

# Universidad de Alcalá

# Escuela Politécnica Superior

## Grado en Ingeniería Informática

### Trabajo Fin de Grado

Estudio de técnicas de visión e inteligencia artificial aplicadas a un  
caso práctico

ESCUELA POLITÉCNICA  
**Autor:** Pablo García García  
**Tutor:** Adrián Domínguez Díaz

2024



**UNIVERSIDAD DE ALCALÁ  
ESCUELA POLITÉCNICA SUPERIOR**

**Grado en Ingeniería Informática**

**Trabajo Fin de grado**

**Estudio de técnicas de visión e inteligencia artificial aplicadas a  
un caso práctico**

Autor: Pablo García García

Tutor: Adrián Domínguez Díaz

**Tribunal:**

**Presidente:**

**Vocal 1º:**

**Vocal 2º:**

Fecha de depósito:



*“Computer Science is no more about computers than astronomy is about telescopes.”*  
Edsger W. Dijkstra



# Resumen

**Palabras clave:**



# Abstract

**Keywords:**



# Índice general

<b>Resumen</b>	<b>7</b>
<b>Abstract</b>	<b>9</b>
<b>Introducción</b>	<b>16</b>
<b>1. Fundamentos de la Inteligencia Artificial y sus herramientas</b>	<b>18</b>
1.1. Breve historia de la Inteligencia Artificial . . . . .	18
1.2. Tipos de aprendizaje y problemas . . . . .	19
1.3. Convolución y correlación cruzada . . . . .	19
<b>2. Aprendizaje automático y profundo</b>	<b>24</b>
2.1. Perceptrón . . . . .	24
2.2. Redes neuronales artificiales . . . . .	30
2.2.1. Funciones de activación . . . . .	34
2.3. Redes neuronales convolucionales . . . . .	37
2.4. Transformers . . . . .	39
2.4.1. Codificación . . . . .	40
2.4.2. Decodificación . . . . .	42
2.4.3. Ejecución y entrenamiento . . . . .	43
<b>3. Optimización del proceso de valoración de puntos de interés</b>	<b>45</b>
3.1. Tecnologías empleadas . . . . .	45
3.1.1. Python . . . . .	45
3.1.2. CUDA . . . . .	47
3.2. Clasificación de imágenes mediante redes convolucionales . . . . .	49
3.2.1. Proceso ETL . . . . .	50
3.2.2. Creación y entrenamiento de una red convolucional en TensorFlow . . . . .	52
3.2.3. Transfer learning en TensorFlow . . . . .	53
3.2.4. Aumento de datos . . . . .	56
3.2.5. Evaluación de los modelos . . . . .	57
3.2.6. Detección de clases desconocidas . . . . .	62
<b>Bibliografía</b>	<b>66</b>



# Índice de figuras

1.1. Detección de bordes aplicando los kernel Sobel . . . . .	21
2.1. Arquitectura de un perceptrón . . . . .	24
2.2. Puntos etiquetados en $\mathbb{R}^2$ . . . . .	25
2.3. Puntos separados en $\mathbb{R}^2$ . . . . .	26
2.4. Valores de $x \oplus y$ en $\mathbb{R}^2$ . . . . .	26
2.5. Arquitectura de una red neuronal multicapa . . . . .	30
2.6. Red neuronal multicapa . . . . .	31
2.7. Descenso por gradiente . . . . .	33
2.8. Función lineal . . . . .	34
2.9. Función logística . . . . .	35
2.10. Tangente hiperbólica . . . . .	35
2.11. Función ReLU . . . . .	36
2.12. Imágenes y sus mapas de características[16] . . . . .	37
2.13. Red convolucional VGG-16 . . . . .	39
2.14. Arquitectura de un transformer codificador-decodificador . . . . .	40
3.1. Información de una propuesta en Wayfarer . . . . .	46
3.2. Cuestionario de una propuesta en Wayfarer . . . . .	46
3.3. Mapa y clasificación de una propuesta de Wayfarer . . . . .	46
3.4. Python . . . . .	47
3.5. Bibliotecas populares de Python . . . . .	47
3.6. NVIDIA . . . . .	48
3.7. Arquitectura CUDA a nivel de hardware y software[22] . . . . .	48
3.8. Comparativa de entrenamiento en CPU frente a en GPU . . . . .	49
3.9. Árbol de carpetas del dataset . . . . .	50
3.10. Visualización de ejemplo de un dataset creado . . . . .	51
3.11. Arquitectura de la red convolucional . . . . .	52
3.12. Pérdida y precisión durante el entrenamiento de la CNN . . . . .	54
3.13. MobileNetV2 . . . . .	55
3.14. Pérdida y precisión durante el entrenamiento aplicando transfer learning . . . . .	56
3.15. Visualización de ejemplo de un dataset aplicando aumento de datos . . . . .	57
3.16. Matrices de confusión . . . . .	58
3.17. Precisión, sensibilidad, y $F_1$ . . . . .	60
3.18. Curvas ROC . . . . .	61
3.19. Capacidad de generalización de la red, predicción en Galicia y Guadalajara . . . . .	62
3.20. Métricas en clasificación con clase desconocida . . . . .	63
3.21. Arquitectura de clasificación con clases desconocidas . . . . .	63
3.22. Diagrama de probabilidades . . . . .	64
3.23. Métricas en clasificación con clase desconocida (doble modelo) . . . . .	65



# Índice de algoritmos

2.1.	Regla de aprendizaje del perceptrón . . . . .	27
2.2.	Descenso por gradiente . . . . .	32
2.3.	Retropropagación ( <i>backpropagation</i> ) . . . . .	34
2.4.	Atención simple . . . . .	41
2.5.	Atención múltiple . . . . .	42
2.6.	Atención múltiple-cruzada . . . . .	43

# Introducción



# Capítulo 1

## Fundamentos de la Inteligencia Artificial y sus herramientas

En este capítulo se abordará una breve introducción al campo de la Inteligencia Artificial, comenzando desde su historia a lo largo del tiempo para contextualizar, pasando por entender los fundamentos de lo que busca lograr, y comentando alguna herramienta matemática que será de utilidad y que en ciertos casos causa confusiones.

### 1.1. Breve historia de la Inteligencia Artificial

Ya desde hace varias décadas, se planteaba la posibilidad de que las máquinas fuesen capaces de realizar tareas diferentes a meros cálculos. La persona que realizó dicha afirmación fue la matemática Ada Lovelace, que en 1842 programó el considerado primer algoritmo. No fue hasta bastantes años más tarde en 1956 cuando se celebra la Conferencia de Dartmouth por los hoy considerados padres de la IA, en la que se propone estudiar la Inteligencia Artificial como una ciencia más. En dicha época existían dos corrientes, la simbólica y la conexiónista.

La primera de ellas se ocupaba de resolver problemas de toma de decisiones y de obtención de conclusiones. Fueron populares los algoritmos de búsqueda y los sistemas expertos. Por otro lado, la corriente conexiónista trataba de simular el comportamiento de las neuronas humanas de manera artificial, haciendo que estas neuronas artificiales pudiesen aprender. De aquí surgió el perceptrón que también fue presentado en esta conferencia. Más tarde, entre 1970 y 1980, con el libro de Minsky sobre los perceptrones y sus limitaciones, investigaciones en falso, y bajos recursos, decayó el interés y la investigación de la IA. Sin embargo, esta etapa vacía finaliza con la llegada del algoritmo de retropropagación que permitía entrenar redes neuronales multicapa, trayendo consigo infinidad de nuevos proyectos.

Años después, avanzados los 2000, se empieza a ver lo poderosos que pueden llegar a ser ciertos modelos de IA al vencer a campeones del mundo en juegos, como a Kasparov en el ajedrez o a Lee en Go. Con la llegada de más y mejores recursos y financiación, hacia 2010 se populariza el uso de redes neuronales para trabajar con imágenes, resolver problemas de clasificación, etc[1].

Hace diez años surge uno de los modelos de IA con el que se inicia el paradigma que más popularidad está tomando en la actualidad. Este es el de la IA generativa. En 2014 Ian Goodfellow presenta las redes GAN con las que crear nuevos datos, por ejemplo rostros humanos que no existen en realidad, tal y como se muestra en <https://thispersondoesnotexist.com/>. Gracias a la IA generativa combinada con modelos de lenguajes, han ido surgiendo en los últimos años herramientas muy poderosas como ChatGPT con la que establecer cualquier tipo de conversación, solicitar información o ayuda para resolver cualquier problema; DALL-E o MidJourney a las que solicitar crear una imagen mediante una descripción por texto, o incluso un video con Sora.

## 1.2. Tipos de aprendizaje y problemas

Al intentar resolver un problema relacionado con el aprendizaje automático, normalmente el primer paso es elegir un modelo que se adapte correctamente al dominio y tipo del problema. De manera ingenua, se puede entender como una especie de caja negra a la que dada una serie de entradas devuelve una serie de salidas que dependen de dichas entradas y una serie de operaciones con respecto a un conjunto de parámetros  $\Theta$ . Por tanto, para obtener las salidas deseadas para una serie de entradas, el trabajo es encontrar los parámetros óptimos  $\Theta^*$  que produzcan dichas salidas. Esto se logra mediante un algoritmo de aprendizaje o entrenamiento, siendo el segundo paso elegir uno acorde al modelo. Normalmente se dispone de dos tipos de aprendizaje, **aprendizaje supervisado** y **aprendizaje no supervisado**. Como tercer paso, se debe de medir de alguna manera cómo de bien o mal se está comportando el modelo y el algoritmo, tal y como se estudiará más adelante[2].

En los problemas de aprendizaje supervisados, se dispone de un conjunto de datos o *dataset*, que contiene los valores de salida deseados para diferentes valores de entrada, normalmente recogiendo situaciones del pasado para poder extraer este conocimiento a situaciones del futuro. Los principales problemas que de aprendizaje supervisado son los problemas de clasificación y de regresión. En los **problemas de clasificación**, se dispone de una serie de clases  $C_1, C_2, \dots, C_n$ , y para una serie de valores de entrada  $x_1, x_2, \dots, x_m$ , debe decidirse a qué clase pertenece dicha entrada. Un ejemplo sería decidir si un paciente va a sufrir un cierto tipo de cáncer dada su edad, peso, y otras constantes vitales. Algunos de los modelos más populares para llevar a cabo este tipo de tareas son árboles de decisión, máquinas de soporte vectorial, Naïve Bayes,  $k$ -vecinos, y redes neuronales; siendo estas últimas objeto de estudio en este trabajo. Otro tipo de problema popular a la hora de disponer de datos etiquetados, son los **problemas de regresión**, que se diferencia principalmente de la clasificación en que en este caso, los valores no son clases (valores discretos) sino valores continuos. Para resolver este tipo de problemas se suelen utilizar regresiones lineales y no lineales (exponencial, polinómica, etc). Un ejemplo de un problema de regresión sería predecir las horas que dormirá una persona dada su edad, horas trabajadas en el día, horas de recreo en el día, etc.

Por otro lado, en los problemas de aprendizaje no supervisado no se dispone de los valores de salida esperados para una cierta observación (justo al contrario que en el caso supervisado), pues será trabajo del algoritmo encontrar relaciones y patrones entre los datos proporcionados. En este tipo de aprendizaje también se trata el problema de clasificación, sin embargo, es más común llamarlo **clustering** o **segmentación**, pues a priori no se conoce el número de clases y cuáles son, es el algoritmo el que deberá encontrar relaciones entre los datos para determinar esto. Algoritmos populares para realizar esta tarea son  $k$ -medias (y su variante  $k$ -medianas), clusterización jerárquica aglomerativa, modelos de mixtura gaussianos, y DBSCAN. Un ejemplo sencillo de este problema es detectar las diferentes regiones y objetos representados en una imagen, pues inicialmente no se conoce el número de regiones u objetos, y deben detectarse todas, asignando cada píxel de la imagen a cada una de ellas.

## 1.3. Convolución y correlación cruzada

Una de las operaciones matemáticas más conocidas y que es más usada al trabajar con señales e imágenes es la llamada convolución, denotada por  $*$ , y dadas las funciones  $f(t)$  y  $g(t)$ , su convolución se define de la siguiente manera.

