrada con 3 neuronas y una capa oculta de 4 neuronas con una capa de salida de 2 neuronas.

En la fig. 3.1 podemos observar un ejemplo de red neuronal. Siendo este tipo de redes el más común, se conocen como *fully-connected*. Esto es debido a que cada neurona está conectada a todas las neuronas de la capa siguiente. Podemos distinguir, por tanto, la capa de entrada, la cual representa el vector de datos de entrada y que tendrá tantas neuronas como elementos tenga el vector. En el caso de una imagen, el vector de entrada serían los píxeles de dicha imagen recolocados como un vector, por lo que, cada píxel vendría representado por una neurona diferente. A continuación nos encontramos con un número arbitrario de capas ocultas. Tanto el número de capas ocultas presentes como el número de neuronas en cada capa es algo desconocido y que tiene que ser probado empíricamente para averiguar cuál es la configuración que funciona mejor para un problema concreto. Estas capas ocultas transforman los datos de entrada a través de los pesos que las neuronas tienen entre sí (esto son, las conexiones) y, junto con la función no lineal presente, actúa como extractor de características. Finalmente, la capa de salida representa las posibles etiquetas o clases en las que se divide el conjunto de datos del problema. Para el ejemplo de la figura, los vectores de 3 elementos de entrada se dividirán y clasificarán en 2 clases.

Explicado más formalmente, las transformaciones que se producen a través de la red (entre capas) se modelan, matemáticamente, de la siguiente forma:

$$f_l(v) = g(Wv + b)$$

Donde W es la matriz de pesos de las conexiones entre neuronas. Cada fila de la matriz corresponde con los pesos o fuerza de conexión que existe entre la neurona de la capa l-*1* y la del resto de neuronas de la capa l. v es el vector de entrada a la capa l y b es un vector que representa el *bias*. La función g se le llama función de activación y se trata de una función no lineal que define el tipo de neuronas con la que estamos tratando.

Por tanto, la serie de transformaciones que sufre el vector de datos de entrada, sea este, v, a través de la red tomando como ejemplo la red de la figura y siendo la capa de entrada la número 0 es:

$$f(v) = f_2(f_1(v))$$

Y, extrapolado a un número arbitrario L de capas:

$$f(v) = f_L(f_{L-1}(\dots f_1(v)))$$

Como ya hemos comentado anteriormente, la elección de la función de activación define el tipo de neurona con la que estamos tratando. Existen muchas de estas funciones (sigmoid, tanh, maxout, ReLU, etc) pero nosotros solo explicaremos las más relevantes.

Por un lado tenemos la función sigmoid, la cual transforma el valor x de entrada en un valor de salida entre 0 y 1:

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

Esta función ha sido usada frecuentemente en redes neuronales debido a la similitud que tiene con el disparo de una neurona. De no disparar (0) a disparar cuando está completamente saturada (1). Sin embargo, recientemente ha caído en desuso debido a los problemas que acarrea. El más significativo ocurre cuando la "x" es muy pequeña o muy grande (saturación); esto resulta en un gradiente muy pequeño, lo que ralentiza mucho el proceso de aprendizaje.

Estos problemas han hecho que se popularice otras funciones que mitiguen este problema, como la función lineal rectificada (*Rectified linear Unit),* o *ReLU.* La cual ha ganado popularidad por acelerar el proceso de aprendizaje de la red como se describe en [12] y, en contraste con las funciones sigmoid o tanh que requieren operaciones costosas (como la exponencial), esta función es tan simple como:

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

Sin embargo, la elección del factor de aprendizaje es una de las cosas que hay que tener en cuenta ya que un valor muy alto puede hacer que las neuronas "mueran". Esto es debido a que un gradiente con un valor alto pasando a través de una neurona ReLU puede hacer que los pesos se actualicen de una forma en la que esa neurona no

se vuelva a activar nunca y que siempre valga 0. Sin embargo, la probabilidad de que esto pase se puede reducir ajustando el factor de aprendizaje.

Estas funciones de activación y por tanto, tipos de neuronas, se suelen utilizar en las capas ocultas. La capa de salida es un caso especial debido a que el resultado que obtengamos de ellas representará en cierta medida la puntuación conseguida en cada clase (en una tarea de clasificación), los cuales serán valores arbitrarios, o algún tipo de valor objetivo realmente útil (en el caso de regresión). Es por ello que se usa la función softmax:

$$f_j(z) = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{f_k}}$$

Dicha función asigna a la salida de la neurona j un valor proporcional con respecto a la salida del resto de neuronas. Con ello, el conjunto de valores de salida de todas las neuronas de la capa de salida sumarán 1 y por tanto, podremos tomar esas salidas como la probabilidad que tiene la entrada que hemos introducido en la red de ser de esa clase. Clasificaremos dicha muestra en la clase que represente la neurona con mayor valor.

3.2 El algoritmo *backpropagation*

Hemos hablado de qué es una red neuronal, como está formada, cuál es su funcionamiento y cómo se usa para, en nuestro caso, clasificar unos datos de entrada en la clase correspondiente. Recordemos que una red neuronal es alimentada con una muestra (la cual queremos clasificar) y como resultado obtenemos una puntuación para cada clase que nos dice la "probabilidad" de pertenencia a esa clase. Sin embargo, el hecho de que una red neuronal acierte en la predicción es muy improbable con unos parámetros (esto son, los pesos de las conexiones de las neuronas y los *bias*) arbitrarios. Es necesario, por tanto, una forma de hacer que la red aprenda qué parámetros son los adecuados para desempeñar la tarea satisfactoriamente.

Esa es la función del algoritmo *backpropagation*. La idea intuitiva que hay detrás de este algoritmo es la de minimizar una función objetivo que calcula el error entre la clasificación que ha realizado una red de unas muestras concretas y de la clasificación correcta de esas muestras. El algoritmo debe adaptar los parámetros de la red para intentar reducir el error. Para saber en qué medida ha de modificar los pesos para reducir el valor de la función objetivo el algoritmo calcula el vector gradiente de los parámetros el cual indica el incremento o decremento del error obtenido cuando el valor de los pesos es ligeramente modificado. Estamos, por tanto, buscando el mínimo de la función de error en un espacio dimensional definido por los valores de los pesos.

Con dicho gradiente buscamos la dirección en la que tienen que cambiar los parámetros para conseguir un error menor en la función objetivo.

El proceso de aprendizaje se aplica en las redes neuronales en 2 pasos. El primero consiste en hacer pasar una muestra por la red para obtener la salida que esta produce (esto es, la clase en la que se ha clasificado la muestra). Esto se conoce como paso hacia delante. Una vez obtenida la salida, podemos compararla con la salida correcta y calcular así el error. El segundo paso consiste en propagar ese error desde la capa de salida hacia atrás calculando las derivadas parciales con respecto a los pesos de cada capa para saber qué impacto tiene un peso en concreto (aumentando o disminuyendo) en la función de error si lo modificamos ligeramente. Sabiendo este valor, sabemos cómo tenemos que modificar ese peso en concreto para minimizar el error en la función objetivo. Para conseguir esto es necesario usar la regla de la cadena la cual es una fórmula para encadenar correctamente las derivadas de una función compuesta por otras funciones.

El entrenamiento de una red neuronal con este algoritmo se hace sobre un conjunto de muestras de entrenamiento para las cuales se sabe su etiqueta (por etiqueta nos referimos al concepto que representa la imagen). Para proceder a actualizar los pesos se suele utilizar el método de descenso por gradiente. Este método calcula el gradiente medio sobre un número pequeño de muestras para después, actualizar los pesos de forma correcta. Este pequeño número de muestras utilizadas para hacer una única actualización de parámetros se conoce como *batch.* Este proceso se repite sobre distintos *batch* hasta que el error se estabiliza. Utilizamos el término *epoch* cuando todas las muestras han sido usadas una vez para la actualización de los pesos. Es normal hacer ver a la red varias veces todos los ejemplos de entrenamiento (o lo que es lo mismo, entrenar durante varios *epochs*).

3.3 Redes neuronales convolucionales

Las redes neuronales convencionales (las expuestas hasta ahora) presentan un problema difícil de solucionar cuando se trabaja con muestras de alta dimensionalidad (como las imágenes). La naturaleza de estas redes (el hecho de que las capas de neuronas están completamente conectadas entre ellas) hace que, por un lado, no escalen bien cuando el número de dimensiones de los datos de entradas es grande y que, por otro, en el caso de las imágenes, el hecho de dedicar una conexión de un píxel a todas las neuronas puede que no funcione bien en la práctica. Este hecho lo explica Krizhevsky en [13], argumentando que podemos observar propiedades especiales que tienen las imágenes que no tienen otros conjuntos de datos simplemente mirando la matriz de covarianza. En ella se observa la principal

característica que poseen las imágenes, esto es, los píxeles están fuertemente relacionados con píxeles cercanos y débilmente relacionados con píxeles lejanos.

Sabiendo esto, podemos intuir que cuando se trata de imágenes, necesitamos otra herramienta que ataje este problema de la conexión total y se centre en regiones locales de la misma (pues es ahí donde los pixeles como conjunto tienen sentido) para extraer información útil.

Es aquí cuando entran en juego otro tipo de redes neuronales, las cuales son, las redes neuronales convolucionales (*Convolutional neural network*, CNN). Las CNN hacen la asunción de que los datos de entrada son imágenes, lo que permite codificar ciertas propiedades de las antes comentadas en la arquitectura. Esto hace que este tipo de arquitectura reduzca drásticamente el uso de parámetros (en comparación con las redes neuronales convencionales) y haga el proceso de clasificación más eficiente.

3.3.1 Visión general de la arquitectura

Como se ha explicado anteriormente las redes neuronales convencionales no escalan bien con las imágenes. Para hacer consciente al lector de este problema presentamos el siguiente ejemplo. Si disponemos de una imagen de 300x300 en color (canales R, G y B), esto nos llevaría a tener neuronas con un total de 300x300x3 = 270.000 conexiones. Y, por supuesto, si queremos tener varias de estas neuronas, el número de parámetros crece de manera desproporcionada. Claramente, la conectividad total de las neuronas es un desperdicio de espacio cuando se trata de imágenes.

Las CNN se aprovechan de que los datos de entrada son imágenes para disponer la arquitectura de manera acorde. Al contrario que en las redes neuronales convencionales, las neuronas se disponen de forma tridimensional. Así pues, cada capa de neuronas tendrá ancho, largo y profundidad. Aunque empleamos aquí el término profundidad, hay que aclarar que por profundidad nos referimos a la tercera dimensión de la capa de neuronas, no a la profundidad de la red. Varias capas de este tipo podrán ser dispuestas una a continuación de la otra las cuales sí que determinarán la profundidad de la red. Como veremos a continuación, cada neurona está conectada a una parte pequeña de la imagen (evitando así el problema que surge con el uso de las redes neuronales convencionales).

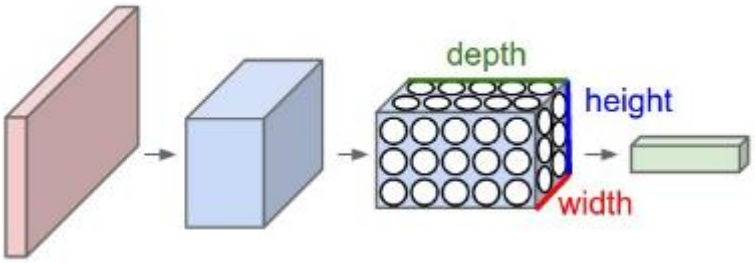

Figura 3.2: Disposición de las neuronas en una red convolucional de forma tridimensional.