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t - \tau) d\tau$$

En este caso se está asumiendo que el dominio de  $f(\tau)g(t - \tau)$  es  $\mathbb{R}$ , lo que permite integrar sobre todo  $\mathbb{R}$ , de lo contrario, se modifica la definición para integrar solo sobre un intervalo  $[a, b]$ . En general, esta operación crea una nueva función a partir de otras dos, que indica cómo interactúan entre sí, y que permite aplicar filtros a señales e imágenes. Como se acaba de comentar para el caso de las imágenes, se puede tratar con señales que no dependan únicamente de una variable, pues estas se representan como una función de dos variables  $f(u, v)$ . En este caso, la convolución queda definida de la siguiente manera.

$$(f * g)(u, v) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(\xi, \eta)g(u - \xi, v - \eta) d\xi d\eta$$

Partiendo de las primera definición mostrada, se pueden demostrar algunas propiedades útiles que cumple la convolución[3]:

- Comutativa:  $f * g = g * f$
- Asociativa:  $f * (g * h) = (f * g) * h$
- Distributiva:  $f * (g + h) = (f * g) + (f * h)$
- Derivada:  $\frac{d}{dt}(f * g) = \frac{df}{dt} * g = \frac{dg}{dt} * f$
- Relación con las transformadas de Laplace y Fourier:  $\mathcal{L}\{f * g\} = \mathcal{L}\{f\} \cdot \mathcal{L}\{g\}$ , o lo que suele ser más útil,  $f * g = \mathcal{L}^{-1}\{\mathcal{L}\{f\} \cdot \mathcal{L}\{g\}\}$ , de forma que se puede calcular la convolución en tiempo  $\mathcal{O}(n \log(n))$  con el algoritmo FFT[4].

Si bien en las definiciones previas se ha tomado la integral y tanto  $\mathbb{R}$  como  $\mathbb{R}^2$  como dominios continuos sobre los que calcular la convolución, no se debe olvidar que las imágenes no dejan de ser matrices o funciones de dos variables con un dominio discreto, por lo que se debe presentar una definición adecuada a este caso[5].

$$(f * g)(u, v) = \sum_{i=-k}^k \sum_{j=-k}^k g(i, j)f(u - i, v - j)$$

En la expresión anterior, a la función  $g(u, v)$  se le llama filtro o *kernel* de convolución. A continuación se muestra un ejemplo de cómo calcular una convolución. Se puede verificar que el cálculo realizado a mano concuerda con el resultado de aplicar la función `convn(X, K, 'valid')` en MATLAB.

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 5 & 6 & 7 & 8 & 9 \\ 9 & 8 & 7 & 6 & 5 \\ 5 & 4 & 3 & 2 & 1 \\ 1 & 2 & 3 & 4 & 5 \end{pmatrix} * \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 14 & 15 & 16 \\ 16 & 15 & 14 \\ 16 & 15 & 14 \end{pmatrix}$$

$$1 \cdot 1 + 2 \cdot 0 + 3 \cdot 0 + 5 \cdot 0 + 6 \cdot 1 + 7 \cdot 0 + 9 \cdot 0 + 8 \cdot 0 + 7 \cdot 1 = 14$$

$$2 \cdot 1 + 3 \cdot 0 + 4 \cdot 0 + 6 \cdot 0 + 7 \cdot 1 + 8 \cdot 0 + 8 \cdot 0 + 7 \cdot 0 + 6 \cdot 1 = 15$$

⋮

$$7 \cdot 1 + 6 \cdot 0 + 5 \cdot 0 + 3 \cdot 0 + 2 \cdot 1 + 1 \cdot 0 + 3 \cdot 0 + 4 \cdot 0 + 5 \cdot 1 = 14$$

Aplicando diferentes kernels de convolución a una imagen se pueden extraer diferentes tipos de características de una imagen, como por ejemplo bordes. El filtro Sobel es capaz de hacer esto con los kernels que se muestran a continuación, pues se comportan como aproximaciones de las derivadas parciales de la imagen en un punto teniendo en cuenta los píxeles cercanos[6].

$$G_u = \frac{\partial f(u, v)}{\partial u} \approx \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \quad G_v = \frac{\partial f(u, v)}{\partial v} \approx \begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}$$

A continuación se va a calcular la convolución de la matriz del ejemplo anterior con  $G_u$ . Para verificar los cálculos, de nuevo se aplica la función `convn` de MATLAB. Al realizar los cálculos a mano tal y como se ha mostrado en el ejemplo anterior, se obtiene como resultado la matriz  $A$ , mientras que MATLAB devuelve la matriz  $B$ , esta vez los resultados no concuerdan. ¿Qué acaba de suceder? ¿Está mal codificada la función de MATLAB? ¿Está mal realizado el ejemplo?

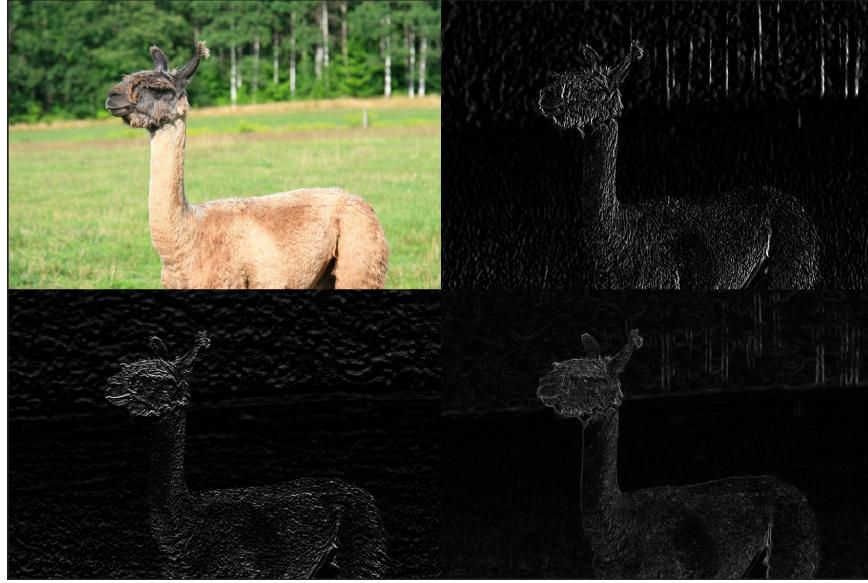


Figura 1.1: Detección de bordes aplicando los kernel Sobel

$$A = \begin{pmatrix} 4 & 4 & 4 \\ -4 & -4 & -4 \\ -4 & -4 & -4 \end{pmatrix} \quad B = \begin{pmatrix} -4 & -4 & -4 \\ 4 & 4 & 4 \\ 4 & 4 & 4 \end{pmatrix}$$

La respuesta a estas preguntas se podría resumir en que se ha realizado una “pequeña trampa” a la hora de explicar cómo calcular la convolución manualmente en el ejemplo, ya que no se ha aplicado correctamente la definición dada. ¿Qué sentido tiene hacer esto? En visión artificial y tratamiento de imágenes, muchos autores y librerías llaman convolución a la operación que se ha mostrado en el primer ejemplo, cuando en realidad no lo es y trae lugar a confusión. Dicha operación se llama correlación cruzada, denotada por  $(\star)$ , y que es muy similar a la convolución, pues su principal diferencia es que en la convolución “real”, el kernel se rota 180 grados antes de calcular la convolución “falsa” o correlación cruzada, es decir,  $X * Y = X \star (R_{180} \cdot Y)$ , donde  $R_{180}$  es la matriz de rotación de 180 grados. En el primer ejemplo se ha utilizado de manera intencionada la matriz  $I_3$  para ver los casos que traen lugar a confusión, pues  $I_3 \cdot R_{180} = I_3$ . La correlación cruzada de dos imágenes (matrices) se define de la siguiente manera[5].

$$(f \star g)(u, v) = \sum_{i=-k}^k \sum_{j=-k}^k g(i, j) f(u + i, v + j)$$

Esta operación sí es con la que realmente se aplican los filtros a las imágenes y con la que se trabaja en general en el campo de la visión artificial. Es importante ver que ahora, al contrario que con la convolución,  $f \star g \neq g \star f$ . Algunas librerías de visión artificial tratan a la correlación cruzada como convolución debido al frecuente uso que tiene una sobre la otra y la forma similar que tienen de calcularse. Un ejemplo es OpenCV para Python y C++ en la documentación de su función `filter2D`, donde se comenta que aplica una convolución cuando realmente aplica la correlación cruzada[7]. Como se verá en próximos capítulos, las famosas redes neuronales convolucionales, no aplican convoluciones sino correlaciones cruzadas.

Finalmenete, a la hora de realizar estas operaciones, se pueden definir una serie de parámetros según convenga para obtener un tamaño diferente de salida[8]. Estos son *stride* y *padding*. El primero de ellos hace referencia a cada cuántos elementos se desplaza el kernel y se calcula el producto escalar, mientras que el segundo a cómo rellenar los bordes de la matriz original para obtener mayor tamaño de salida, habitualmente

se colocan ceros en los bordes y se conoce como *zero-padding*. El tamaño de salida se puede calcular como

$$\frac{n - k + 2p}{s} + 1,$$

y algunos valores por defecto para el *padding* son *valid*, *full*, y *same*. Con *valid* no se añade ningún *padding* de manera que el kernel solo se desliza en las zonas donde la matriz de entrada y el kernel coinciden por completo, con *full* se añaden las filas y columnas de ceros necesarias para poder deslizar el kernel por cualquier zona en la que coincidan la matriz y el kernel, y con *same* también se añaden las necesarias como para producir un tamaño de salida igual al de entrada. A partir de ahora, cuando se utilice el parámetro *full*, las operaciones se escribirán como  $\circledast$  y  $\circledast^*$ , pues la definición original se entiende como *valid*. En ambos casos se tomará un *stride* de 1.



## Capítulo 2

# Aprendizaje automático y profundo

### 2.1. Perceptrón

Ya en el año 1958, el psicólogo Frank Rosenblatt propuso un modelo llamado perceptrón el cual estaba basado en el comportamiento y funcionamiento de las neuronas de un humano, y que podía aprender ponderando cada coeficiente de entrada a la neurona[1]. Hoy en día, tal y como se mostrará en esta sección, el perceptrón es la unidad fundamental de muchos modelos de *machine learning* y *deep learning*.

Como se verá durante esta sección, este modelo ayuda a solucionar problemas de clasificación supervisada. Se dispone de una serie de valores de entrada  $x_1, x_2, \dots, x_n$  y se tiene una serie de valores de salida  $y_1, y_2, \dots, y_m$  que representan a qué clase pertenece la entrada ( $2^m$  clases posibles). Esto se consigue mediante la ayuda de sus parámetros, que son una serie de pesos  $w_1, w_2, \dots, w_n$  y un sesgo o *bias*  $b$ ; y sus hiperparámetros, entre los que se encuentra una función  $f$  de activación.

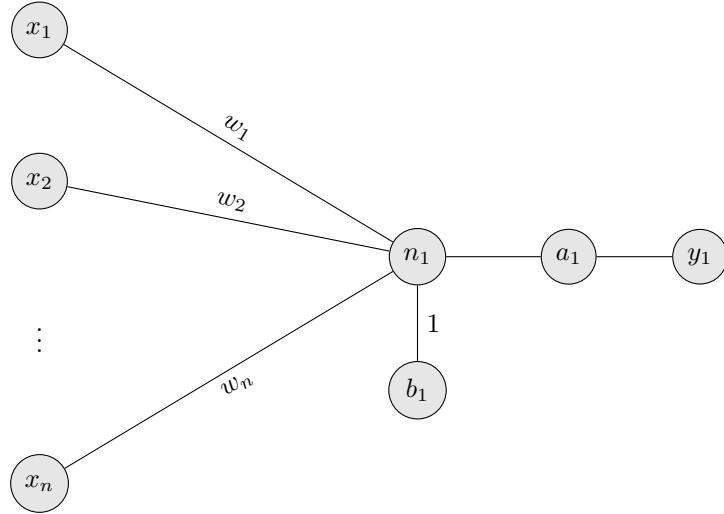


Figura 2.1: Arquitectura de un perceptrón

En la Figura 2.1 se muestra la arquitectura del caso más simple de un perceptrón. Se tienen  $n$  entradas y una única salida. La primera parte del diagrama representa que tal y como decía Rosenblatt, cada valor de entrada debe multiplicarse por un cierto peso, de tal forma que si se representa esto en función de sus valores en un instante  $k$ , lo que se computa en el nodo  $n_1$  es la siguiente operación.

$$n_1(k) = b_1(k) + \sum_{i=1}^n x_i(k)w_i(k)$$

Una vez se ha realizado este cálculo, el valor pasa por una función de activación en el nodo  $a_1$ , pues esta arquitectura es común utilizarla para clasificar una entrada y es muy útil obtener una salida binaria donde se active únicamente la salida que represente la clase a la que pertenece la entrada dada. Aunque existen diferentes funciones de activación para las neuronas, al trabajar con un perceptrón, la función de activación por excelencia es la función escalón de Heaviside, donde  $\mathcal{U} : \mathbb{R} \rightarrow \{0, 1\}$  y su expresión analítica es

$$\mathcal{U}(x) = \begin{cases} 0 & \text{si } x < 0 \\ 1 & \text{si } x \geq 0 \end{cases}.$$

Combinando ambas expresiones, se puede resumir en que la salida del perceptrón es equivalente a la siguiente ecuación:

$$y_1(k) = \begin{cases} 0 & \text{si } b_1(k) + \sum_{i=1}^n x_i(k)w_i(k) < 0 \\ 1 & \text{si } b_1(k) + \sum_{i=1}^n x_i(k)w_i(k) \geq 0 \end{cases}$$

Para dar un ejemplo claro de cómo funciona el perceptrón, se pueden tomar una serie de observaciones que tengan dos valores de entrada y uno de salida. Además, se supondrá que existen dos clases. Esto a fin de cuentas es asignar un valor de 0 o 1 a cada punto de  $\mathbb{R}^2$  tal y como se describe en la Figura 2.2.

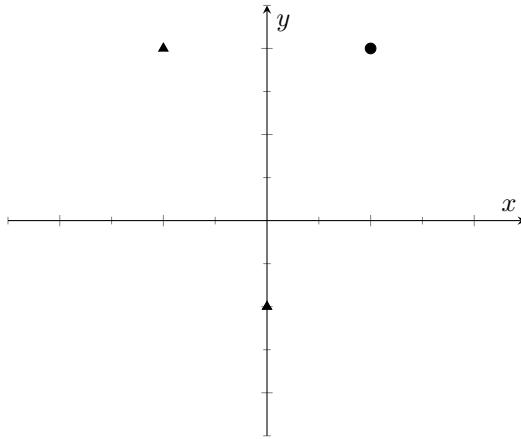
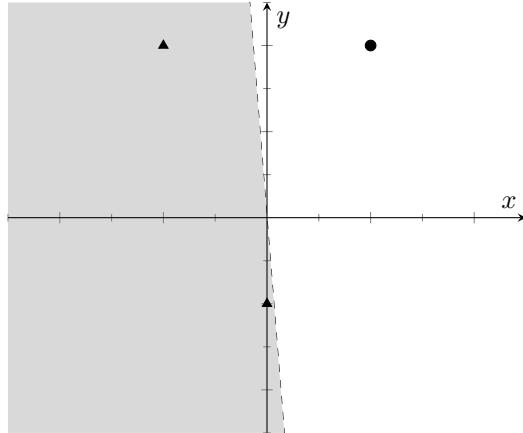
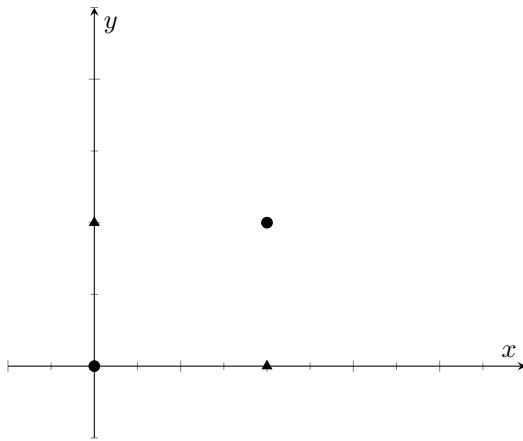


Figura 2.2: Puntos etiquetados en  $\mathbb{R}^2$

Una solución rápida sería trazar una recta  $r : ax + by + c = 0$  que separe  $\mathbb{R}^2$  en dos regiones, de forma que todo punto que pertenezca a una región pertenece entonces a una misma clase, tal y como se observa en la Figura 2.3. Esta recta suele llamarse *decision boundary* o frontera de decisión. El problema entonces es hallar la recta  $r$ , pero se cumple que para este ejemplo es de la forma  $w_1x + w_2y + b = 0$ , siendo el problema encontrar los parámetros adecuados del modelo. La idea puede extrapolarse a diferente tamaño de entrada tomando un hiperplano de la forma  $\mathbf{w}^t \mathbf{x} + b = 0$ .

Las preguntas a resolver ahora son, ¿existen siempre dichos parámetros? ¿Cómo pueden hallarse? El propio Minsky se hizo estas preguntas en [9] y se dio cuenta de que dichos parámetros sí pueden hallarse en un número finito de pasos, siempre y cuando los puntos sean linealmente separables. Un ejemplo que no es linealmente separable es el de la función XOR tal y como se muestra en la Tabla 2.1 y Figura 2.4, pues no existe una recta  $r$  que separe  $\mathbb{R}^2$  en dos regiones de tal forma que cada región contenga puntos de una única clase, sería necesaria una frontera de decisión no lineal.

Figura 2.3: Puntos separados en  $\mathbb{R}^2$ Figura 2.4: Valores de  $x \oplus y$  en  $\mathbb{R}^2$ 

$x$	$y$	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

Tabla 2.1: Función XOR

En cuanto a la pregunta de cómo hallar los parámetros, se consideran las siguientes ecuaciones[10], donde  $\mathbf{w}$  es el vector de pesos,  $t$  el valor esperado, y  $a$  la salida del perceptrón y se aplica el Algoritmo 2.1 para obtener los parámetros óptimos. En dicho algoritmo se supondrá que existe una matriz  $X$  de  $n$  filas que contiene los diferentes  $\mathbf{x}$ .

$$\begin{aligned}
 \mathbf{w}(k+1) &= \mathbf{w}(k) + e(k)\mathbf{x}(k) \\
 b(k+1) &= b(k) + e(k) \\
 e(k) &= t(k) - a(k) \\
 a(k) &= \mathcal{U}(\mathbf{w}^t(k)\mathbf{x}(k))
 \end{aligned} \tag{2.1}$$

A cada una de las iteraciones que realiza el bucle exterior se les denomina épocas o *epoch*, que consiste en realizar el proceso de entrenamiento sobre todo el conjunto de datos. En este caso se está suponiendo que no va a recibir casos que no sean linealmente separables, pero de lo contrario se puede añadir un contador

---

**Algoritmo 2.1:** Regla de aprendizaje del perceptrón

---

```

Datos:  $X, t$ 
Resultado:  $w, b$ 
 $b \leftarrow 0$ 
 $w \leftarrow \text{random}$ 
 $k \leftarrow 0$ 
repetir
     $acabar \leftarrow \text{true}$ 
    para  $i \leftarrow k$  hasta  $k + n - 1$  hacer
         $e(i) \leftarrow t(i) - a(i)$ 
         $w(i+1) \leftarrow w(i) + e(i)x(i) \text{ (mód } n)$ 
         $b(i+1) \leftarrow b(i) + e(i)$ 
         $acabar \leftarrow acabar \wedge e(i) == 0$ 
    fin
     $k \leftarrow k + n - 1$ 
mientras  $\neg acabar$ 

```

---

`max_epochs` y fijar un número máximo para no caer en un bucle infinito. No sería tarea fácil determinar dicho valor, pues aunque el algoritmo converge en los casos previamente explicados, no en todos lo hace de manera rápida. A continuación se muestra cómo obtener una solución para el caso de la Figura 2.2 aplicando el Algoritmo 2.1.

$$X = \begin{pmatrix} 1 & -1 & 0 \\ 2 & 2 & -1 \end{pmatrix} \quad t = (1 \quad 0 \quad 0) \quad w(0) = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad b(0) = 0$$

0.     ■  $a(0) = \mathcal{U}(w^t(0)x(0)) = \mathcal{U}\left((1 \quad 1)\begin{pmatrix} 1 \\ 2 \end{pmatrix} + 0\right) = 1$ 
  - $e(0) = t(0) - a(0) = 1 - 1 = 0$
  - $w(1) = w(0)$
  - $b(1) = b(0)$
1.     ■  $a(1) = \mathcal{U}(w^t(1)x(1)) = \mathcal{U}\left((1 \quad 1)\begin{pmatrix} -1 \\ 2 \end{pmatrix} + 0\right) = 1$ 
  - $e(1) = t(1) - a(1) = 0 - 1 = -1$
  - $w(2) = w(1) + e(1)x(1) = \begin{pmatrix} 1 \\ 1 \end{pmatrix} - \begin{pmatrix} -1 \\ 2 \end{pmatrix} = \begin{pmatrix} 2 \\ -1 \end{pmatrix}$
  - $b(2) = b(1) + e(1) = -1$
2.     ■  $a(2) = \mathcal{U}(w^t(2)x(2)) = \mathcal{U}\left((2 \quad -1)\begin{pmatrix} 0 \\ -1 \end{pmatrix} - 1\right) = 1$ 
  - $e(2) = t(2) - a(2) = 0 - 1 = -1$
  - $w(3) = w(2) + e(2)x(2) = \begin{pmatrix} 2 \\ -1 \end{pmatrix} - \begin{pmatrix} 0 \\ -1 \end{pmatrix} = \begin{pmatrix} 2 \\ 0 \end{pmatrix}$
  - $b(3) = b(2) + e(2) = -2$
  - $\text{acabar} = \text{true} \wedge \text{false} \wedge \text{false} = \text{false}$
3.     ■  $a(3) = \mathcal{U}(w^t(3)x(3) \text{ (mód } 3)) = \mathcal{U}\left((2 \quad 0)\begin{pmatrix} 1 \\ 2 \end{pmatrix} - 2\right) = 1$ 
  - $e(3) = t(3) - a(3) = 1 - 1 = 0$
  - $w(4) = w(3)$
  - $b(4) = b(3)$

4. ■  $a(4) = \mathcal{U}(\mathbf{w}^t(4)\mathbf{x}(4 \text{ (mód 3)})) = \mathcal{U}\left((2 \quad 0) \begin{pmatrix} -1 \\ 2 \end{pmatrix} - 2\right) = 0$

■  $e(4) = t(4) - a(4) = 0 - 0 = 0$

■  $\mathbf{w}(5) = \mathbf{w}(4)$

■  $b(5) = b(4)$

5. ■  $a(5) = \mathcal{U}(\mathbf{w}^t(5)\mathbf{x}(5 \text{ (mód 3)})) = \mathcal{U}\left((2 \quad 0) \begin{pmatrix} 0 \\ -1 \end{pmatrix} - 2\right) = 0$

■  $e(5) = t(5) - a(5) = 0 - 0 = 0$

■  $\mathbf{w}(6) = \mathbf{w}(5)$

■  $b(6) = b(5)$

■  $\text{acabar} = \text{true} \wedge \text{true} \wedge \text{true} = \text{true}$

La ejecución del algoritmo finaliza tras dos épocas, donde en la primera de ellas va ajustando los pesos y el sesgo de manera adecuada, y en la segunda verifica que todas las observaciones han sido clasificadas de manera correcta. La frontera de decisión obtenida es la recta  $r : 2x - 2 = 0$ , que es una recta vertical. Una observación a realizar es que  $\mathbf{x}(1) \in r$ , por tanto ¿a qué clase pertenece? Esto depende de la función de activación empleada, al utilizar  $\mathcal{U}$  la clasificación es correcta, pero al cambiarla por otra, podría no serlo y necesitaría de más épocas para realizar correctamente la clasificación.

La última cuestión que queda por tratar respecto al perceptrón es el porqué se verifica que en el caso de que los puntos dados sean linealmente separables, en un número finito de pasos, el Algoritmo 2.1 terminará su ejecución y con el  $\mathbf{w}$  óptimo, tal y como se enunciaba en el **Teorema de Convergencia del Perceptrón**. A continuación se realiza su demostración basándose en la que se encuentra en [10] pero de forma más clara y simple. Para comenzar con la demostración será necesario definir una serie de elementos, comenzando por  $\Omega(k)$  y  $\mathbf{z}(k)$ . Se entiende que se están concatenando vectores, no definiendo “vectores dentro de vectores”.

$$\Omega(k) = \begin{pmatrix} \mathbf{w}(k) \\ b(k) \end{pmatrix} \quad \mathbf{z}(k) = \begin{pmatrix} \mathbf{x}(k) \\ 1 \end{pmatrix} \quad \Omega(0) = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

Con estos elementos y el cálculo de  $a(k)$  en la Ecuación (2.1) es fácil ver que  $n(k) = \Omega^t(k)\mathbf{z}(k)$ . Recordando los posibles valores de  $t(k)$ , se deseaba que si dicho valor era 1, entonces  $n(k) \geq 0$ ; y en caso de que valiese 0 entonces se deseaba tener  $n(k) < 0$ , en resumen, que  $t(k) - a(k) = 0$ . Otra forma de ver esto es afirmar que en el caso en que  $t(k) \neq a(k)$ , entonces  $\Omega(k)$  debe actualizarse de acuerdo a la Ecuación (2.1). De esta forma  $\Omega(k+1) = \Omega(k) + e(k)\mathbf{z}(k)$ . Ahora se considerará un vector  $\Omega^*$  de forma que

$$\forall k \exists \Omega^* \quad \mathcal{U}(\Omega^{*^t} \mathbf{z}(k)) = t(k),$$

es decir,  $\Omega^*$  es el vector de pesos óptimo. Además, por comodidad se normalizarán todas las distancias del problema, de manera que  $\|\Omega^*\| = 1$  y  $\|\mathbf{z}(k)\| \leq 1$ . El último elemento a considerar será  $\delta$ , que será definido como

$$\delta = \min\{\Omega^{*^t} \mathbf{z}(i)\},$$

tomando además que  $\delta > 0$  pues otra manera de definirlo es la distancia al punto más cercano a la frontera de decisión óptima. Con estos elementos se puede comenzar la demostración. Como la regla de actualización es  $\Omega(k+1) = \Omega(k) + e(k)\mathbf{z}(k)$ , al vector de pesos en un determinado instante (clasificación fallida) se le suma o resta  $\mathbf{z}(k)$  y a priori no se sabe cuántas veces se va a repetir esto, por lo que la idea de la demostración será ver si la norma del vector de pesos tiene una cota superior e inferior, es decir, se para de sumar o restar otros vectores  $\mathbf{z}(i)$ , de manera que el algoritmo terminaría. Para obtener esto basta con comparar el comportamiento de  $\Omega^t(k)\Omega^*$  frente a  $\Omega^t(k)\Omega(k)$  (es decir,  $\|\Omega\|^2$ ).