3.3.2 Tipos de capas empleadas

Como las redes neuronales convencionales, las CNN disponen las neuronas en capas. Sin embargo, las CNN hacen uso de tres tipos distintos de capas para configurar la arquitectura completa. Estas son: capa convolucional, capa *pooling*, y capa *fully-connected*.

3.3.2.1 Capa convolucional

En esta capa es donde se encuentra la idea clave de este tipo de red neuronal y donde se produce la mayor parte del esfuerzo computacional. A continuación describiremos la operación llevada a cabo.

La idea principal es conectar cada neurona a una región local del volumen de entrada. Cuán grande sea esta región es un parámetro que definiremos nosotros y constituirá el campo receptivo de la neurona. Si desplazamos esa región a lo largo del ancho y largo del volumen de entrada como si de una ventana que solo nos deja ver esa zona se tratara y calculamos el producto escalar de la región con los pesos de la neurona pertinente, obtenemos como resultado un mapa de activación bidimensional que responde a lo que se ha detectado para ese filtro en todo el espacio de entrada. Por supuesto, cada capa convolucional tendrá varios de estos filtros y cada uno de ellos producirá un mapa bidimensional distinto los cuales serán puestos a lo largo de la dimensión profundidad para formar el volumen de salida. En la fig. 3.3 se presenta de forma gráfica cómo funciona la computación de esta capa.

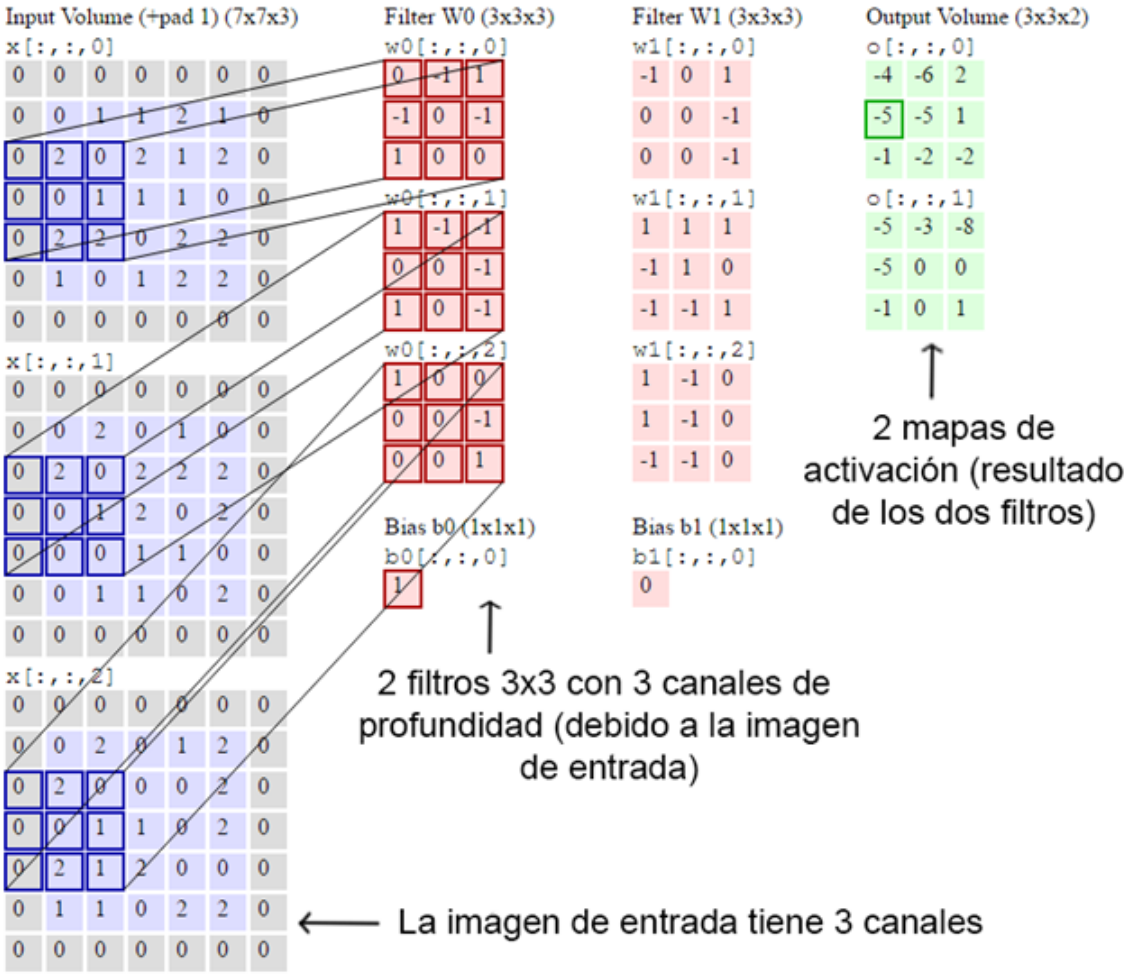

Figura 3.3: Computación de la convolución de una capa convolucional con 2 filtros de tamaño 3x3 para una imagen de entrada con 3 canales.

Sin embargo hay que hacer notar que, mientras que la extensión de este campo receptivo a lo largo del ancho y largo del volumen de entrada es local, a lo largo de la profundidad es total. Explicaremos esto mejor con un ejemplo. Si suponemos que el volumen de entrada es por ejemplo 28x28x3 y definimos un campo receptivo de 7x7, cada neurona en la capa convolucional tendrá conexiones a una región 7x7x3, haciendo un total de 7x7x3 = 147 parámetros (más el bias) por neurona. Notar que la conectividad a lo largo de la profundidad del volumen de entrada es total.

Por tanto, las dimensiones del volumen de salida vendrán determinadas por: el tamaño del campo receptivo, el tamaño del desplazamiento que se emplea de este campo receptivo y del número de filtros que queramos emplear para esa capa.
Tomando el ejemplo anterior, si partimos de una imagen con dimensiones 28x28x3 y le aplicamos una convolución con un campo receptivo de 7x7, un desplazamiento del

campo de 1 píxel y 10 filtros, el volumen que obtendremos como salida será: 22x22x10.

Sin embargo, al disponer las neuronas de forma tridimensional y conectar cada una a lo largo de toda la profundidad del volumen de entrada podemos ver, que aunque conectemos cada neurona a una región local del volumen, los parámetros que estamos manejando siguen siendo demasiados. Para hacernos una idea, tomando el ejemplo anterior, la capa convolucional tendría 22x22x10 = 4840 neuronas, cada una de las cuales con 7x7x3=147 parámetros (más el bias) correspondientes a su campo receptivo, lo cual hace un total de 4840 * 147 = 711.480 parámetros (sin contar los bias) sólo para la primera capa convolucional. Claramente, esto no se puede manejar.

El problema anterior se solventa mediante la compartición de parámetros. Para conseguirlo, se hace la asunción de que si la información para calcular una posición espacial es útil, entonces, debería ser útil para calcular una posición distinta. Con ello, todas las neuronas de una misma profundidad (si dividimos el volumen en función de su profundidad), o filtro, compartirán los mismos parámetros. Retomando el ejemplo anterior, cada porción de 22x22 neuronas compartirán los mismos 7x7x3 = 147 parámetros (más el bias). Así reduciremos el número de parámetros de 711.480 a 7x7x3x10 = 1.470.

3.3.2.2 Capa pooling

Es frecuente introducir este tipo de capas entre capas convolucionales para reducir el tamaño de la representación y reducir la cantidad de parámetros con el doble objetivo de, por un lado reducir el esfuerzo computacional requerido y por otro, controlar el *overfitting*. La capa *pooling* opera independientemente en cada nivel de profundidad realizando la operación max. Es común realizar esta operación con una ventana de 2x2 y un desplazamiento de 2 píxeles, lo que reduce el volumen de entrada (en términos de anchura y altura) a la mitad. En la fig. 3.4 puede verse un ejemplo de su uso.

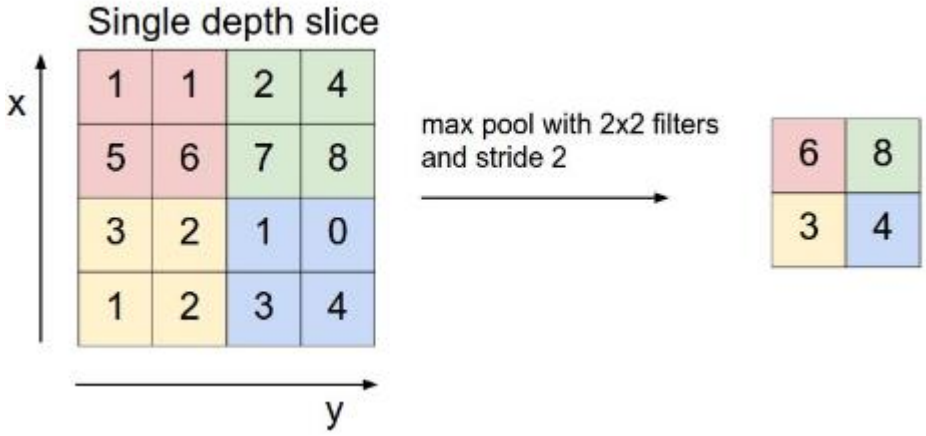

Figura 3.4: Operación *max pooling* con un tamaño de 2x2 y un desplazamiento de 2.

También se pueden aplicar otras operaciones como la media o la norma L2, pero su uso es menos frecuente.

3.3.2.3 Capa fully-connected

Este tipo de capas son las mismas que se usan en las redes neuronales convencionales y, como comentaremos más adelante, suelen colocarse al final de la red convolucional para calcular las puntuaciones de cada clase. Cabe destacar que para poder usar esta capa a continuación de una convolucional o una *pooling* hay que hacer una conversión, la cual implica, recolocar los elementos de la matriz tridimensional en un vector. Su funcionamiento es el mismo al explicado en la sección 2.1.

3.3.3 Arquitecturas comunes

Una vez descritas los tipos de capas que se utilizan en las redes convolucionales solo falta describir cómo se ensamblan en la práctica. Si nos fijamos en aplicaciones exitosas de este tipo de redes (como las presentes en [12] y [14]), vemos que hay algunos patrones que se repiten.

La forma más común de CNN está constituida por un número arbitrario de capas convolucionales seguidas por una capa *pooling*, repitiendo este patrón hasta que las dimensiones de la imagen se han reducido lo suficiente. A continuación se colocan un número arbitrario de capas *fully-connected*, la última de las cuales nos dará la puntuación para cada clase. En la fig. 3.5 se puede ver la red empleada por LeCun en la clasificación de dígitos en el conjunto de datos MNIST. Dicha red consiguió un error < 1%.

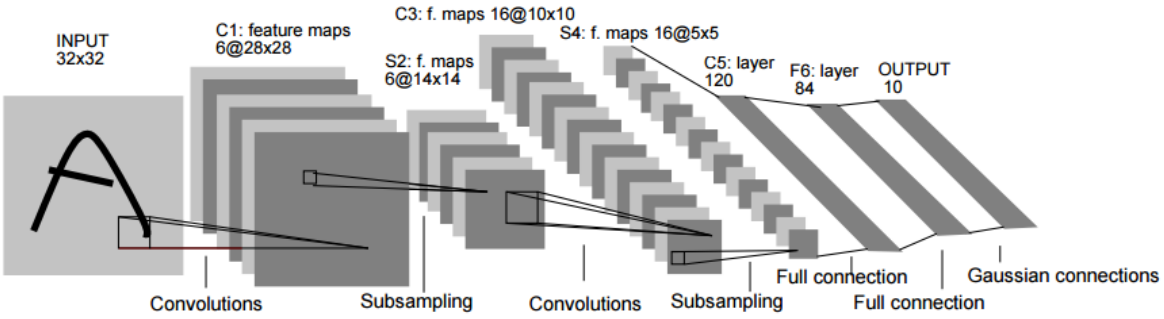

Figura 3.5: Arquitectura de LeNet-5, una red convolucional para la clasificación de dígitos. Fuente: Gradient- *Based Learning Applied to Document Recognition, Yann LeCun (1998)*

En la fig. 3.6 podemos ver que la configuración de esta red (siendo FC, la capa *fully-connected*) es:

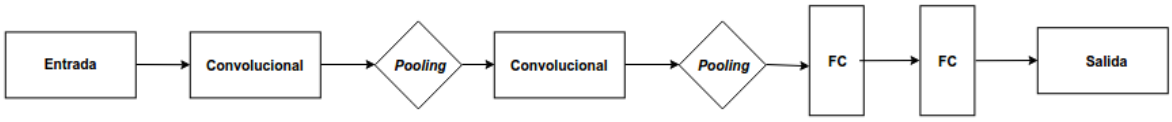

Figura 3.6: Representación esquemática de la configuración de la red LeNet-5.

Se aplica el patrón convolucional-*pooling* dos veces hasta reducir la representación del dígito de entrada hasta filtros de 5x5, momento en el cual se pasan a un par de capas *fully-connected*, obteniendo al final la salida deseada.

Esta red fue la primera aplicación exitosa de una red convolucional y se le conoce con el nombre de LeNet. Junto con esta han surgido varias arquitecturas de redes convolucionales que tienen nombre. Cabe destacar la que lleva el nombre de AlexNet[12], la cual, en 2012 popularizó el uso de este tipo de redes al disminuir el error de clasificación del mejor método evaluado en la competición de ImageNet a casi la mitad. Otras redes conocidas por sus logros que tienen nombre propio son ZF Net[23], GoogLeNet[21], VGGNet[19] y ResNet[10].

La complejidad de estas redes no es una broma. La fig. 3.7 y 3.8 muestran la arquitectura de AlexNet y ZF Net.

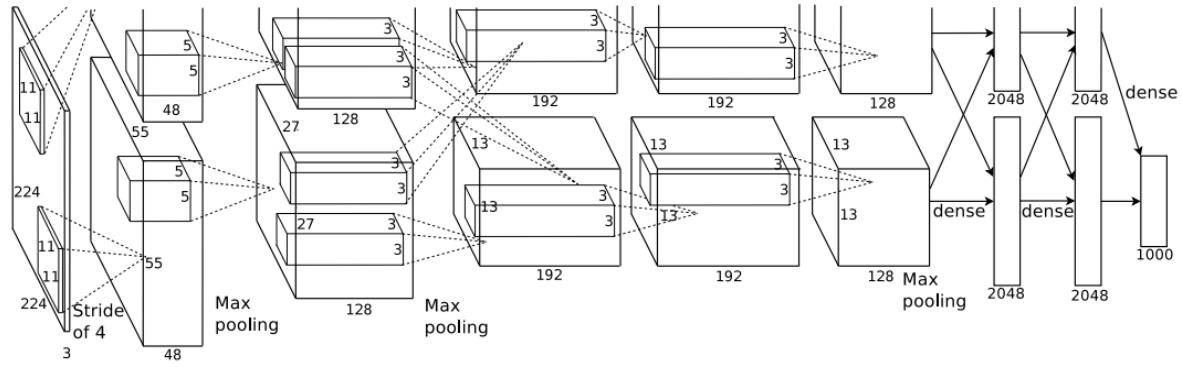

Figura 3.7: Arquitectura de AlexNet. Ganadora de *ImageNet ILSVRC challenge* en 2012[2]. La red está distribuida en 2 GPUs. La primera realiza los cálculos de la parte superior de los filtros mientras que la segunda realiza los cálculos de la parte inferior. Las GPUs solo se comunican en ciertas capas.
Fuente: *ImageNet Classification with Deep Convolutional Neural Networks, Alex Krizhevsky (2012)*

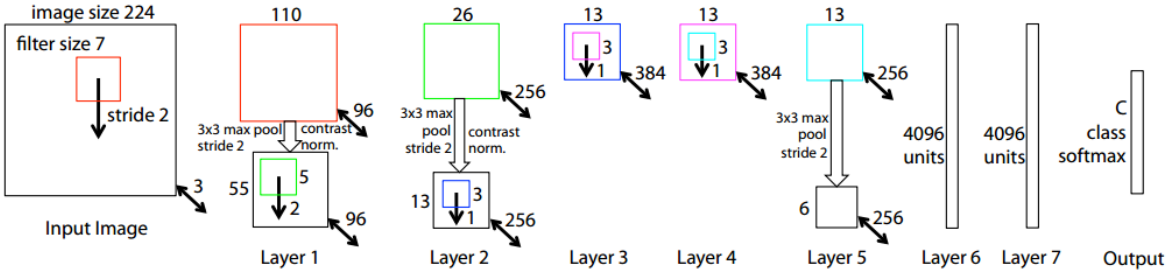

Figura 3.8: Arquitectura de ZFNet. Ganadora de *ImageNet ILSVRC challenge* en 2013[3]
Fuente: *Visualizing and Understanding Convolutional Networks, Matthew D. Zeiler (2013)*

[2] http://www.image-net.org/challenges/LSVRC/2012/
[3] http://www.image-net.org/challenges/LSVRC/2013/

CAPÍTULO 4

Tecnología Empleada

Con la creciente popularización de las CNN, ha surgido recientemente multitud de *toolkits* que hacen del implemento de las mismas una tarea algo más sencilla y que proporcionan ventajas como la ejecución de procesos en GPU en lugar de la CPU. En este capítulo se describe tanto las opciones que hemos valorado como las razones que nos han llevado a la elección del que finalmente hemos usado. También se realiza una descripción de cómo se trabaja con la solución elegida y los retos que presenta.

4.1 Introducción: Elección de la herramienta

La búsqueda de una herramienta flexible que nos permitiera implementar redes neuronales convolucionales para afrontar este problema y se ajustara a nuestras necesidades fue sin duda un elemento determinante en la realización de este proyecto. Estas necesidades son:

- Tiene que ser una herramienta que utilice Python como lenguaje de programación ya que este permite un prototipado rápido de código

- Tiene que tener soporte para la ejecución de operaciones en GPU (las redes convolucionales requieren un poder computacional que no se puede satisfacer con una CPU)

- Tiene que tener una comunidad activa y una buena documentación para poder consultar en caso de duda.

A pesar de existir otras herramientas muy utilizadas pero que, tristemente, emplean C++ como lenguaje de programación como Torch o nnForge u otras herramientas que utilizan un pseudo lenguaje para la definición de las redes como Caffe, los candidatos que encontramos que cumplen todos los requisitos antes mencionados son dos : TensorFlow y Theano.

TensorFlow es una herramienta que fue liberada por Google poco antes de comenzar con este proyecto. Se trata de una librería de Python para computación numérica que trabaja con un grafo en el que los nodos representan operaciones matemáticas

mientras que los extremos del grafo representan arrays multidimensionales (llamados *tensor*) que nos permite desplegar modelos de aprendizaje profundo tanto en la CPU como en la GPU.

Similar a TensorFlow, Theano es una librería para Python enfocada a la computación numérica eficiente que involucra arrays multidimensionales. Theano también nos permite ejecutar las operaciones numéricas tanto en la CPU como en la GPU. Tiene como ventaja que genera y utiliza código C para la evaluación más rápida de expresiones y el cálculo de las derivadas de expresiones simbólicas. Además, se han desarrollado muchos proyectos que hacen uso de Theano para generar los modelos a alto nivel reduciendo la dificultad de implementar arquitecturas dinámicas.

Tras valorar las posibles opciones disponibles para abordar este proyecto nos quedamos con Theano principalmente porque llevaba más tiempo en el mercado, poseía mucha documentación y ejemplos de uso, una comunidad bastante grande que podría ayudarnos en cualquier problema que tuviéramos y, como ya hemos comentado, uno de estos proyectos "envoltorio" que facilitan la experiencia de uso.

En adición a Theano, usamos Blocks, una librería para Python que hace uso de Theano para la implementación de distintas arquitecturas de redes para aprendizaje profundo. Blocks ofrece la facilidad de encapsular los distintos tipos de capas de una red neuronal en *bricks*, o ladrillos. Los cuales se colocan a placer para crear la arquitectura deseada adaptando los tamaños de entrada y salida en función de los parámetros de construcción. También ofrece facilidades para el entrenamiento de redes neuronales ya que posee implementado varias funcionalidades como por ejemplo, el algoritmo de descenso por gradiente para entrenar el modelo, o distintas formas de monitorizar el entrenamiento. Todo lo que Blocks ofrece se encuentra recopilado en [15].

4.2 Preparación de los datos

Cuando se trabaja en problemas de aprendizaje automático es primordial tener los datos en un formato adecuado a la herramienta que vas a utilizar. Normalmente la primera tarea que tienes que hacer es transformarlos y adecuarlos porque no es nada común que ya los encuentres preparados para usar. Estos datos pueden estar corruptos, no ser uniformes (por ejemplo, imágenes de varios tamaños, con nombres distintos, resoluciones o formatos diferentes), etc. Por tanto lo primero que hay que hacer es uniformar los datos y cribarlos para eliminar aquellos defectuosos o que no son relevantes (a no ser que quieras trabajar con datos algo distorsionados para introducir algo de ruido al modelo y darle más robustez).

Lo siguiente que hay que hacer es disponer los datos de forma adecuada para que tu aplicación sea capaz de leerlos y trabajar con ellos. Lo último que hay que hacer es dividirlos en al menos dos subconjuntos, uno para entrenamiento y otro para test. Cuando se trabaja con grandes cantidades de datos (millones) es frecuente dividir a su vez, el subconjunto de entrenamiento en otro subconjunto mucho más pequeño llamado normalmente "conjunto de validación". Este se usa para determinar la arquitectura más adecuada (número de capas, número de neuronas, elección de parámetros, etc.) que se cree que actuará de forma similar cuando se entrene el modelo con los datos de entrenamiento. Con esto se evita tener que esperar días o semanas a obtener los resultados de cada modificación de parámetros cuando se entrena con el conjunto de datos completo.

En nuestro caso, el primer paso no tuvimos que hacerlo ya que los datos estaban preparados de antemano para usarse en la competición. Los datos consistían en un archivo de texto plano organizado en columnas. La primera columna contenía en fila los valores de los píxeles de las imágenes con valores entre 0 y 255, la segunda columna contenía la clase a la que pertenecía esa imagen y la última columna contenía el subconjunto al que pertenecía esa imagen, ya sea entrenamiento o test. Una vez sabido esto hay que adecuarlos a la herramienta en cuestión. Blocks trabaja estrechamente con Fuel, *framework* especialmente pensado para herramientas de aprendizaje automático que te permite leer de forma cómoda los datos desde archivos .csv o .hdf5 hasta archivos binarios de Python. También tiene otras ventajas como la de tener ya implementados varios conjuntos de datos que se usan comúnmente en tareas de aprendizaje automático como MNIST o CIFAR-10, permitir la iteración sobre los datos tanto de forma secuencial como barajada, y la más importante, te permite aplicar transformaciones a los datos al vuelo (por ejemplo, aplicar rotaciones aleatorias a las imágenes, ampliarlas, desplazarlas, etc.).