Con el primero de los términos, al tratar de corregir un error se verifica que

$$\Omega^t(k+1)\Omega^* = (\Omega(k) + e(k)\mathbf{z}(k))^t\Omega^* = \Omega^t(k)\Omega^* + e(k)\Omega^{*^t}\mathbf{z}(k).$$

Además, por la manera en la que se ha definido  $\delta$ , el segundo sumando pertenece al intervalo  $(-\infty, -\delta] \cup [\delta, \infty)$ , pudiendo deducir la siguiente desigualdad, por lo que en una actualización el término sólo varía por lo menos en  $\delta$  unidades.

$$\Omega^t(k+1)\Omega^* \geq \Omega^t(k)\Omega^* + \delta \quad (2.2)$$

De la misma manera que se ha analizado el comportamiento de  $\Omega^t(k)\Omega^*$  se procede con la actualización de  $\Omega^t(k)\Omega(k)$ .

$$\begin{aligned} \Omega^t(k+1)\Omega(k+1) &= (\Omega(k) + e(k)\mathbf{z}(k))^t(\Omega(k) + e(k)\mathbf{z}(k)) \\ &= \Omega^2(k) + (e(k)\mathbf{z}(k))^2 + 2e(k)\Omega^t(k)\mathbf{z}(k) \\ &= \Omega^t(k)\Omega(k) + e^2(k)\mathbf{z}^t(k)\mathbf{z}(k) + 2e(k)\Omega^t(k)\mathbf{z}(k) \\ &= \Omega^t(k)\Omega(k) + e^2(k)\mathbf{z}^t(k)\mathbf{z}(k) + 2e(k)n(k) \end{aligned}$$

En el segundo sumando se tiene que siempre será menor o igual que 1, pues por definición  $0 \leq \mathbf{z}^t(k)\mathbf{z}(k) = \|\mathbf{z}(k)\|^2 \leq 1$ . Además, el tercer sumando siempre será cero o negativo, pues en todos los casos posibles en los que  $e(k) \neq 0$  se cumple que  $e(k)n(k) < 0$ :

- Si  $t(k) = 1 \wedge n(k) < 0$ , entonces  $a(k) = 0 \wedge e(k) > 0$
- Si  $t(k) = 0 \wedge n(k) \geq 0$ , entonces  $a(k) = 1 \wedge e(k) < 0$

De esta situación se puede deducir la siguiente desigualdad, por lo que en una actualización el término sólo varía en como máximo una unidad.

$$\Omega^t(k+1)\Omega(k+1) \leq \Omega^t(k)\Omega(k) + 1 \quad (2.3)$$

Una vez se ha observado cómo varían estos términos al actualizarlos, se puede observar qué pasaría con ellos al hacer  $m$  actualizaciones. Con el resultado obtenido en las Ecuaciones (2.2) y (2.3) se pueden deducir las desigualdades  $\delta m \leq \Omega^t(m)\Omega^*$  y  $\Omega^t(m)\Omega(m) \leq m$ .

Ahora al aplicar la desigualdad de Cauchy–Schwarz<sup>1</sup>, el valor de  $\|\Omega^*\|$ , y la propiedad transitiva, se obtiene una cota superior y otra inferior (ambas recuadradas) para  $\|\Omega^t(m)\|$ , lo que demuestra que el número de actualizaciones es finito y que por tanto el algoritmo converge.

$$\begin{aligned} \boxed{\delta m} &\leq \Omega^t(m)\Omega^* = \|\Omega^t(m)\Omega^*\| \\ &\leq \|\Omega^t(m)\| \|\Omega^*\| = \|\Omega^t(m)\| = \sqrt{\Omega^t(m)\Omega(m)} \\ &\leq \boxed{\sqrt{m}} \end{aligned}$$

$$\delta m \leq \|\Omega(m)\| \leq \sqrt{m} \implies m \leq \frac{1}{\delta^2} \quad \square$$

Esta última desigualdad muestra cómo existe una relación entre el número de iteraciones del algoritmo y la distancia de los datos de entrenamiento a la frontera de decisión óptima (depende únicamente de esto). Cuanto más cerca estén, mayor será el número de iteraciones necesarias.

<sup>1</sup> Esta desigualdad afirma que  $\|\mathbf{u}\mathbf{v}\| \leq \|\mathbf{u}\| \|\mathbf{v}\|$ .

## 2.2. Redes neuronales artificiales

El descubrimiento del perceptrón junto con el teorema que garantizaba que cualquier conjunto de puntos linealmente separable podría ser aprendido mediante este, supuso un gran avance en la IA al igual que una gran desilusión por parte de muchos al no poder aprender una función tan simple como la XOR. Esto causó el llamado el primer invierno de la IA, que finalizó con la llegada de las redes neuronales multicapa y el algoritmo de la retropropagación.

En 1989, George Cybenko enunció y consiguió demostrar el **Teorema de Aproximación Universal**[11]. Este afirma que dada una red neuronal con una capa de entrada, una capa oculta con suficientes neuronas, y una capa de salida; es un aproximador universal de funciones. Es decir, si existe una relación entre dos variables  $\mathbf{x}$  e  $\mathbf{y}$ , entonces una red neuronal con la arquitectura mencionada y el entrenamiento adecuado, encontrará dicha relación y podrá comportarse como una función  $f$  tal que  $\mathbf{y} = f(\mathbf{x})$ . Este se considera un gran resultado, pues consigue acabar con las limitaciones del perceptrón que habían generado un declinamiento por el interés en la IA.

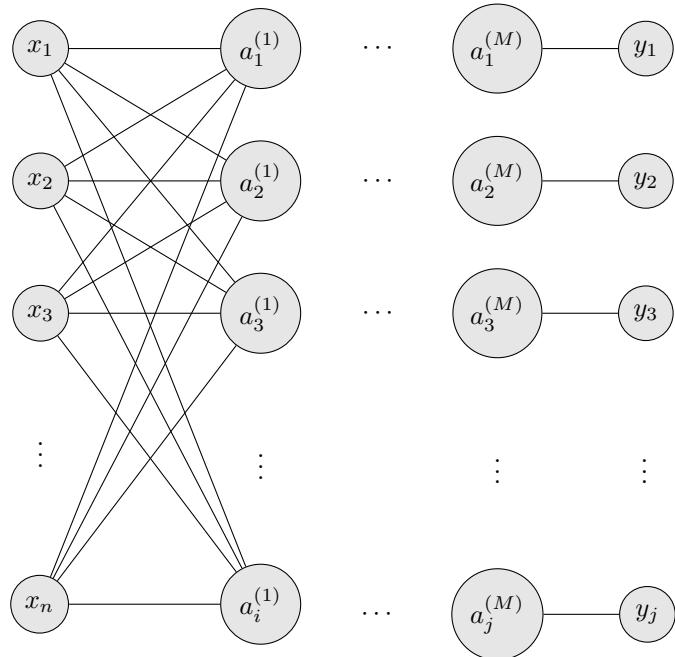


Figura 2.5: Arquitectura de una red neuronal multicapa

En la Figura 2.5 se muestra un diagrama que resume la arquitectura de una red neuronal multicapa. Todas las salidas de una capa están conectadas con todas las entradas de la siguiente con un peso y un *bias*. En el diagrama de la Figura 2.6 se puede observar esto en mayor detalle. Estas pueden ser utilizadas en problemas de clasificación o regresión supervisada, pero en una primera aproximación se supondrá que se está resolviendo un problema de regresión.

Al igual que un perceptrón quedaba representado mediante un vector  $\mathbf{w}$  de pesos y un valor de sesgo  $b$ , para representar una red neuronal multicapa se hace mediante las matrices  $W^{(m)}$  y los vectores  $\mathbf{b}^{(m)}$  y  $\mathbf{f}^{(m)}$ , que contienen los pesos, los sesgos, y las funciones de activación. La notación para los sesgos es  $b_i^{(m)}$ , que representa el sesgo de la entrada  $i$ -ésima de la capa  $m$ . De igual manera  $f_i^{(m)}$  representa la función de activación la neuona  $i$ -ésima de la capa  $m$ . Para los pesos,  $w_{ij}^{(m)}$  denota el peso que une la salida  $j$  con la entrada  $i$  de la capa  $m$ . Para las capas,  $1 \leq m < M$ . En ningún momento un exponente entre paréntesis representa una potencia, en dicho caso aparecerá sin paréntesis para distinguirlo. Siguiendo los mismos pasos que con el perceptrón, la primera pregunta será cómo calcular la salida de una red neuronal. La salida de

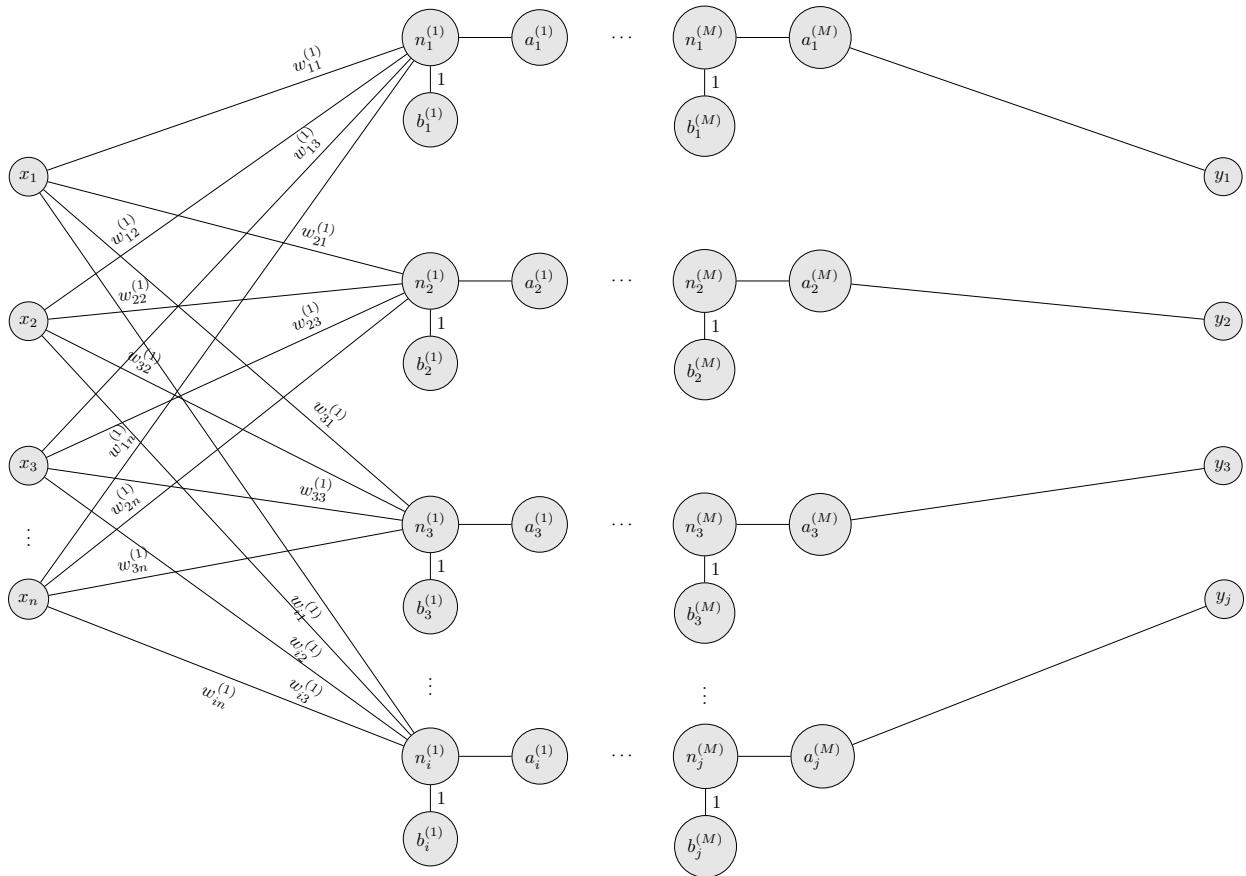


Figura 2.6: Red neuronal multicapa

una capa se calcula con la Ecuación (2.4), teniendo en cuenta que  $\mathbf{x}^{(m)} = \mathbf{y}^{(m-1)}$ , por lo que para calcular la salida de la red no hay más que aplicar dicha ecuación hasta llegar a la última capa.