Para poder emplear este *framework* tuvimos que, primero, leer los datos, transformarlos en arrays NumPy (famosa librería de Python para trabajar con arrays n-dimensionales) y, muy importante, almacenando los datos como números de coma flotante en 32 bits (los experimentos se ejecutarán en la GPU, lo que implica que solo se puede alcanzar una aceleración sustancial frente a la ejecución en CPU si se trabaja con ese tamaño) y a continuación generar un archivo .hdf5 (archivo que leerá Fuel posteriormente). Pero para generar este archivo es necesario especificar una serie de parámetros. Primero, es necesario indicar con forma de diccionario (nombre y valor) los diferentes subconjuntos que van a tener nuestros datos (puesto que todos los datos se guardarán en el mismo archivo), indicando con una tupla principio y fin del subconjunto. Y segundo, es necesario especificar el nombre de cada dimensión de los datos almacenados. Esto quiere decir, que si guardamos una imagen en forma de array bidimensional, la primera dimensión tendrá el nombre de "alto" y la segunda de

"ancho". En nuestro caso tenemos que guardar las imágenes en un array 4-dimensional. La primera dimensión se llama *batch* y representa el índice de cada imagen de nuestro conjunto de datos, a continuación tendremos el "canal", el cual se refiere a los distintos canales RGB de las imágenes y por último tendremos "ancho" y "alto" que representan las dimensiones de la imagen. Todo ello queda recogido en la fig. 4.1. Con esto solo falta, por comodidad, crear una clase que lea automáticamente el archivo generado para poder llamarlo con comodidad sin tener que especificar continuamente la ruta en la que se encuentra.

```
train_partition = (0,57418)
test_partition = (57418,64596)

features[...] = dataset
targets[...] = labels

features.dims[0].label = 'batch'
features.dims[1].label = 'channel'
features.dims[2].label = 'height'
features.dims[3].label = 'width'

targets.dims[0].label = 'batch'
targets.dims[1].label = 'index'

split_dict = {
    'train':{
        'features':train_partition,
        'targets':train_partition
    },
    'test':{
        'features':test_partition,
        'targets':test_partition
    }
}
```

Figura 4.1: Definición de particiones para la creación del archivo que contendrá el conjunto de datos. Las variables *dataset* y *labels* contienen las imágenes y etiquetas de forma correspondiente del conjunto de datos. Archivo *parseToFuelDatasetKaggle.py,* líneas 20-43.

4.3 Construcción de la arquitectura

Una vez tenemos los datos listos lo siguiente es construir la red. Blocks ofrece una interfaz y una serie de facilidades para crear redes neuronales. Entre ellas se encuentra la de crear las capas con una serie de parámetros como el número de filtros, el tamaño de la convolución, el tipo de neurona empleada, etc.). Como muchos de los problemas de aprendizaje automático se reducen a prueba y error, tenemos la necesidad de crear una clase versátil y configurable que, en función de los parámetros que le pasemos por línea de comando, nos cree la red pertinente para poder hacer la mayor cantidad de pruebas que podamos de forma automática sin tener que modificar ni tener que cambiar parámetros dentro del código. Por ello hemos creado una clase que encapsula a la arquitectura de la red en función de estos parámetros.

La red que creamos recibe como parámetros para su construcción una lista con los tamaños de convolución de cada capa convolucional, seguido de una lista con el

número de filtros que tendrá cada capa y la posición de las capas de *pooling*, y por último, una lista indicando el número de neuronas que tendrán las últimas capas *fully-connected.* Algunos de los parámetros de esta red que podrían ser configurables como el tipo de neurona empleado en cada capa, el tamaño del desplazamiento de la convolución o el tamaño del *pooling* están prefijados porque son valores comunes que se emplean. Hablaremos más de estos parámetros en la sección de experimentos.

Esta red será utilizada e instanciada por la función principal la cual recibirá los parámetros de la línea de comandos y realizará el entrenamiento.

4.4 Evaluación y monitorización de resultados

Creada ya la arquitectura solo hace falta realizar el entrenamiento de la red y comprobar su funcionamiento. Para el entrenamiento, Blocks posee una serie de facilidades que nos permiten por ejemplo, seleccionar el tipo de actualización de los pesos (hablaremos de esto más detenidamente en la sección de experimentos) después de cada iteración del algoritmo de descenso por gradiente, elegir cómo queremos que acabe el algoritmo (después de N *epochs*, cuando ya no se reduce el error o simplemente hasta que no llegue a cierto error objetivo). Además, durante el entrenamiento se pueden añadir una serie de "extensiones" al algoritmo que van desde monitorizar el error que se comete después de cada batch, guardar los parámetros de la red después de cada batch para evitar que situaciones indeseados como cortes de luz interrumpan el entrenamiento y se pierdan los progresos, mostrar tiempos de lectura y transformación de datos, entrenamiento, etc. Una vez seleccionado las extensiones que quieres añadir al entrenamiento, el flujo de datos tanto de entrenamiento como de test, la función de coste y la red son entregados a la clase principal que se encarga de correr el algoritmo.

Pese a tener todas estas funcionalidades, a medida que realizamos el proyecto nos damos cuenta que aún falta cierta funcionalidad a implementar para poder trabajar. De este modo, la monitorización de resultados solo se podía hacer viendo lo que se imprimía por la consola, hecho nada práctico porque primero, no ves de forma gráfica los resultados, ni tampoco se almacena en ningún sitio para su posterior análisis. Por ello desarrollamos una nueva extensión para Blocks la cual guarda en un fichero Pandas (Pandas es una librería Python que permite guardar información de forma estructurada como tablas) los parámetros que has seleccionado para monitorizar en el entrenamiento, como el error de test, el error de entrenamiento, el valor de la función coste, etc. Este fichero se crea con un nombre que identifica la infraestructura de la red. Así, cuando más tarde queramos ver los resultados sabemos qué red dió esos resultados. Estos ficheros pandas tienen perfecta integración con la librería Matplotlib,

la cual nos permite crear gráficas para ver los resultados de una forma mucho más directa e intuitiva.

Debido a que el entrenamiento de una red oscilaba, en nuestro caso, entre 30 min y 2h en función del número de *epochs* y el tamaño de la red, lanzar ejecuciones manualmente no era una opción. Es por ello que desarrollamos la funcionalidad de parametrizar todo el entrenamiento desde la línea de comandos. Así podemos crear un *script* que varíe esos parámetros y lance entrenamientos cuando el anterior haya terminado y así poder recogerlos todos y ver cuál ha funcionado mejor. Para realizar esta tarea empleamos la librería que trae Python *argparse*. Con esta librería recogemos tanto parámetros que afectan a la estructura de la red (como las capas convolucionales y de *pooling*, los tamaños de los filtros de cada capa o el número de neuronas de las capas *fully-connected)* como parámetros que afectan al entrenamiento (como el tipo de actualización de pesos, con todos los sub parámetros que estos tienen asociados, el número de *epochs* que durará el entrenamiento, si quieres cargar parámetros previamente guardados para continuar con el entrenamiento o si quieres guardar los parámetros del entrenamiento).

A continuación pondremos como ejemplo una ejecución de la aplicación principal desde la línea de comandos:

```
$> python convNet.py --num-epochs 30 --conv-sizes 5 3 --feature-maps
32 0 32 0 --mlp-hiddens 1024 --step-rule momentum --learning-rate 0.03
--momentum 0.9
```

Con la anterior línea estamos especificando:

- **--num-epochs 30 :** Acabar el entrenamiento después de 30 *epochs*
- **--conv-sizes 5 3:** El tamaño de la convolución de las capas convolucionales. Para la primera capa será de 5x5 mientras que para la segunda de 3x3
- **--feature-maps 32 0 32 0:** Disposición de las capas convolucionales y de pooling. Un número distinto de 0 indica el número de filtros de la capa convolucional y un 0 indica la presencia de una capa *max pooling*. El número de capas convolucionales tiene que estar a la par con el número de parámetros pasado a --conv-sizes para su correcto funcionamiento.
- **--mlp-hiddens 1024:** Número de neuronas de las capas *fully-connected* de la red. En este caso solo tendremos una capa oculta de 1024 neuronas.
- **--step-rule momentum:** Indica el tipo de actualización de parámetros que se llevará a cabo en el descenso por gradiente. Los valores posibles son: scale, momentum y adam.
- **--learning-rate 0.03:** Indica el factor de aprendizaje empleado en el algoritmo de descenso por gradiente.

- **--momentum 0.9**: Sub parámetro necesario por la elección de momentum como el tipo de actualización de parámetros.

También se pueden especificar dónde se quiere guardar los parámetros del modelo una vez acabe el entrenamiento o la carga de parámetros de un entrenamiento anterior con parámetros adicionales. Otras formas de actualización de parámetros requieren de parámetros adicionales que también se pueden especificar desde línea de comandos.

En el siguiente capítulo describimos los diferentes experimentos que hemos realizado con lo anteriormente explicado.

CAPÍTULO 5

Experimentos

Gran parte del tiempo empleado en este proyecto se ha destinado a hacer experimentos con distintas redes y parámetros con la finalidad de estudiar los resultados obtenidos y tratar de encontrar la configuración que más se adecúa a nuestro problema. Este capítulo tiene como objetivo, describir tanto el procedimiento seguido para la realización de los experimentos, como la muestra de los resultados obtenidos. Ya que hemos visto que los cambios más drásticos en el error cometido se producen al cambiar de método de actualización de pesos, abordaremos cada uno de los que hemos probado por separado describiendo su funcionamiento. A continuación haremos una pequeña comparativa de los mejores resultados obtenidos con distintas redes así como de la configuración que obtiene una mayor precisión en la clasificación.

Como hemos visto, el modelo se puede evaluar en función de su precisión (cuántas imágenes es capaz de clasificar bien) pero también podemos verlo desde el punto de vista de cómo clasifica. Si pensamos de este modo surgen otras preguntas como: ¿clasifica todas las emociones con la misma precisión, o existen algunas que son más fáciles de clasificar que otras? ¿Existen emociones que confunda con otras con más facilidad? ¿Cuáles son las partes más representativas de una imagen que hace que lo clasifique en una emoción concreta? Todas estas preguntas son abordadas al final de esta sección.

Sin embargo, antes que nada, describiremos en detalle el hardware sobre el que hemos corrido los experimentos.

5.1 Hardware utilizado

La ejecución de los experimentos se ha llevado a cabo en la GPU de Nvidia GTX 780 ti disponible en una de las máquinas del laboratorio de procesamiento de imágenes de la Universidad Politécnica de Valencia (UPV). Las pruebas se han realizado en la distribución de Linux Ubuntu versión 12.04. En la tabla 5.1 se recogen las especificaciones técnicas de la GPU y en la tabla 5.2 las especificaciones técnicas de la máquina sobre la que está montada. Con esta configuración, los experimentos tardan en ejecutarse, dependiendo principalmente del número de *epochs* y de la complejidad de la red, entre 30 minutos y 2 horas.

Tarjeta gráfica Nvidia GTX 780 ti 	
Núcleos CUDA	2880
Memoria interna	3072 MB
Interfaz de memoria	GDDR5
Ancho de banda de memoria interna	336 GB/s

Tabla 5.1: Especificaciones de la GPU empleada.

Máquina	
Procesador	Intel Core i7-860 @ 2.8GHz
Memoria	12GB DDR3 1333MHz
Tarjeta gráfica	Nvidia GTX 780ti
Disco duro	2TB

Tabla 5.2: Especificaciones de la máquina empleada.

5.2 Conjunto de datos

El conjunto de datos empleados es el proveído por la página web *Kaggle* en la competición *Challenges in Representation Learning: Facial Expression Recognition Challenge*[4] organizada en 2013 y con una retribución de 500 dólares al ganador. Se trata de un conjunto de datos completamente nuevo (no está sacado de ningún conjunto de datos conocido sobre el que se suelen realizar constantemente pruebas)

[4] https://www.kaggle.com/c/challenges-in-representation-learning-facial-expression-recognition-challenge

para evitar que los aspirantes al reto vayan con trabajo preparado bajo el brazo. Consta de imágenes de rostros con un tamaño de 48x48 píxeles en escala de grises. Los rostros están más o menos centrados (fueron extraídos automáticamente) y ocupan más o menos el mismo espacio en cada imagen. Las emociones presentes son siete y están etiquetadas con números del 0 al 6 (siendo 0: enfado, 1: asco, 2: miedo, 3: felicidad, 4: tristeza, 5: sorpresa y 6: neutral). En la fig. 5.1 se pueden observar algunas imágenes extraídas a modo de ejemplo. El número de imágenes para entrenamiento son 28.709 mientras que para test son 7.178. Cabe decir que, originalmente, los datos de test se dividieron en dos durante la competición siendo la primera mitad el conjunto de test para ver tus propios resultados y la segunda mitad el conjunto de test que se emplearía para la clasificación de la competición. Sin embargo, con tal de reducir la variabilidad de los datos y tener una perspectiva más general del funcionamiento de nuestro modelo, nos hemos tomado la libertad de no hacer distinción en ese sentido y emplear todas esas muestras para test. Así pues, el lector ha de tener esto en cuenta cuando se presenten los resultados.

Figura 5.1: Imágenes de ejemplo del conjunto de datos

Una vez tenemos los datos, es común para cualquier aplicación de aprendizaje automático, hacer algún tipo de preproceso de los mismos. Este preproceso busca la reducción del ruido existente (como por ejemplo la existencia de datos atípicos), normalizar los datos para que no tengan valores numéricos muy dispares, etc. En definitiva, busca tener datos significativos pero homogéneos. Con esto en mente, aplicamos como normalización a las imágenes el *z-score*. El *z-score* es una medida numérica de la relación de un valor con la media del conjunto datos. Un *z-score* de 0 significa que ese valor es idéntico a la media, mientras que si es positivo o negativo, el valor se encuentra por arriba o por debajo de la misma respectivamente. La fórmula para calcular el z-score de un valor x es:

$$z = \frac{x - \mu}{\sigma}$$

Donde μ es la media de la población y σ la desviación típica. Desde el punto de vista geométrico, restar a un valor la media lo podríamos ver como centrar los datos en 0,

mientras que dividir por la desviación típica lo podríamos ver como la normalización de los valores para que se encuentren aproximadamente en la misma escala. Así, aplicamos a cada píxel (x,y) de la imagen esta fórmula con la correspondiente media y desviación típica del píxel (x,y) a lo largo de las imágenes del conjunto. Es importante destacar que para que todo el proceso de separación de datos en entrenamiento y test tenga sentido, cualquier valor estadístico sacado de los mismos (como por ejemplo, la media) ha de extraerse sólo del conjunto de datos de entrenamiento y luego aplicarlo al conjunto de test.

5.2.1 Reducción del *overfitting*: *data augmentation* y *dropout*

Uno de los problemas con los que hemos tenido que tratar es la falta de datos. La falta de datos trae consigo resultados indeseados en cualquier problema de aprendizaje automático. Entre ellos se encuentra la rápida aparición de *overfitting*, una oscilante tasa de error entre iteración e iteración y la mala generalización del modelo frente a nuevos datos. Con el algoritmo de descenso por gradiente buscamos entrenar a nuestro modelo en la tarea que tenemos entre manos haciéndole ver constantemente muestras de ejemplo para que los aprenda y pueda así discernir sobre nuevos ejemplos. A medida que el número de *epochs* crece, es decir, a medida que el modelo vuelve a ver una y otra vez los mismos datos muchas veces, este empieza a aprenderlos demasiado bien, ajustando sus parámetros para minimizar el error con esos datos de forma exagerada. Esto causa que no generalice bien sobre datos nuevos. Para la reducción del *overfitting,* hacemos uso de dos técnicas muy conocidas: *data augmentation* y *dropout.*

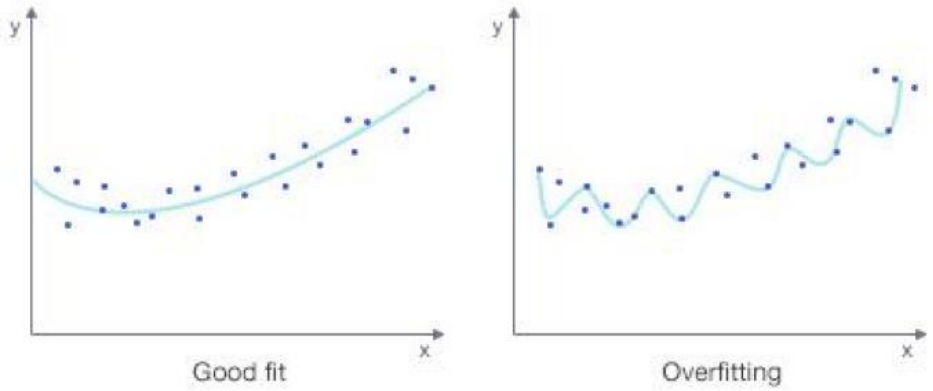

Figura 5.2: Ejemplo simple de *overfitting*. Intentamos clasificar los puntos en 2 clases. La función representada en la izquierda (línea azul) representa de forma adecuada el estado de los datos mientras que en la gráfica de la derecha, la función intenta separar de forma demasiado precisa los puntos existentes, no generalizando bien para nuevos puntos que puedan aparecer (esto es, *overfitting*).

Para poder tener un mayor conjunto de datos y con ello, reducir el *overfitting*, hemos procedido a replicar todas las imágenes del conjunto de entrenamiento espejándolas horizontalmente. Esto se conoce como *data augmentation* y se suele emplear para dotar de robustez al modelo ya que añade nuevos datos relevantes a partir de los que ya se tiene de forma "gratuita". Otros tipos de *data augmentation* que se suelen emplear es la aplicación de una pequeña rotación aleatoria o la de desplazar las imágenes unos pocos píxeles. En el caso de imágenes más grandes también se suelen coger varios segmentos de una imagen y emplearlos como muestras válidas. En nuestro caso quisimos aplicar la rotación aleatoria de las imágenes pero dado que esto se hacía al vuelo (a medida que ibas leyendo datos de disco), y la librería que lo hacía trataba las matrices que representan los datos como imágenes, al realizar la rotación nos eliminaba por completo el z-score aplicado ya que los valores resultado después de la rotación estaban en el rango entre 0 y 255.

Otra técnica muy novedosa y efectiva que surgió hace un par de años es el *dropout*[20]. La combinación de varios modelos para la clasificación de muestras casi siempre mejora la precisión de las predicciones realizadas con modelos individuales. Sin embargo, en redes neuronales grandes esto es inviable debido, en primer lugar, a la enorme cantidad de cómputo necesario y, en segundo lugar, a que puede darse que no existan los suficientes datos como para entrenar dos o más redes distintas. *Dropout* soluciona el problema del *overfitting* y provee una forma de aproximar la combinación de un número exponencial de redes neuronales de forma eficiente. La técnica consiste en eliminar neuronas de la red con cierta probabilidad. Cada neurona tiene una probabilidad p de ser eliminada junto con todas sus conexiones entrantes y salientes durante el entrenamiento. Esto se consigue multiplicando cada capa de neuronas por una máscara de 1 y 0. Los 0 anulan la participación de esa neurona y sus conexiones en la red. Para cada muestra se coge de forma aleatoria (siguiendo la probabilidad asignada a las neuronas) una máscara para cada capa y se entrena a la red resultante (aquella cuyas neuronas han sobrevivido al *dropout)*.

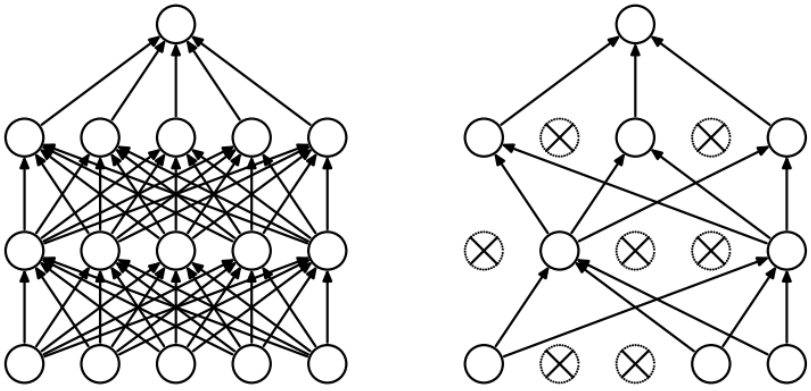

Figura 5.3: Ejemplo de aplicación de *dropout*. **Izquierda**: red neuronal convencional con todas sus conexiones. **Derecha:** red neuronal resultante después de aplicar *dropout*.

Fuente: *Dropout: A Simple Way to Prevent Neural Networks from Overfitting, Nitish Srivastava (2014)*

Podemos ver esto como entrenar distintas redes neuronales más finas. Una red con n neuronas puede ser vista como un conjunto de 2^n redes más finas. Sin embargo, al compartir los pesos, la cantidad de recursos no aumenta de forma exponencial. Ahora, para combinar estas posibles redes simplemente no se aplica *dropout* durante la fase de test y se multiplica los pesos por la probabilidad que se le ha asignado a las neuronas.

En [20] se afirma que un valor de 0.5 para p (probabilidad de *dropout* de una neurona) suele ser el valor óptimo (para neuronas de capas ocultas) en gran variedad de problemas y es el que hemos empleado en nuestro caso. En la fig. 5.4 se puede ver lo que pasa cuando se usa *dropout* y cuando no. Como podemos ver el descenso del error que se produce en la fase de entrenamiento tarda más en aparecer (aparición de *overfitting*) en el caso del uso de *dropout*.

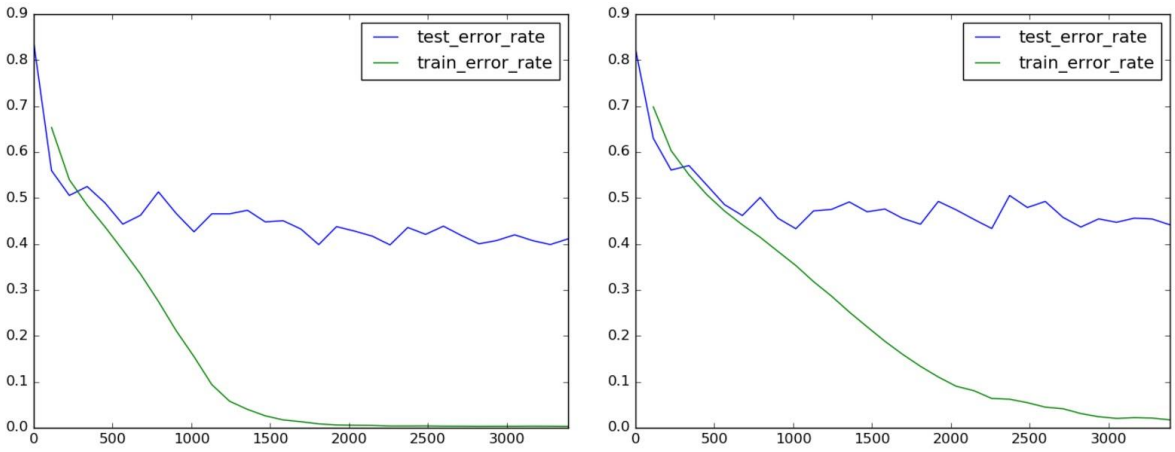

Figura 5.4: Representación gráfica del error del conjunto de test y entrenamiento durante el entrenamiento. El uso del dropout se observa en la apariencia del error de entrenamiento (línea verde). **Izquierda:** Entrenamiento sin *dropout*. **Derecha**: Entrenamiento con *dropout*

5.3 Realización de los experimentos

La realización de experimentos para determinar cuáles son los parámetros óptimos del modelo que hacen que este funcione mejor para los datos que disponemos ha sido, junto a la parte de puesta en marcha del proyecto, la parte que más tiempo ha requerido. El proceso seguido para la realización de los experimentos y elección de qué configuraciones de parámetros/arquitectura probar (puesto que el número de configuraciones distintas es potencialmente infinito) se ha basado en la mayor parte,

en resultados de experimentos previos. Si se observa que con el uso de redes más profundas o con mayor número de filtros los resultados mejoran, seguimos por ese camino a ver a dónde nos lleva. Si el uso de otro tipo de actualización de parámetros en el descenso por gradiente nos da mejores resultados o resultados más fiables, probamos a modificar los distintos parámetros que tiene para ver hasta donde se puede llegar con este nuevo método.