$$\begin{aligned} \mathbf{a}^{(m)}(k) &= \mathbf{f}^{(m)} \left( W^{(m)}(k) \mathbf{x}^{(m)}(k) + \mathbf{b}^{(m)}(k) \right) \\ \begin{pmatrix} a_1^{(m)}(k) \\ a_2^{(m)}(k) \\ \vdots \\ a_j^{(m)}(k) \end{pmatrix} &= \mathbf{f}^{(m)} \left( \begin{pmatrix} w_{11}^{(m)}(k) & w_{12}^{(m)}(k) & \cdots & w_{1i}^{(m)}(k) \\ w_{21}^{(m)}(k) & w_{22}^{(m)}(k) & \cdots & w_{2i}^{(m)}(k) \\ \vdots & \vdots & \ddots & \vdots \\ w_{j1}^{(m)}(k) & w_{j2}^{(m)}(k) & \cdots & w_{ji}^{(m)}(k) \end{pmatrix} \begin{pmatrix} x_1^{(m)}(k) \\ x_2^{(m)}(k) \\ \vdots \\ x_i^{(m)}(k) \end{pmatrix} + \begin{pmatrix} b_1^{(m)}(k) \\ b_2^{(m)}(k) \\ \vdots \\ b_j^{(m)}(k) \end{pmatrix} \right) \end{aligned} \quad (2.4)$$

A continuación, se puede plantear cuál es la ecuación del error para una red neuronal. Esta es simplemente cuestión de elección al igual que las funciones de activación. Al trabajar con redes neuronales para problemas de regresión, la función de error por excelencia es el error cuadrático, definido como

$$L(k) = \sum (\mathbf{t}(k) - \mathbf{a}^{(M)}(k))^2,$$

aunque existen otras muchas, cada una adecuada a cada tipo de problema y modelo, como por ejemplo el error cuadrático medio, error absoluto, error absoluto medio, error logarítmico, error exponencial, entropía cruzada, etc[12].

La pregunta ahora sería que, al igual que existía una ecuación para actualizar los parámetros del perceptrón ¿existe para una red neuronal? Para deducirla fácilmente, basta en pensar que  $L$  es una función que depende de los parámetros de la red y que se quiere que su valor sea lo más pequeño posible para diferentes vectores de entrada, es decir, encontrar los valores de los parámetros que minimizan  $L$ . Esto es un problema clásico de cálculo que en el caso de una variable se resuelve igualando a cero la derivada de la función, y en el caso de varias, mediante la matriz Hessiana. Sin embargo, dichos métodos para una función de tantas variables y con expresiones complejas, no son muy eficientes.

Esto se soluciona con ayuda de un algoritmo conocido como descenso por gradiente[13]. Este algoritmo calcula de manera iterativa una aproximación de los mínimos de una función con ayuda del vector gradiente de una función. El vector gradiente de una función  $f$  se define como

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

e indica la dirección en la que la función crece más rápido, siendo la idea principal del algoritmo ir moviéndose en dirección contraria a este.

---

**Algoritmo 2.2:** Descenso por gradiente

---

**Datos:**  $f, \alpha, k$   
**Resultado:**  $\mathbf{x}(k)$   
 $\mathbf{x}(0) \leftarrow \text{random}$   
**para**  $i \leftarrow 1$  **hasta**  $k$  **hacer**  
  |  $\mathbf{x}(i+1) = \mathbf{x}(i) - \alpha \nabla f(\mathbf{x}(i))$   
**fin**

---

Como se observa en el Algoritmo 2.2, se comienza en un punto aleatorio, lo que puede variar la calidad de la solución dependiendo de la ejecución, y además se introduce un término  $\alpha$  llamado **tasa de aprendizaje**. Pueden existir casos en los que la magnitud del gradiente sea muy grande, lo que resulta en desplazamientos bruscos y una convergencia más lenta, tal y como se refleja en la Figura 2.7. Si se hubiese fijado un valor máximo de  $k = 25$ , en los casos de las Figuras 2.7b y 2.7c, no se hubiese llegado a una buena aproximación del mínimo por haber elegido un  $\alpha$  inadecuado.

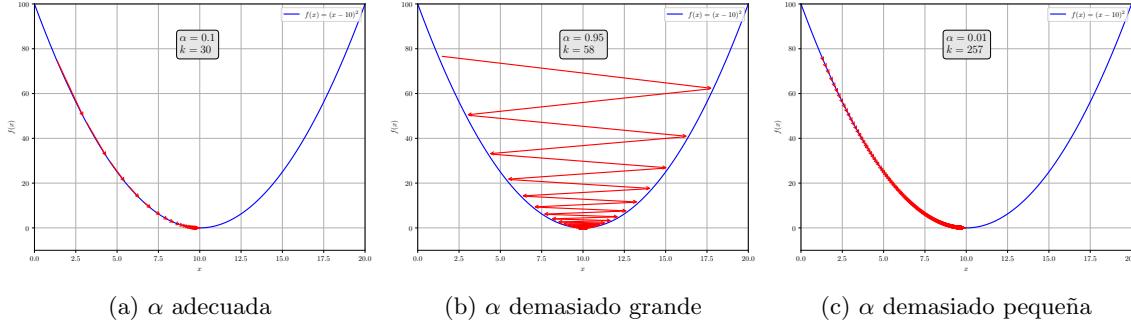


Figura 2.7: Descenso por gradiente

Con esta idea de la función de error y el descenso por gradiente, aparece el famoso algoritmo de la retropropagación o *backpropagation*. Es muy similar al algoritmo de aprendizaje del perceptrón pero adaptado a la estructura de una red neuronal. El primer paso consiste en dada una entrada, calcular la salida de la red con ayuda de la Ecuación (2.4). El segundo paso es calcular el error  $L(k)$ . Esta función depende de todos los pesos y sesgos de la red, por lo que el tercer paso será modificar estos de acuerdo a  $\nabla L$  y repetir el procedimiento para el resto de observaciones. Se realizan tantas épocas como sean necesarias.

El problema que queda por resolver es cómo calcular todos los elementos de  $\nabla L$ , pues por ejemplo es fácil calcular  $\frac{\partial L}{\partial a_i^{(M)}}$ , pero no parece tan obvio calcular  $\frac{\partial L}{\partial w_{ij}^{(m)}}$ , pues hay que retroceder  $M - m$  capas, y habrá muchos valores que dependan de ese peso. La solución a esto es la regla de la cadena.

$$\frac{\partial f}{\partial x} = \sum_{i=1}^n \frac{\partial f}{\partial u_{i1}} \frac{\partial u_{i1}}{\partial x} \prod_{j=1}^{m-1} \frac{\partial u_{ij}}{\partial u_{ij+1}}$$

Con ayuda de esta regla se pueden calcular fácilmente los términos no triviales del gradiente, propagando el error hacia atrás por la red hasta llegar al parámetro deseado mediante las sensibilidades  $(\delta_i^{(s)})$  [10]. De aquí se intuye el porqué del nombre del algoritmo. De manera informal, lo que hace el algoritmo es castigar a cada neurona de manera proporcional a su participación en el error final.

$$\begin{aligned} \frac{\partial L}{\partial b_i^{(s)}} &= \frac{\partial L}{\partial a_i^{(s)}} \frac{\partial a_i^{(s)}}{\partial n_i^{(s)}} \frac{\partial n_i^{(s)}}{\partial b_i^{(s)}} = \delta_i^{(s)} \\ \frac{\partial L}{\partial w_{ij}^{(s)}} &= \frac{\partial L}{\partial a_i^{(s)}} \frac{\partial a_i^{(s)}}{\partial n_i^{(s)}} \frac{\partial n_i^{(s)}}{\partial w_{ij}^{(s)}} = \delta_i^{(s)} \frac{\partial n_i^{(s)}}{\partial w_{ij}^{(s)}} = \delta_i^{(s)} a_j^{(s-1)} \\ \delta_i^{(M)} &= \frac{\partial L}{\partial a_i^{(M)}} \frac{\partial a_i^{(M)}}{\partial n_i^{(M)}} = -2a_i^{(M)} \frac{\partial a_i^{(M)}}{\partial n_i^{(M)}} = -2a_i^{(M)} \frac{\partial f_i^{(M)}}{\partial n_i^{(M)}} \\ \delta_i^{(m)} &= \frac{\partial L}{\partial n_i^{(m)}} = \sum_{l=1}^p \frac{\partial L}{\partial n_l^{(m+1)}} \frac{\partial n_l^{(m+1)}}{\partial n_i^{(m)}} = \sum_{l=1}^p \delta_l^{(m+1)} \frac{\partial n_l^{(m+1)}}{\partial n_i^{(m)}} = \sum_{l=1}^p \delta_l^{(m+1)} w_{li}^{(m+1)} \frac{\partial f_i^{(m)}}{\partial n_i^{(m)}} \end{aligned}$$

con  $0 < m < M$  y  $0 < s \leq M$

Estas ecuaciones pueden reescribirse de manera matricial para aligerar la notación, que combinándolas con las ideas explicadas referentes al Algoritmo 2.2, da lugar al Algoritmo 2.3, bastante similar al Algoritmo 2.1 pero adaptado a una red neuronal.

En esta variante del algoritmo (denominada estocástica o **SGD**) se actualizan los parámetros por cada observación del *dataset* y se ha decidido elegir como criterio de parada alcanzar un número de épocas, aunque también se suelen tomar otros criterios, como la magnitud del error. Además, esta variante es computacionalmente costosa, pues para cada observación se calcula el gradiente. Otra variante del algoritmo es el **batch**. Esta no actualiza por cada observación como SGD, halla el gradiente del error promedio de

**Algoritmo 2.3:** Retropropagación (*backpropagation*)

---

**Datos:**  $f^{(s)}, \mathbf{x}^{(1)}(k), \mathbf{t}(k), \varepsilon$   
**Resultado:**  $W^{(s)}, \mathbf{b}^{(s)}$

para  $i \leftarrow 1$  hasta  $\varepsilon$  hacer

para cada  $\mathbf{x}^{(1)}$  hacer

calcular  $\mathbf{a}^{(M)}(k)$   
 calcular  $L(k)$  y  $\nabla L(k)$   
 $\delta^{(M)}(k) \leftarrow -2 \frac{\partial f^{(M)}}{\partial \mathbf{n}^{(M)}}(\mathbf{t}(k) - \mathbf{a}^{(M)}(k))$   
 $\delta^{(m)}(k) \leftarrow \frac{\partial f^{(M)}}{\partial \mathbf{n}^{(M)}} W^{(m+1)^t}(k) \delta^{(m+1)}(k)$   
 $W^{(m)}(k+1) \leftarrow W^{(m)}(k) - \alpha \delta^{(m)}(k) (\mathbf{a}^{(m-1)}(k))^t$   
 $\mathbf{b}^{(m)}(k+1) \leftarrow \mathbf{b}^{(m)}(k) - \alpha \delta^{(m)}(k)$

fin

fin

---

todas las observaciones del *dataset*. No es recomendable, pues es muy costosa y si el *dataset* ocupa mucho, puede no caber en memoria. Ambas aproximaciones se pueden combinar para dar lugar a una variante más eficiente que estas dos, denominada **mini-batch**, que realiza lo mismo que *batch* pero dividiendo el *dataset* en *minibatches* y realizando una actualización de los parámetros por cada uno de ellos. De esta manera no se necesita tener todo el *dataset* en memoria[13].

### 2.2.1. Funciones de activación

Durante el estudio del perceptrón, se muestra cómo se emplea la función de Heaviside como función de activación en la neurona. Esto se debe a que se busca una función que independientemente de los valores de entrada que reciba, produzca una salida binaria. En ciertos casos, no se deseará una salida binaria pues el problema no tendría porqué ser de clasificación, como se ha visto con las redes neuronales. Además, como se ha visto en el Algoritmo 2.3, será imprescindible poder calcular la derivada de estas funciones. Por estos motivos, se presentan las funciones de activación más conocidas e importantes[14], [15].