En las siguientes subsecciones se explicarán los dos aspectos que se han evaluado durante la realización de las pruebas. Estos son: los distintos tipos de actualización de parámetros y la profundidad de la red. En la siguiente sección se explicarán los resultados obtenidos con cada uno.

5.3.1 Actualización de parámetros

Una vez se ha computado el gradiente con el algoritmo *backpropagation*, este se usa para la actualización de los parámetros. Hay varias formas de realizar esta actualización. A continuación se describirán los empleados.

5.3.1.1 Básico

Esta es la forma más simple de actualización de parámetros. Siendo "x" el vector de parámetros y "dx" el gradiente, la actualización de los parámetros se hace de la siguiente forma:

```
x += - factor_de_aprendizaje * dx
```

El gradiente indica, como ya hemos explicado previamente, como aumenta la función de error con una ligera modificación de un parámetro. Como tenemos como objetivo reducir el resultado de esta función de error, actualizamos los parámetros de forma negativa (de ahí el signo menos). La variable factor_de_aprendizaje es un híper parámetro que elegimos nosotros. Típicamente es una constante pequeña (menor que 1) por la que se multiplica el gradiente para reducir el impacto de valores muy grandes que hagan oscilar el error cometido por actualizar un parámetro de forma excesiva.

5.3.1.2 Básico + Momentum

La actualización con *momentum* es una forma alternativa de actualizar los parámetros que podríamos ver desde una perspectiva física. Trata a la función de error como si fuera una colina que los parámetros tienen que descender. A medida que estos parámetros descienden la colina (para reducir el error) ganan velocidad. Por tanto la velocidad que tendrá en un punto determinado viene dada por la velocidad que tenía

anteriormente en adición al gradiente en ese punto. La forma de actualizar los parámetros tiene la siguiente forma:

```
v = mu * v - factor_de_aprendizaje * dx
x += v
```

Podemos ver que en este caso se emplea como parámetro adicional mu. A este parámetro se le conoce como *momentum* y tiene la función de actuar, visto desde una perspectiva física, como el coeficiente de rozamiento de la superficie. Los valores típicos de mu son [0.9, 0.95, 0.99].

5.3.1.3 Adam

Adam [11] es otro tipo de actualización de parámetros, algo más complejo, que intenta adaptar el factor de aprendizaje a cada parámetro, en lugar de utilizar un factor de aprendizaje global para todos ellos. La forma de actualizar los parámetros tiene la siguiente forma:

```
m = beta1 * m + (1-beta1) * dx
v = beta2 * v + (1-beta2) * (dx**2)
x += - factor_de_aprendizaje * m / (numpy.sqrt(v) + epsilon)
```

Notar que este método requiere de más híper parámetros (beta1, beta2, épsilon) los cuales tienen como valores recomendados, según los autores, 0.9 para beta1, 0.999 para beta2 y 1e-8 para épsilon.

5.3.2 Profundidad de la red

La profundidad de la red es otro de los aspectos que hemos probado. Por profundidad de la red nos referimos al número de capas convolucionales empleadas. Para cada red de una misma profundidad hemos probado varias configuraciones de número de filtros por capa convolucional y distintos tamaños de convolución. El número de filtros por capa convolucional que hemos probado han sido 16, 32, 64 y 128, mientras que tamaños de convolución hemos probado 3, 5 y 7. Tamaños de filtro más grandes con imágenes tan pequeñas como las nuestras hacía que rápidamente desapareciera la representación y no ofrecía resultados concluyentes.

Durante los primeros experimentos comenzamos con redes poco profundas (2 capas convolucionales). Por mucho que modificamos parámetros como el tamaño de convolución, el número de filtros o el factor de aprendizaje, todas las configuraciones tendían a estabilizarse en cierto punto, por lo que pensamos que el factor limitante era la profundidad de la red. A partir de ahí, pasamos a experimentar con redes de 3,4 y 5 capas para ver qué resultados ofrecían.

En la siguiente sección se hace un estudio detallado de los resultados obtenidos fruto de probar distintas configuraciones y como el hecho de aumentar la profundidad de la red ha influido o no en el rendimiento del modelo.

5.4 Desarrollo de los experimentos

Comenzamos los experimentos de forma simple, con la actualización de parámetros básica y con una red pequeña para tener una primera toma de contacto con el problema. Probamos la configuración de red de la fig. 5.5:

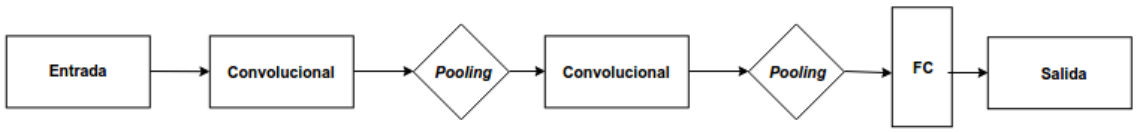

Figura 5.5: Arquitectura convolucional simple. Primeros experimentos.

Con esta configuración de red, variando el número de filtros de las capas convolucionales, tamaño de convolución y número de neuronas de la capa *fully-connected* obtenemos un error de aproximadamente el 50%. Sin embargo, en estas primeras pruebas nos encontramos con dos problemas. El primero, poner el factor de aprendizaje muy alto (0.1) causaba inestabilidad en algunas configuraciones y hacía que la función de coste se fuera a infinito, cosa que paraba completamente el aprendizaje de la red. En la fig. 5.6 se puede ver los efectos en el error cometido cuando esto pasa. Calibrando el factor de aprendizaje con valores menores que 0.1 (0.05, 0.03) este fenómeno prácticamente desaparece.

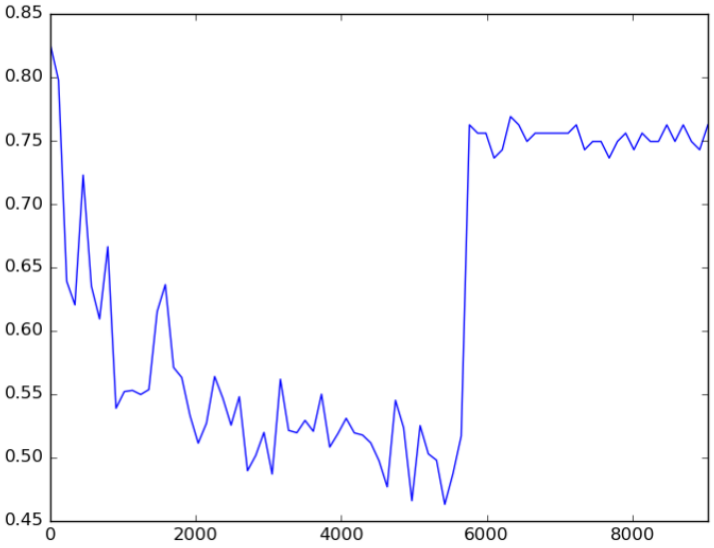

Figura 5.6: Gráfico del error de test durante el entrenamiento. En cierto punto entre la iteración 4000 y 6000 el error se dispara y el modelo ya no aprende.

El segundo problema que nos encontramos era la gran oscilación del error de *epoch* a *epoch*. En algunos casos, la función presentaba hasta un 10% de error de diferencia (hacia arriba o hacia abajo). En la fig. 5.7 queda ilustrado este fenómeno. Esto puede deberse por diversas razones. La primera y más intuitiva es que el factor de aprendizaje es muy alto y eso hace que los pesos se actualicen de forma excesiva, causando esa oscilación en la optimización. Para comprobar si este era el problema redujimos el factor de aprendizaje varios órdenes de magnitud. Sin embargo, pese a reducir la oscilación en cierta medida los resultados eran similares a los obtenidos en primera instancia.

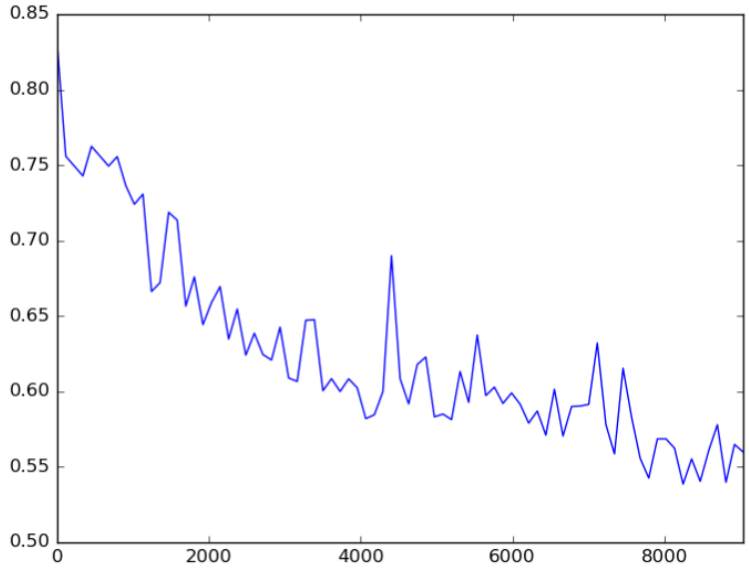

Figura 5.7: Gráfico del error cometido con el uso de la actualización de parámetros básico con un factor de aprendizaje de 0.01.

La segunda posible razón, y por la que nos decantamos, es que existe una falta de datos de test que hace que presenten una gran variabilidad. Lo podemos ver con el siguiente ejemplo. Si tengo 3 muestras de test y clasifico 2 mal y una bien en una iteración, hago una iteración del algoritmo de descenso por gradiente y clasifico 1 muestra mal y 2 bien, el error ahora es del 33% mientras que antes era del 66%. Luego puede darse el caso que vuelva a clasificar 2 mal o incluso 3, modificando en gran medida el error obtenido y no dando resultados concluyentes en relación a si se está mejorando o no simplemente por una falta de datos; es un ejemplo muy exagerado pero ilustra el problema que enfrentamos con la variabilidad de los datos.

Debido a que no podíamos evaluar las pruebas con este tipo de actualización de parámetros ya que la oscilación era tan grande que, los resultados obtenidos de una red a otra eran difíciles de comparar, decidimos usar otro tipo de actualización con parámetros con la esperanza de obtener mejores resultados.

Con ello, pasamos a usar la actualización de parámetros con *momentum*. Para nuestra sorpresa, nos funcionó realmente bien en comparación a la actualización de parámetros básica. En este caso nos eliminó en gran medida esa oscilación de la que hablábamos antes y además, el error promedio que obteníamos realizando las pruebas se redujo a entre 40-45%. Con este descubrimiento, decidimos darle una oportunidad a la actualización de parámetros Adam ya que era algo más sofisticado. Para nuestra decepción, Adam se comportaba de forma similar a *momentum* (nos eliminaba la oscilación que teníamos al principio) pero el error promedio que obteníamos estaba

alrededor del 50-55%. Visto que la actualización con *momentum* era la que mejor nos funcionaba decidimos emplearla en el resto de experimentos.

Una vez elegida la actualización de parámetros pertinente sólo faltaba probar con distintas arquitecturas. Las pruebas realizadas hasta ese momento parecían no querer pasar de ese 40% de error, por lo que nos pusimos en contacto con Maxim Milakov, investigador en aprendizaje automático (el cual logró obtener un 32% de error en esta misma competición, otorgándole el tercer puesto) para que nos diera algunos consejos en la materia. En su respuesta nos indicó que probáramos con redes más profundas y que cuando nos encontráramos con *overfitting* aplicáramos *dropout* y *data augmentation*. Con ello probamos redes con 3 4 e incluso 5 capas, variando tanto la posición como el número de capas *pooling* para evitar que la transformación de la imagen quedara demasiado pequeña al llegar a la capa *fully-connected.* Sin embargo, ninguna configuración igualaba el mejor resultado que obtuvimos con una red de dos capas (37.4%).

En la siguiente sección se detalla los mejores resultados que obtuvimos con redes de distintas profundidades, indicando su arquitectura y su porcentaje de precisión.

5.4.1 Resultados de los experimentos

En la tabla 5.3 se recogen los resultados más relevantes obtenidos con diferentes arquitecturas, ordenados de mejor a peor.

Red neuronal	Error
Entrada→C(5x5, 64f.)→ POOL→C(3x3,64f.) →POOL→FC(512 n.) → SALIDA	37.4%
Entrada→C(3x3, 32f.)→ POOL→C(3x3,32f.) →POOL→ C(3x3,32f.) →POOL→FC(2048 n.) → SALIDA	39.5%
Entrada→C(5x5, 32f.)→ POOL→C(3x3,32f.) →POOL→FC(256 n.) → SALIDA	39.7%
Entrada→C(3x3, 32f.)→ POOL→C(3x3,32f.) →POOL→FC(2048 n.) → SALIDA	39.8%

Tabla 5.3: Mejores resultados obtenidos con diversas redes

5.5 Visualización de los experimentos

El hecho de trabajar con imágenes nos brinda la ventaja de estudiar el problema desde un punto de vista no sólo numérico (situación que se da en la mayoría de problemas de aprendizaje automático), sino desde un punto de vista visual (mucho más intuitivo). En esta sección se presentan dos aspectos del problema que se han extraído a medida que se realizaban los experimentos.

5.5.1 Zonas más representativas de una imagen

La reciente popularidad del uso de las CNN en problemas de clasificación de imágenes y su particular característica de la no necesidad por parte de los humanos de diseñar y señalar qué características son relevantes para la resolución de un problema (lo hacen automáticamente) ha llevado a los investigadores a encontrar una forma de visualizar cuales son las característica que estas aprenden. Con ello se busca entender dichos modelos en más profundidad ya que no se sabe del todo por qué funcionan tan bien.

Varios trabajos se han llevado a cabo para resolver este problema en [23] y [18]. El primero lo consigue mediante el uso de una red deconvolucional. Una red deconvolucional es similar al modelo convolucional ya que usa los mismos componentes (*pooling* y filtros) pero hace lo contrario. En lugar de extraer de los píxeles las características de una imagen hace el proceso inverso. Ya que esta no es la técnica que hemos empleado pasaremos a describir la siguiente. Para más información sobre el funcionamiento de esta técnica consultar [23].

En el caso de [18] el enfoque es algo más simple. Se trata de asignar un valor de importancia a los distintos píxeles en función de una clase y el valor que ha asignado la CNN al clasificar esa imagen en esa clase. Tomaremos prestada la forma de explicarlo en [18] para explicarlo de la misma forma aquí y tomar una intuición del método empleado. Nuestro problema pues, consiste en dar a los píxeles de una imagen I un valor de importancia en función de su influencia en el cálculo de la puntuación asignada a una clase c ($S_c(I)$). Para ello tomemos primero un ejemplo más simple. Considerar esta función para la clasificación en la clase c:

$$S_c(I) = W_c{}^T I + b_c$$

Donde la imagen I se representa en forma de vector y, w y b son respectivamente el vector de pesos y el *bias* del modelo. No es difícil observar, que la importancia de un píxel vendrá dado por la magnitud que tenga su elemento del vector de pesos asociado. En el caso de las CNN la función $S_c(I)$ es algo más complicada ya que implica varias funciones no lineales, por lo que el razonamiento de arriba no se puede aplicar de inmediato. Sin embargo, se puede aproximar esta función con una función lineal aplicando la expansión de Taylor de primer orden:

$$S_c(I) \approx W^T + b$$

Donde W^T es la derivada de S_c con respecto a la imagen I en el punto (más bien, imagen), I_0:

$$w = \left. \frac{\partial S_c}{\partial I} \right|_{I_0}$$

El valor de las derivadas de cada píxel nos indica los píxeles que, cambiando muy poco su valor afectarían en gran medida al valor de la función de clasificación de la clase c. Con ello obtenemos un "mapa" del tamaño de la imagen de entrada que nos indica la importancia de ese píxel en la imagen. En la fig. 5.8 se puede ver algunos ejemplos extraídos de [18].

Figura 5.8: Mapas de importancia obtenidos de diversas imágenes con una red convolucional entrenada. Fuente: *Deep inside convolutional networks: Visualising image classification models and saliency maps (2014)*

En nuestro caso lo hicimos algo diferente ya que nuestras imágenes eran muy pequeñas. En lugar de tener el mapa a un lado y la imagen a otro decidimos superponer el mapa en el canal rojo de la imagen para ver con más claridad que partes de la imagen son los que realmente son relevantes. En la fig. 5.9 podemos ver varios ejemplos de los resultados:

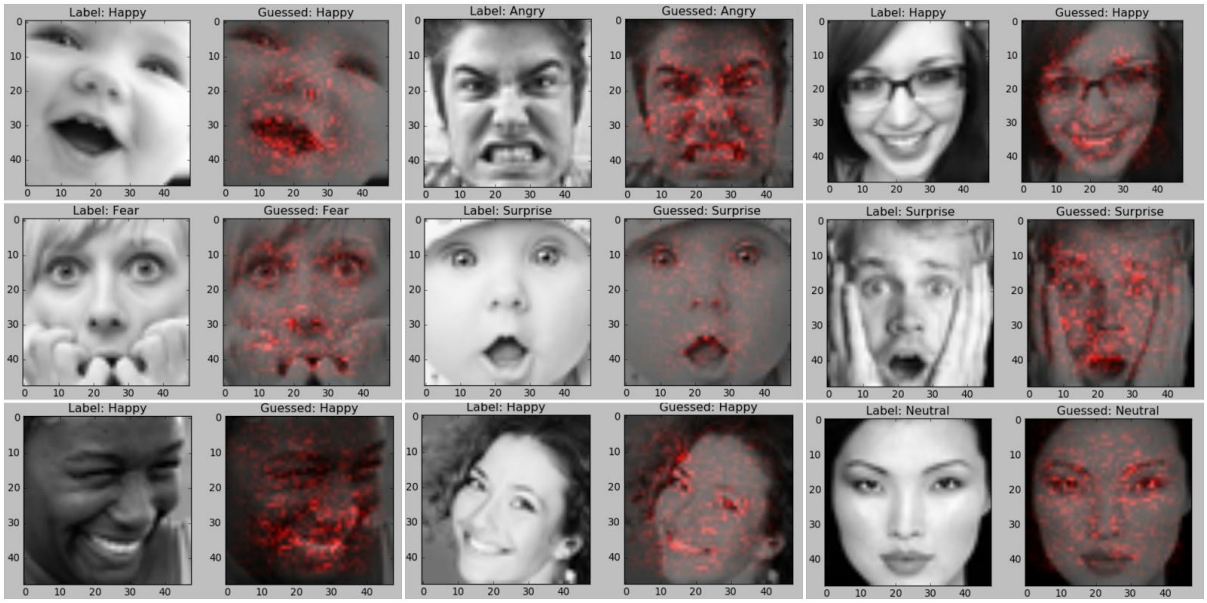

Figura 5.9: Mapas de importancia superpuestos a los rostros

Como podemos observar en la figura 5.9, en emociones como la felicidad, los píxeles más relevantes se encuentran concentrados en la zona de la boca y los ojos mientras que en emociones como el enfado o la sorpresa, los píxeles se distribuyen más uniformemente por el rostro. Podemos hacer la comparativa nosotros mismos cuando tratamos con este problema como seres humanos. Una sonrisa la asociamos fácilmente con el estado de felicidad, mientras que para detectar una expresión de enfado o neutralidad hemos de fijarnos en otros aspectos como el ceño, las mejillas, la apertura de los ojos y la boca, etc.

Viendo estos resultados, no es difícil pensar que la red está realmente aprendiendo características relevantes de los rostros sin tener nosotros que intervenir en el proceso (es decir, indicarle que cosas tener en cuenta y que no) y que no se guía por cosas irrelevantes como el pelo (en las imágenes apenas existen píxeles coloreados en las zonas del cabello). Sin embargo, también observamos que, existen emociones como la sorpresa o el enfado que exhiben un patrón similar en este aspecto. Esto nos hace preguntarnos si nuestro modelo confunde algunas emociones con otras o si le es más fácil distinguir unas emociones más que otras. Todas estas cuestiones serán tratadas en la siguiente sección.

5.5.2 Entendiendo los errores

Observando los resultados obtenidos en la sección anterior y observando de primera mano qué imágenes clasificaba bien y cuáles no, nos dimos cuenta de que tendía a clasificar mejor las que representaban felicidad. Observamos también, ciertos patrones en la clasificación de algunas emociones, las cuales, si las fallaba, siempre solía

confundirlas con otra emoción semejante. Esto nos llevó a querer comprobar este hecho para ver si era real o era simplemente impresión nuestra. La tabla 5.4 recoge los resultados obtenidos con una red entrenada.

Etiqueta de la muestra	Clasificación de la emoción realizada por el modelo						
	Enfadado	Asco	Miedo	Feliz	Triste	Sorpresa	Neutral
Enfadado	14.93%	1.88%	24.74%	23.28%	13.26%	1.88%	20.04%
Asco	8.11%	21.62%	21.62%	15.32%	14.41%	2.7%	16.22%
Miedo	2.34%	0.49%	40.04%	16.80%	15.14%	5.08%	20.12%
Feliz	1.07%	0.34%	4.85%	81.06%	4.23%	0.56%	7.89%
Triste	3.69%	0.72%	24.22%	18.44%	30.07%	1.28%	21.57%
Sorpresa	0.96%	0.24%	25.99%	13.72%	6.98%	39.23%	12.88%
Neutral	2.51%	0.32%	16.30%	15.90%	14.92%	1.05%	48.99%

Tabla 5.4: Matriz de confusión en la clasificación de emociones de un clasificador entrenado.

Emoción	Enfadado	Asco	Miedo	Feliz	Triste	Sorpresa	Neutral
Nº de muestras	958	111	1024	1774	1247	831	1233

Tabla 5.5: Número de muestras de cada clase

Como podemos observar, las diferencias a la hora de clasificar distintas emociones son significativas. Salta a la vista la gran precisión que presenta el modelo al clasificar correctamente a las imágenes que representan la emoción de felicidad. Si nos paramos a pensar detenidamente, este hecho tiene sentido ya que esta emoción es la que más se diferencia del resto. Normalmente se da con los labios separados, enseñando los dientes, con los ojos entrecerrados y las mejillas elevadas. Esta serie de características son muy distintivas y hacen que catalogarlo por error en alguna otra emoción sea algo poco probable. La segunda emoción que más acierta es curiosamente la neutral, y decimos curiosamente porque no posee ninguna característica distintiva que haga decantarse por esa elección. Tal vez es por eso mismo que el hecho de no encontrar características significativas en una imagen por una razón u otra haga que el modelo se decante por esta emoción.