- **Función lineal**

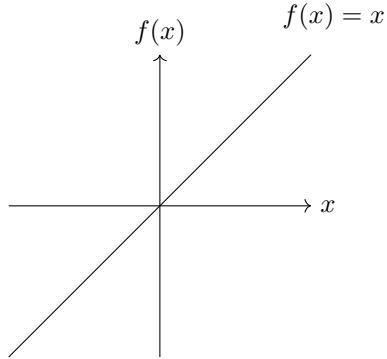


Figura 2.8: Función lineal

Con motivo de emplear una función que no produzca valores binarios, se pueden tomar funciones lineales  $f : \mathbb{R} \rightarrow \mathbb{R}$  de la forma  $f(x) = ax$ . No es muy útil al trabajar con tareas complejas, solo es capaz de cumplir con su tarea en problemas sencillos, y esto en parte se debe a la expresión de su

derivada. Al ser un polinomio, es continua y derivable en todo  $\mathbb{R}$ , teniendo que

$$\frac{df}{dx} = a.$$

#### ■ Función logística

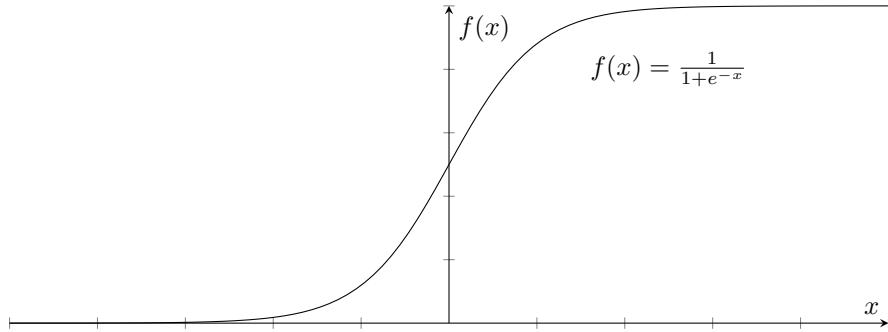


Figura 2.9: Función logística

Esta también es una función que no devuelve valores binarios, es conocida como función sigmoide por la forma de S que tiene, y es una función  $f : \mathbb{R} \rightarrow (0, 1)$ . Una propiedad que es muy útil es que es solución de la ecuación diferencial

$$\frac{df}{dx} = f(x)(1 - f(x)).$$

Es un intento de mejora de la función de activación empleada en el perceptrón, pues una ligera variación en la entrada puede crear un gran cambio en la salida. Esta función evita que eso suceda, pues una pequeña variación en la entrada produce una variación pequeña en la salida.

#### ■ Tangente hiperbólica

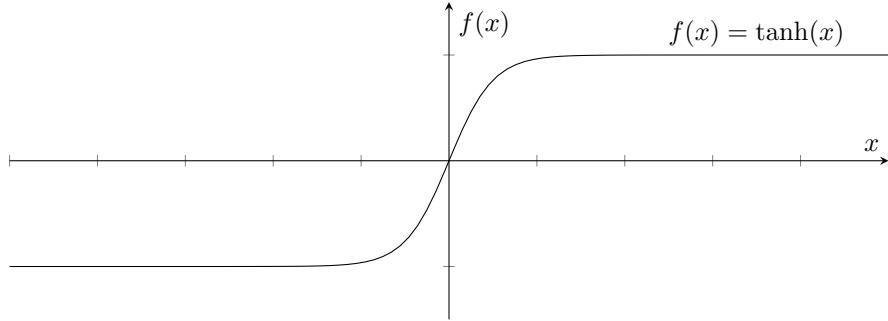


Figura 2.10: Tangente hiperbólica

De nuevo, esta función no devuelve valores binarios, y que guarda cierta relación con la logística, pues esta tiene también forma de sigmoide, pero a diferencia que la logística, esta verifica que  $f(-x) = -f(x)$  por lo que es preferible sobre esta, y también hace que al usarla en redes neuronales su entrenamiento converja más rápido. Además  $f : \mathbb{R} \rightarrow (-1, 1)$  y su expresión es

$$f(x) = \tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}},$$

y verifica una segunda propiedad muy útil, es solución de la ecuación diferencial

$$\frac{df}{dx} = 1 - f^2(x).$$

### ■ Función ReLU

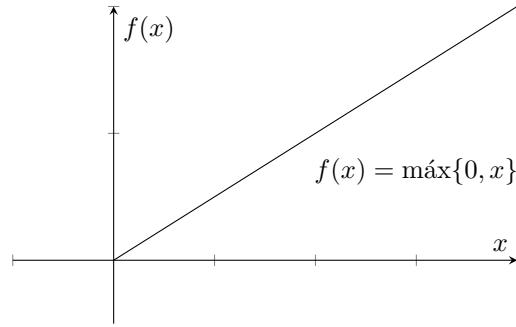


Figura 2.11: Función ReLU

La función ReLU (Rectified Linear Unit) es una de las más populares al trabajar con redes neuronales, pues a pesar de que no es derivable en  $x = 0$  (normalmente se soluciona tomando  $f'(0) = 1$ ), soluciona un serio problema que causan las funciones logística y tangente hiperbólica durante el entrenamiento de una red neuronal. Este problema es conocido como *vanishing gradient* y de forma resumida consiste en que cuando la derivada de una función de activación es muy próxima a cero, relentiza enormemente el proceso de aprendizaje, pues si se observa con detalle cómo se calculan las actualizaciones de los pesos y sesgos en el Algoritmo 2.3, si los términos  $\delta^{(s)}$  tienden a cero, la diferencia entre los parámetros de una iteración a otra tiende a cero, necesitando una cantidad enorme de iteraciones. Como se observa a continuación, la función ReLU soluciona este problema. Además, la derivada es mucho más sencilla de calcular.

$$\lim_{x \rightarrow \infty} \frac{d}{dx} \frac{1}{1 + e^{-x}} = 0 \quad \lim_{x \rightarrow \infty} \frac{d}{dx} \tanh(x) = 0 \quad \lim_{x \rightarrow \infty} \frac{d}{dx} \text{ReLU}(x) = 1$$

Si bien soluciona este problema mencionado para valores de  $x > 0$ , genera el mismo problema para valores negativos. Para solucionar este problema se suelen tomar variantes de la función ReLU conocidas como LReLU, PReLU, o ELU; que modifican su expresión para valores de  $x \leq 0$  como funciones lineales o exponenciales.

### ■ Función softmax

Al abordar un problema de regresión con una red neuronal, puede entenderse como encontrar una función  $\mathbf{f} : \mathbb{R}^n \rightarrow \mathbb{R}^m$ , sin embargo, al utilizar redes neuronales para un problema de clasificación, lo que se necesita es una función  $\mathbf{f} : \mathbb{R}^n \rightarrow [0, 1]^m$ , es decir, se quiere obtener como respuesta el nombre de la clase a la que pertenece la entrada. La solución a este problema es introducir la función softmax en la capa de salida de la red.

$$\mathbf{f}(\mathbf{x}) = \frac{e^{\mathbf{x}_i}}{\sum_{j=1}^k e^{\mathbf{x}_j}}$$

Esta función verifica que  $\sum_{p=1}^q \mathbf{f}_p(\mathbf{x}) = 1$ , es decir, en la capa de salida de la red se obtiene la probabilidad de que una muestra presentada a la red pertenezca a una determinada clase. A dicha muestra se le asigna la clase que mayor probabilidad tiene. En este caso, no debe hablarse de su derivada sino de su matriz Jacobiana donde aparecerán las derivadas que puedan necesitarse durante *Backpropagation*, siendo de la forma

$$\frac{\partial \mathbf{f}_i(\mathbf{x})}{\partial \mathbf{x}_j} = \mathbf{f}_i(\mathbf{x})(\delta_{ij} - \mathbf{f}_j(\mathbf{x})),$$

donde  $\delta_{ij}$  es la función Delta de Kronecker.

## 2.3. Redes neuronales convolucionales

Una vez explicado cómo funciona una red neuronal y cómo puede usarse para problemas de regresión y clasificación, es interesante poder aplicar estas tareas sobre imágenes en vez de sobre conjuntos de datos numéricos. Una imagen de  $n \times m$  píxeles en escala de grises puede entenderse como una matriz de  $n \times m$  elementos, sin embargo, suelen utilizarse imágenes a color y esto se puede conseguir utilizando tres canales, rojo, azul, y verde. De esta forma, una imagen se representa como tres matrices. Formalmente, una imagen se representa como un tensor, que puede entenderse como un vector de matrices, o una estructura que indexa elementos mediante una tupla  $(i, j, k)$ .

De manera ingenua, una primera aproximación para clasificar imágenes en función de objetos que aparecen en estas, podría ser linealizar este tensor y pasarlo como vector de entrada a un red neuronal multicapa, obteniendo un vector de salida que indique a qué clase pertenece la imagen. Esto, además de que no da resultados positivos pues si se mueve o varía de tamaño el objeto a reconocer, la red ya no lo entendería igual al haber aprendido los valores de cada píxel concreto en cada caso; computacionalmente es muy complicado de abarcar. Suponiendo que se tiene una imagen RGB cuadrada de  $n \times n$  píxeles, y que la primera capa oculta tuviera  $m$  neuronas, el número de parámetros de esta capa sería  $m(3n^2 + 1)$ . Suponiendo imágenes de baja calidad, por ejemplo  $64 \times 64$  píxeles, y que el número de neuronas de la primera capa fuese 5, el número de parámetros es de 61445. Ajustar adecuadamente esa cantidad de parámetros muy costoso, y además daría resultados muy pobres en el caso de encontrar los parámetros adecuados. Se necesita simplificar la magnitud del problema y enseñar a la red a ver y entender, no solo aprender valores de píxeles.

La forma de trabajar con imágenes y redes neuronales consiste en “resumir” las diferentes regiones de la imagen pasando como entrada de la red neuronal aquellas características destacables, es decir, un vector o mapa de características propias de la imagen. Un ejemplo de estos se muestra en la Figura 2.12. Esto se puede lograr mediante las operaciones de convolución y correlación cruzada explicadas en la Sección 1.3.

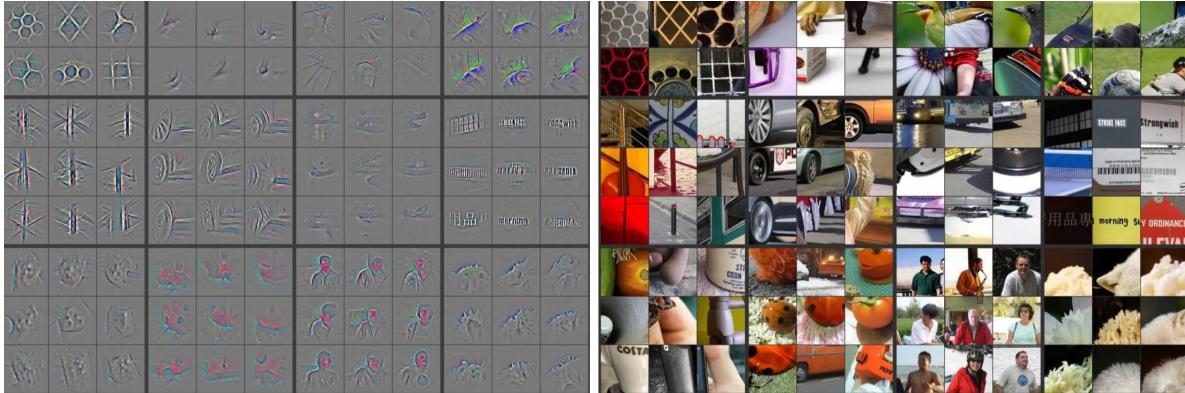


Figura 2.12: Imágenes y sus mapas de características[16]

En general, para poder clasificar imágenes, se utilizan redes convolucionales por el motivo explicado, y para realizar dicha tarea se van concatenando diferentes tipos de capas, encontrándose entre estas las que se muestran a continuación[8]. Combinándolas se crea una red convolucional como por ejemplo, la que se muestra en la , que se trata de la red VGG-16.

### ■ Capa de entrada

Esta capa recibe la entrada de la red, normalmente la imagen representada como un tensor, es decir, una matriz por cada canal de la imagen.

### ■ Capa de convolución

En este tipo de capas es donde reside principalmente el conocimiento de una red convolucional, pues dispone de unos parámetros llamados *kernels* y *bias*, o sesgos. Lo que hace esta capa es calcular la correlación cruzada entre la entrada y el kernel y sumar un sesgo, a pesar de que comúnmente se

dice que calcula la convolución, como se discutió en la Sección 1.3. El número de salidas de esta capa depende del número de kernels y sesgos que se definan, se puede establecer un símil con el número de neuronas de una capa de una red neuronal. La entrada de la capa, que es un tensor, puede entenderse como una serie de matrices  $X_i^{(m)}$ , los kernels que serían otros tensores, como otra serie de matrices  $K_{ij}^{(m)}$ , y los sesgos como las matrices  $B_i^{(m)}$ , teniendo la siguiente ecuación para la salida de una capa de convolución.

$$N_i^{(m)} = B_i^{(m)} + \sum_{j=1}^n X_j^{(m)} \star K_{ij}^{(m)}$$

#### ■ Capa de activación

Al igual que en las redes neuronales clásicas se aplicaba una función de activación a la salida de cada neurona, en las redes convolucionales es habitual hacer esto mismo con las salidas de una capa de convolución y la función ReLU. Se aplica esta a cada elemento del tensor de salida, pudiendo verse como sustituir por ceros aquellos valores negativos producidos por la correlación cruzada.