Conforme vamos mirando el resto de emociones la precisión va bajando pero es entonces cuando podemos ver más claramente qué está pasando. En el caso de la sorpresa por ejemplo, podemos ver como la confunde muchas veces con el miedo. Los rostros que representan ambas emociones suelen tener la boca abierta (en el caso del miedo, si está gritando) acompañado con un levantamiento de cejas, apertura de los ojos y la aparición de arrugas en la frente. Como podemos ver comparten bastantes características faciales haciendo esto probable que se produzcan errores. En el caso del asco vemos como no sabe distinguir entre asco y miedo principalmente. Aunque los porcentajes obtenidos para esta emoción no los podemos tomar en serio desde un punto de vista estadístico debido a la gran disparidad de número de muestras que existe con el resto de emociones (estamos hablando aproximadamente de 9 veces más muestras de media para el resto de emociones), nos deja claro que clasificar correctamente esta emoción no es tarea fácil. Las fallas del modelo se hacen más visibles en el caso del enfado el cual se consigue clasificar más veces como miedo y felicidad (dos emociones muy dispares).

A la vista de estos resultados nos vemos en la obligación de consultar en la literatura si esto sucede con frecuencia o es un caso aislado debido a la falta de muestras de nuestro conjunto de datos. Efectivamente, encontramos que se han llegado a conclusiones similares [1] estudiando las matrices de confusión (como la presente en la tabla 5.4). En [1] también se comenta que un estudio realizado por los psicólogos Ekman y Friesen [17], muestra que los humanos tendemos a confundir también emociones como enfado con asco, o miedo con sorpresa. También se expone que de las 6 expresiones típicas (eliminan en este caso la emoción neutral), las más fáciles de reconocer son sorpresa y felicidad.

A pesar de las conclusiones que podamos sacar de estos resultados hay que recordar que estamos tratando con un modelo que tiene una moderada tasa de acierto y que tanto el número de muestras de entrenamiento como de test es bastante pequeña (unas pocas miles de muestras) en comparación con los estudios realizados por los investigadores en este campo (con millones de muestras de entrenamiento y test). Así pues, tanto los números obtenidos como las conclusiones hay que tomarlas con cierta cautela.

CAPÍTULO 6

Conclusiones

Cerramos la memoria de este proyecto con el capítulo de conclusiones. En este último capítulo se expondrán los principales problemas que hemos encontrado en la realización del proyecto. También se añaden las consideraciones finales obtenidas de la realización de este trabajo junto con un resumen final analizando los resultados y la relevancia del trabajo desarrollado.

6.1 Problemas encontrados

Durante la realización del proyecto hemos sufrido contratiempos inesperados que han frenado el desarrollo del mismo. A continuación se hace un breve resumen de los principales problemas con los que nos hemos enfrentado y cómo se han solucionado:

- **Puesta en marcha del entorno de trabajo:** Para la realización de este proyecto ha sido necesario el uso de gran cantidad de librerías para el lenguaje de programación Python. El uso de librerías como Blocks y Theano que están actualmente en continuo desarrollo ha hecho que la integración de dichas herramientas en el sistema haya sido uno de los mayores quebraderos de cabeza del proyecto. Y no solo eso, las librerías de Python que ambas herramientas poseen en común a veces ofrecían problemas de compatibilidad entre ellas por no usar la misma versión o usar una versión más reciente o más antigua. Para la solución de este problema se ha confiado mucho en el uso de los entornos virtuales de Python con tal de aislar en un mismo sitio las librerías a las que accedía el lenguaje y tener así, un mayor control de las versiones de los paquetes que habían instalados. Además, puesto que hemos hecho uso de la ejecución de operaciones en la GPU, software adicional ha sido necesario instalar en el sistema junto con su respectiva integración con las herramientas antes expuestas.

- **Falta de documentación de Blocks:** Pese a haber sido una herramienta tremendamente útil para la realización de este trabajo, la documentación de la misma ha dejado mucho que desear. La que había muchas veces estaba desactualizada o no ofrecía la información que se buscaba. Esto ha llevado la mayoría de las veces a ir al código fuente de la herramienta para ver cómo se

desarrollaban los acontecimientos bajo el capó. En casos extremos, nos hemos visto en la posición de contactar con los desarrolladores para la resolución de dudas.

- **Curva de aprendizaje de Theano:** Blocks trabaja con Theano a un nivel de abstracción bastante alto. Esto ofrece una forma simple y fácil de hacer las cosas cotidianas. Sin embargo, cuando quieres hacer cosas específicas que se salen de la norma, como por ejemplo, los mapas de importancia de los píxeles de una imagen, no hay más opción que trabajar con Theano directamente. Entender la forma en la que trabaja Theano, cómo se constituye el grafo computacional y aprender a utilizar las expresiones simbólicas para la evaluación de operaciones con variables simbólicas han sido los principales retos que ha presentado esta herramienta. Gracias a la gran cantidad de documentación existente en la red, se ha superado este problema de la forma menos dolorosa posible.

6.2 Consideraciones finales

Mediante la realización de este proyecto nos hemos podido meter de lleno en el terreno del aprendizaje automático y en el uso de herramientas profesionales que explotan esta área para el planteamiento y solución de un problema de gran relevancia en nuestros días: el reconocimiento de emociones. Con la investigación llevada a cabo para la realización de este proyecto hemos podido descubrir que la resolución de este problema es algo que no solo encontramos en el terreno de investigación sino que, además, es explotado industrialmente. También hemos podido ver que cuando se trata de resolver estos problemas, el hallazgo de mejores resultados viene ligado muchas veces a la contrastación del trabajo realizado por otros en este tema y a la prueba y error de nuevos métodos, técnicas y arquitecturas.

En cuanto al trabajo desarrollado, destacamos la infraestructura construida alrededor del modelo de aprendizaje automático. En concreto el *script* parametrizado, que nos permite lanzar múltiples ejecuciones con distintas arquitecturas, métodos de actualización de parámetros y parámetros propios de la red neuronal. También destacar la funcionalidad desarrollada para el análisis y evaluación de los resultados, incluyendo tanto las extensiones desarrolladas para Blocks para guardar en un archivo estructurado los resultados del aprendizaje como la serie de *scripts* que leen estos archivos para mostrar de forma gráfica e intuitiva los cientos de resultados numéricos almacenados.

Los resultados obtenidos fruto de optimizar al máximo el modelo (62.3% de tasa de acierto) se acercan en gran medida al realizado por el ser humano en este mismo conjunto de datos (alrededor del 65% de tasa de acierto según se muestra en [9]), hecho que nos dejaría clasificado dentro del top 10 de la competición que Kaggle llevó a cabo. Sabiendo que solo los 4 primeros de la competición consiguieron mejorar el resultado humano por un leve margen (hasta un 4% mejor) y que algunos de estos desarrollos se hicieron en equipos (entre los que se encuentran integrantes de la prestigiosa universidad de Toronto, universidad puntera en tareas de aprendizaje automático), los resultados son satisfactorios.

Como trabajo para el futuro nos gustaría continuar con la resolución de este problema trabajando en la combinación de varios modelos en un proceso de votación con el fin de mejorar los resultados. También nos gustaría cambiar de conjunto de datos por otro más grande con el fin de obtener resultados más sólidos y fiables ya que con los que hemos trabajado ofrecían mucha variabilidad en los resultados.

Bibliografía

[1] Bettadapura, V. (2012). Face Expression Recognition and Analysis: The State of the Art. *Journal of CoRR*, abs/1203.6722.

[2] Chen,L. S. , Tao, H., Huang, T. S., Miyasato, T., & Nakatsu, R. (1998). Emotion recognition from audiovisual information. In *Proc. IEEE Workshop on Multimedia Signal Processing*, 83–88.

[3] Cohen, I., Garg, A. & Huang, T. S. (2000). Emotion recognition from facial expressions using multilevel HMM. *Neural Inf. Process. Syst. (NIPS) Workshop Affective Comput., Colorado*.

[4] Cohen, I., Sebe, N., Garg, A., Chen, L. S., & Huang, T. S. (2003). Facial expression recognition from video sequences: Temporal and static modeling. *Computer Vision and Image Understanding, 91*(1-2), 160-187. doi:10.1016/s1077-3142(03)00081-x

[5] Dellaert, F., Polzin, T., & Waibel,A. (1996). Recognizing emotion in speech. *in Proc. ICSLP*, 1970–1973.

[6] Deng, L., & Yu, D. (2014). Deep Learning: Methods and Applications. *FNT in Signal Processing Foundations and Trends® in Signal Processing, 7*(3-4), 197-387. doi:10.1561/2000000039

[7] Ekman, P. & Friesen,W.V., (1977). Manual for the Facial ActionCoding System. *Consulting Psychologists Press*.

[8] Essa, I., & Pentland, A., (1995). Coding, analysis, interpretation and recognition of facial expressions, *Tech. Rep.* 325.

[9] Goodfellow, I. J., Erhan, D., Carrier, P. L., Courville, A., Mirza, M., Hamner, B., . . . Bengio, Y. (2013). Challenges in Representation Learning: A Report on Three Machine Learning Contests. *Neural Information Processing Lecture Notes in Computer Science,* 117-124. doi:10.1007/978-3-642-42051-1_16

[10] He, K., Zhang, X., Ren, S. & Sun, J. (2015). Deep residual learning for image recognition. arXiv preprint arXiv:1512.03385

[11] Kingma, D. & Ba, J. (2014). Adam: A method for stochastic optimization. arXiv preprint arXiv:1412.6980

[12] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks.
[13] Krizhevsky, A. (2009). Learning multiple layers of features from tiny images. Master's thesis, Department of Computer Science, University of Toronto.

[14] Lecun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE Proc. IEEE, 86*(11), 2278-2324. doi:10.1109/5.726791

[15] Merriënboer, B., Bahdanau, D., Dumoulin, V., Serdyuk, D., Warde-Farley, D., Chorowski, J., & Bengio, Y. (2015). Blocks and Fuel: Frameworks for deep learning, *arXiv preprint arXiv:1506.00619 [cs.LG]*.

[16] Rosenblum, M., Yacoob, Y., & Davis, L. (n.d.). Human emotion recognition from motion using a radial basis function network architecture. *Proceedings of 1994 IEEE Workshop on Motion of Non-rigid and Articulated Objects*. doi:10.1109/mnrao.1994.346256

[17] Sebe, N., Cohen, I., Garg, A. & Huang, T.S., (2005). *Machine Learning in Computer Vision*. New York, Springer-Verlag.

[18] Simonyan, K., Vedaldi, A. & Zisserman, A. (2014). Deep inside convolutional networks: Visualising image classification models and saliency maps. In *Proc. ICLR*.

[19] Simonyan, K., & Zisserman, A. (2015). Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556.

[20] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. & Salakhutdinov, R. (2014) Dropout: A simple way to prevent neural networks from overfitting. *JMLR*.

[21] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., . . . Rabinovich, A. (2015). Going deeper with convolutions. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. doi:10.1109/cvpr.2015.7298594

[22] Tao, H. & Huang, T.S., (2002). A Piecewise Bezier Volume Deformation Model and Its Applications in Facial Motion Capture, in *Advances in Image Processing and Understanding: A Festschrift*.

[23] Zeiler, M. D., & Fergus, R. (2014). Visualizing and Understanding Convolutional Networks. *Computer Vision – ECCV 2014 Lecture Notes in Computer Science,* 818-833. doi:10.1007/978-3-319-10590-1_53