#### ■ Capa de *pooling* o agrupación

Como previamente se ha mencionado, la ventaja de trabajar con redes convolucionales en vez de con las clásicas, es que estas son capaces de extraer las características importantes de una imagen, “resumiendo” esta antes de comenzar el trabajo de clasificación. Parte de este “resumen” se realiza mediante las capas de *pooling* o agrupación. Se suelen utilizar las funciones maxpooling o avgpooling con kernels de tamaño  $2 \times 2$  sobre la entrada, reduciendo el tamaño de esta en un 25 %.

$$\text{maxpooling}\{x_1, x_2, \dots, x_n\} = \max\{x_1, x_2, \dots, x_n\}$$

$$\text{maxpooling} \left( \begin{array}{cc|cc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{array} \right) = \begin{pmatrix} 6 & 8 \\ 14 & 16 \end{pmatrix}$$

$$\text{avgpooling}\{x_1, x_2, \dots, x_n\} = \frac{x_1 + x_2 + \dots + x_n}{n}$$

$$\text{avgpooling} \left( \begin{array}{cc|cc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{array} \right) = \begin{pmatrix} 3,5 & 5,5 \\ 11,5 & 13,5 \end{pmatrix}$$

#### ■ Capa densa o totalmente conectada

Esta capa se trata de una de las capas de una red neuronal clásica. Recibe los vectores de características una vez han sido simplificados lo suficiente como para que una red clásica pueda trabajar con ellos y encontrar las relaciones entre las características de las imágenes de una misma clase.

#### ■ Capa softmax

Esta capa aplica la función softmax en la salida de la red neuronal para poder ajustar la salida a un problema de clasificación usual, tal y como se explicó anteriormente.

Habiendo mostrado ya la arquitectura de una red convolucional con sus correspondientes parámetros (kernels y sesgos) e hiperparámetros (*padding* y *stride*), la pregunta clásica es, ¿cómo obtener los parámetros adecuados? La respuesta de cómo obtenerlos es simple, siendo algo más complicada de responder esta vez la de cuáles son. Al no dejar de ser una red neuronal, para obtener los parámetros de la red se utiliza *Backpropagation* (Algoritmo 2.3) también, añadiendo las siguientes ecuaciones[17] al cálculo del gradiente para las capas de convolución. En este caso también aparece la recursividad mediante las sensibilidades de convolución  $\Lambda_i^{(m)}$  hasta llegar a las sensibilidades clásicas  $\delta_i^{(m)}$ .

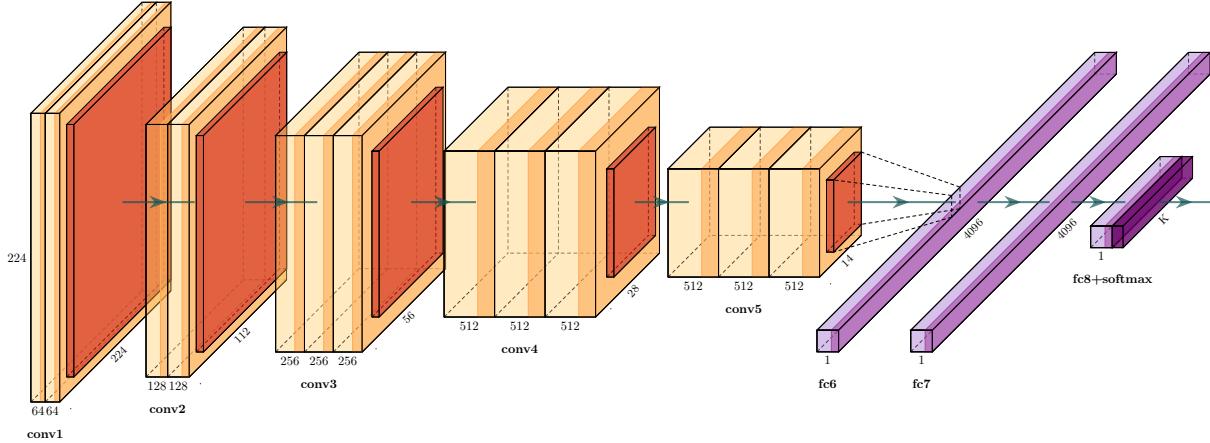


Figura 2.13: Red convolucional VGG-16

$$\begin{aligned}\Lambda_i^{(m)} &= \frac{\partial L}{\partial N_i^{(m)}} \\ \frac{\partial L}{\partial B_i^{(m)}} &= \Lambda_i^{(m)} \\ \frac{\partial L}{\partial K_{ij}^{(m)}} &= X_j^{(m)} \star \Lambda_i^{(m)} \\ \frac{\partial L}{\partial X_j^{(m)}} &= \sum_{i=1}^n \Lambda_i^{(m)} \circledast K_{ij}^{(m)}\end{aligned}$$

En este caso, como se comentaba anteriormente, a pesar de contar con el algoritmo para hallar los parámetros, no es suficiente para entrenar la red y obtener un resultado adecuado. Las redes convolucionales son un modelo de deep learning que necesita cantidades de datos y un poder computacional muy elevado. Para obtener un resultado decente se necesitan muchísimos datos de ejemplo y equipos con muchos recursos, como diferentes GPUs y grandes cantidades de memoria.

## 2.4. Transformers

Si bien llegado este punto se podría afirmar que se ha conseguido una arquitectura que puede lograr el fin de este proyecto (clasificar imágenes), esta es mejorable en diversas situaciones. Supóngase que se dispone de un gran conjunto de imágenes de animales terrestres, acuáticos, y aéreos; y se necesita un modelo capaz de distinguir animales entre estas tres clases. Tras etiquetar estas imágenes manualmente, encontrar una arquitectura de CNN adecuada, y entrenarla, se obtiene un clasificador capaz de distinguir entre estas tres clases. Tiempo más tarde, surge la necesidad de poder clasificar dentro de estas tres clases, por ejemplo, dentro de los animales acuáticos poder distinguir entre peces de río, ballenas, tiburones, etc. Esto supondría reetiquetar el dataset manualmente, modificar la arquitectura de las capas densas, y volver a entrenar el modelo. Considerando un caso extremo, podría además ser el dataset muy grande y no conocerse el número de especies dentro de cada una de las clases iniciales y querer separar las imágenes por cada una de las especies. Esto no sería posible con una CNN, pues necesitaría tener definidas una serie de clases.

Por otro lado, supóngase que dado un texto que describa una imagen, por ejemplo “*Un pastor alemán corriendo en un bosque*”, se quiere obtener la imagen del dataset que se adapta mejor a esa descripción, o se quiere obtener cómo de similares son dos imágenes del dataset, o se quiere extraer el texto que contiene una imagen, o se quiere comprobar cómo de bien se adapta una descripción de una imagen a esta. Todo este tipo

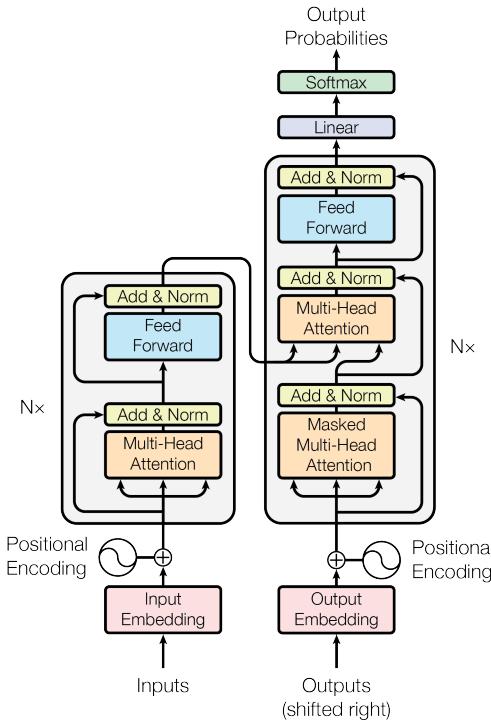


Figura 2.14: Arquitectura de un transformer codificador-decodificador

de tareas y similares, serían imposibles de realizar mediante redes convolucionales, ya que no están enfocadas a trabajar con texto, y es aquí donde entran este tipo de modelos llamados transformers. Se presentaron en 2017 en un artículo escrito por investigadores de Google (con un nombre bastante acertado, *Attention is all you need* [18]) como un modelo sustituto de las redes neuronales recurrentes debido al problema que tienen al intentar recordar información de instantes alejados. La idea original era utilizarlos para problemas de NLP, como por ejemplo la traducción de textos, pero se pueden aplicar a una diversidad de problemas. Basándose en dicho artículo, se presenta una breve explicación de cómo funciona la arquitectura de un transformer.

#### 2.4.1. Codificación

El flujo que sigue un elemento de entrada en un transformer, es bastante más elaborado que el que se sigue en una red neuronal, tal y como se observa en la Figura 2.14. Inicialmente comienza con la etapa de codificación, la que el elemento recibido (en este primer ejemplo, un texto) se divide en una serie de tokens o elementos básicos e indivisibles, se añaden los tokens `$SOS$` (start of sequence) y `$EOS$` para marcar el inicio y el final del contenido, y se convierte cada uno de ellos en una serie de embeddings  $\mathbf{e} \in \mathbb{R}^{d_m}$  que codifican el token. Un ejemplo sería dividir una frase por palabras, y asignar un vector diferente a cada palabra del diccionario, aunque en los modelos reales de NLP no se separa de esta manera. Continuando con el proceso, se necesita almacenar de alguna manera la posición de los tokens de entrada, pues el transformer procesa todos los tokens de manera simultánea y la frase “*Juan es mayor que María*”, es totalmente diferente a, por ejemplo “*María es mayor que Juan*”, a pesar de ser dos secuencias con los mismos tokens. Para lograr esto, a cada embedding  $\mathbf{e}$  se le suma un vector posicional  $\mathbf{p}$  donde cada una de sus componentes  $p_i$  se obtienen de la siguiente manera, siendo  $d_m$  la dimensión de los embeddings y  $n$  la posición del token.

$$p_i(n) = \begin{cases} \operatorname{sen}\left(\frac{n}{10000^{\frac{2i}{d}}}\right) & \text{si } i \equiv 0 \pmod{2} \\ \operatorname{cos}\left(\frac{n}{10000^{\frac{2i}{d}}}\right) & \text{si } i \not\equiv 0 \pmod{2} \end{cases}$$

Existen diferentes maneras de codificar posiciones de tokens, pero en la arquitectura original se utilizó

la presentada. El siguiente paso es el conocido como algoritmo de atención, siendo su misión la siguiente. Observando las frases “*Juan es un catador profesional de vino*” y “*María vino ayer de Madrid*” se ve que el token **vino** aparece en ambas, e inicialmente se le asignaría un embedding idéntico, sin embargo en ambos casos se le asignaría el mismo embedding, ignorando el contexto en el que aparece el token y perdiendo su significado original. De la misma manera, en la frase “*Ellos son Juan y María, él es alto y ella es baja*”, necesitaría codificarse que los tokens **alto** y **baja** se refieren respectivamente a los tokens **él** y **ella**, y a su vez a los tokens **Juan** y **María**. La misión del algoritmo de atención es justo esta, transformar los embeddings de cada token para que su codificación se adapte al contexto.

Esto se logra mediante tres matrices  $W_Q$ ,  $W_K$ , y  $W_V$  que almacenan parámetros entrenables del transformer (es decir, aquí reside parte del conocimiento del modelo). Estas se llaman query o consulta, key o clave, y value o valor. El primer paso es calcular el producto de la matriz  $W_Q$  con cada embedding  $e_i$  obteniendo un vector de consulta  $q_i$ . De manera similar se realiza con las claves, para obtener los vectores  $k_i$ . Una vez se tienen todos los vectores consulta y clave, se forman las matrices  $Q$  y  $K$  que contienen estos vectores, pues al calcular  $QK^t$  se obtiene una matriz donde cada una de las entradas es el producto escalar  $q_i \cdot k_i$ , que es una manera de indicar la relación que existe entre los dos embeddings, o la “atención” que presta el uno al otro. Para evitar problemas como el *vanishing gradients*, la matriz se reescalía por un factor  $\sqrt{d_k}^{-1}$ . En algunos casos se aplica el llamado enmascaramiento, que consiste en sustituir ciertos valores por  $-\infty$  para eliminar su atención, pues para visualizar en términos de probabilidad qué embedding se relaciona más con otro dado, se aplica la función softmax por columnas. Para dichos casos, se supondrá que  $\text{mask}(C)$  aplica la máscara deseada, y en caso contrario  $C = \text{mask}(C)$ .

En este punto, se habría detectado la relación entre los tokens de **Juan** y **alto**, y los de **María** y **baja**, sin embargo, se necesita modificarlos para que dicha información quede codificada. Para realizar esto, se calculan una serie de vectores de valor  $v_i$  como el producto de  $W_V$  con  $e_i$ , que forman la matriz  $V$ . Al multiplicar dicha matriz por la obtenida en el paso anterior, se obtiene la matriz de atención  $A$ .

$$A = \text{softmax}_j \left( \frac{\text{mask}(QK^t)}{\sqrt{d_k}} \right) V$$

Esta matriz  $A$  contiene una serie de vectores que reflejan el cambio que debe hacerse a cada embedding para que codifiquen la información del contexto recopilada. Si se tiene una matriz  $E$  de embeddings, se actualizaría de manera que  $E' = E + A$ . Al realizar esta suma, el embedding del token **Juan** ya almacena información sobre que es alto, o por ejemplo en las frases mostradas previamente, los embeddings para el token **vino** ya serían completamente diferentes. Este proceso se conoce como atención “de cabeza única” o simple, pues normalmente se aplica el conocido como “cabeza múltiple”.

---

**Algoritmo 2.4:** Atención simple

---

**Datos:**  $E \in \mathbb{R}^{n \times d_m}$ ,  $W_Q \in \mathbb{R}^{d_m \times d_m}$ ,  $W_K \in \mathbb{R}^{d_m \times d_m}$ ,  $W_V \in \mathbb{R}^{d_m \times d_m}$

**Resultado:**  $E' \in \mathbb{R}^{n \times d_m}$

$Q \leftarrow EW_Q$

$K \leftarrow EW_K$

$V \leftarrow EW_V$

$A \leftarrow \text{softmax}_j \left( \frac{\text{mask}(QK^t)}{\sqrt{d_m}} \right) V$

$E' \leftarrow E + A$

---

Consiste en ejecutar en paralelo  $h$  cabezas de atención, cada una con sus respectivas matrices  $W_Q^{(i)}$ ,  $W_K^{(i)}$ , y  $W_V^{(i)}$ , para obtener  $h$  matrices  $A^{(i)}$ . De esta manera, se consigue que cada cabeza aprenda los diferentes contextos que puede adoptar un token, para finalmente entre todas las cabezas, elegir el adecuado. Esto se ve reflejado en el Algoritmo 2.5.

**Algoritmo 2.5:** Atención múltiple

**Datos:**  $E \in \mathbb{R}^{n \times d_m}$ ,  $W_Q^{(i)} \in \mathbb{R}^{d_k \times d_m}$ ,  $W_K^{(i)} \in \mathbb{R}^{d_k \times d_m}$ ,  $W_V^{(i)} \in \mathbb{R}^{d_v \times d_m}$ ,  $W_O \in \mathbb{R}^{hd_v \times d_m}$ ,  $h$

**Resultado:**  $E' \in \mathbb{R}^{n \times d_m}$

$$d_k \leftarrow \frac{d_m}{h}$$

$$d_v \leftarrow d_k$$

paralelamente hacer

$$\begin{cases} Q^{(i)} \leftarrow E \left( W_Q^{(i)} \right)^t \\ K^{(i)} \leftarrow E \left( W_K^{(i)} \right)^t \\ V^{(i)} \leftarrow E \left( W_V^{(i)} \right)^t \\ A^{(i)} \leftarrow \text{softmax}_j \left( \frac{\text{mask} \left( Q^{(i)} (K^{(i)})^t \right)}{\sqrt{d}} \right) V^{(i)} \end{cases}$$

esperar

$$E' \leftarrow E + (A^{(1)} | A^{(2)} | \dots | A^{(h)}) W_O$$

El proceso de codificación continúa con una normalización de los embeddings. Para realizar dicha normalización[19], se supone que los elementos de cada embedding siguen una distribución  $\mathcal{N}(\mu_i, \sigma_i)$ , y se normalizan de acuerdo a la ecuación  $E' = \Sigma \Gamma \odot (E - M) + B$ , es decir,

$$E' = \begin{pmatrix} \sqrt{\sigma_1^2 + \varepsilon}^{-1} & 0 & \dots & 0 \\ 0 & \sqrt{\sigma_2^2 + \varepsilon}^{-1} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \sqrt{\sigma_n^2 + \varepsilon}^{-1} \end{pmatrix} \begin{pmatrix} \gamma_{11} & \gamma_{12} & \dots & \gamma_{1d_m} \\ \gamma_{21} & \gamma_{22} & \dots & \gamma_{2d_m} \\ \vdots & \vdots & \ddots & \vdots \\ \gamma_{n1} & \gamma_{n2} & \dots & \gamma_{nd_m} \end{pmatrix} \odot \begin{pmatrix} e_{11} - \mu_1 & e_{12} - \mu_1 & \dots & e_{1d_m} - \mu_1 \\ e_{21} - \mu_2 & e_{22} - \mu_2 & \dots & e_{2d_m} - \mu_2 \\ \vdots & \vdots & \ddots & \vdots \\ e_{n1} - \mu_n & e_{n2} - \mu_n & \dots & e_{nd_m} - \mu_n \end{pmatrix} + \begin{pmatrix} \beta_{11} & \beta_{12} & \dots & \beta_{1d_m} \\ \beta_{21} & \beta_{22} & \dots & \beta_{2d_m} \\ \vdots & \vdots & \ddots & \vdots \\ \beta_{n1} & \beta_{n2} & \dots & \beta_{nd_m} \end{pmatrix}$$

donde las matrices  $\Gamma$  y  $B$  son parámetros del modelo, y  $\varepsilon$ , ruido para evitar divisiones por cero. Se pretende que cada valor siga una distribución  $\mathcal{N}(0, 1)$ , quedando multiplicado y sumado por los parámetros de ganancia y sesgo. Para finalizar la etapa de la codificación, cada embedding normalizado pasa por una red neuronal clásica y se vuelve a normalizar la suma de la entrada y la salida de la red.

### 2.4.2. Decodificación

La etapa de decodificación es bastante similar a la de codificación, ya que la misión del decodificador se puede resumir de manera informal en mezclar las codificaciones del codificador, y las codificaciones de la entrada del decodificador. Todos los bloques del decodificador que se muestran en dicha Figura 2.14 ya han sido explicados a falta de dos. El más simple es el que aparece como *Linear*, que no son más que redes neuronales que traducen embeddings a tokens.

Por último, aparece una capa de atención múltiple que es algo diferente de la ya explicada, ya que en realidad está calculando la atención múltiple-cruzada (Algoritmo 2.6). Este cálculo es idéntico al de la atención múltiple, pero en vez de utilizar los embeddings del decodificador ( $E_2$ ) para calcular las consultas, claves, y valores; se utilizan solo para calcular las consultas. Para calcular las claves y valores se utilizan los embeddings de la salida del codificador ( $E_1$ ), pues de esta manera “preguntan” a los del decodificador para encontrar las relaciones entre los tokens de entrada del codificador y los del decodificador.

**Algoritmo 2.6:** Atención múltiple-cruzada

---

**Datos:**  $E_1 \in \mathbb{R}^{n \times d_m}$ ,  $E_2 \in \mathbb{R}^{n \times d_m}$ ,  $W_Q^{(i)} \in \mathbb{R}^{d_k \times d_m}$ ,  $W_K^{(i)} \in \mathbb{R}^{d_k \times d_m}$ ,  $W_V^{(i)} \in \mathbb{R}^{d_v \times d_m}$ ,  $W_O \in \mathbb{R}^{hd_v \times d_m}$ ,  $h$

**Resultado:**  $E'_2 \in \mathbb{R}^{n \times d_m}$

$$d_k \leftarrow \frac{d_m}{h}$$

$$d_v \leftarrow d_k$$

paralelamente hacer

$$\begin{cases} Q^{(i)} \leftarrow E_2 \left( W_Q^{(i)} \right)^t \\ K^{(i)} \leftarrow E_1 \left( W_K^{(i)} \right)^t \\ V^{(i)} \leftarrow E_1 \left( W_V^{(i)} \right)^t \\ A^{(i)} \leftarrow \text{softmax}_j \left( \frac{\text{mask} \left( Q^{(i)} (K^{(i)})^t \right)}{\sqrt{d}} \right) V^{(i)} \end{cases}$$

esperar

$$E'_2 \leftarrow E_2 + (A^{(1)} | A^{(2)} | \dots | A^{(h)}) W_O$$


---

### 2.4.3. Ejecución y entrenamiento

Esta arquitectura fue planteada originalmente para lograr la traducción de textos de un idioma a otro, aunque como se comentará más adelante, es aplicable a diversas tareas. Suponiendo que se quiere traducir el texto “Juan es alto”, el flujo de ejecución es el siguiente.

1. Se obtienen los tokens `$SOS$, Juan, es, alto, $EOS$`
2. El codificador genera una matriz  $E_1 \in \mathbb{R}^{5 \times d_m}$  y se envía al decodificador.
3. En la entrada del decodificador se introduce el token `$SOS$`, y a la salida deberían aparecer los tokens `$SOS$, John`.
4. En la entrada del decodificador se introducen los tokens generados por el propio decodificador `$SOS$` y `John`, y a la salida deberían aparecer los tokens `$SOS$, John, is`.
5. Se continúa introduciendo la salida del decodificador en su entrada, hasta obtener en la salida el token `$EOS$`.

Respecto al entrenamiento, se utilizan las mismas técnicas que en las secciones anteriores, Backpropagation con las funciones de error y optimizadores adecuados, normalmente entropía cruzada y Adam, pues se quiere comparar el token esperado con el predicho. El motivo de utilizar Backpropagation es que un transformer no deja de ser una sucesión de redes neuronales. Por otro lado, añadir que para cada ejemplo de traducción se obtienen diferentes ejemplos con los que entrenar, pues para predecir el token  $i$ , se necesita predecir correctamente el token  $i - 1$ , por lo que la predicción hasta cada token sirve como un ejemplo de entrenamiento diferente, lo que permite tener una gran cantidad de ejemplos.



## Capítulo 3

# Optimización del proceso de valoración de puntos de interés

Después de analizar diversos modelos de aprendizaje automático, junto con sus características y los algoritmos asociados, se ofrece un ejemplo práctico que ilustra la aplicación de estos modelos para abordar un problema de la vida real y buscar una solución adecuada. Para ello se presenta la compañía Niantic, fundada en 2010 como parte de una startup de Google, que se especializa en el desarrollo de juegos para dispositivos móviles que utilizan realidad aumentada (AR). Algunas de sus creaciones más destacadas han sido los juegos Ingress y Pokémon GO.

Una de las herramientas creadas por esta empresa es Niantic Lightship, que permite a desarrolladores Unity integrar realidad aumentada y mapas con puntos de interés basados en la ubicación real del jugador. Dado que para Niantic resultaba inviable marcar dichos puntos de interés alrededor de todo el mundo, creó Niantic Wayfarer. En esta herramienta, usuarios experimentados de sus juegos pueden hacer propuestas de puntos de interés (llamados Wayspots) para que de manera colaborativa, otros usuarios las valoren. Sin embargo, tras varios años desde su lanzamiento, debido al gran número de propuestas y al reducido número de valoradores, la comunidad notifica largos tiempos de espera en el proceso de valoración de las propuestas. Por este motivo, se propone en este proyecto realizar una primera aproximación a la automatización de este proceso mediante las técnicas de visión e inteligencia artificial presentadas en el marco teórico.

La valoración de un Wayspot en Wayfarer consta de diferentes etapas. En primer lugar se muestra un título y descripción del Wayspot junto con la imagen que aparecería en los juegos. Además aparece una imagen secundaria con una visión desde otra perspectiva, junto con otro texto que ayudarían al valorador a ubicar la propuesta, tal y como se muestra en la Figura 3.1. También aparece un breve cuestionario con una serie de preguntas genéricas que ayudan a determinar si aquello que se solicita cumple una serie de criterios (Figura 3.2). Además, se muestra un mapa que contiene los Wayspots cercanos para verificar que no exista ya, y comprobar con Street View la ubicación sugerida (Figura 3.3). Finalmente, se pide clasificar la propuesta en una o varias categorías.

### 3.1. Tecnologías empleadas

#### 3.1.1. Python

Para realizar la codificación del proyecto se ha utilizado el lenguaje de programación Python. Es un lenguaje de alto nivel en el que se pueden seguir diferentes paradigmas de programación, como por ejemplo, el orientado a objetos o el funcional. Sus dos grandes ventajas son su sintaxis, que es muy sencilla y legible; y la gran variedad de bibliotecas que posee para realizar cualquier tipo de tareas, desde desarrollo web, proyectos de aprendizaje automático, desarrollo de videojuegos, creación de animaciones, etc.

**Información de la propuesta**  
Consulta a continuación la información de la propuesta

**Título y descripción**  
El título y la descripción no deben contener texto inapropiado. La descripción no es obligatoria.

**Estatua Víctimas del terrorismo**

Estatua del escultor Juan Carlos Díaz Durán en honor a las Víctimas del Terrorismo.

Foto  
Las fotos deben ser nítidas y aportar una vista clara de la ubicación al completo.

Dirección de la calle: C. Víctimas del Terrorismo, 15, 28390 Belmonte de Tajo, Madrid, España

**Información complementaria**  
Información adicional enviada por el jugador sobre por qué esto debería ser un Wayspot.

Es una escultura para visitar en una zona peatonal con bancos.

Figura 3.1: Información de una propuesta en Wayfarer

<b>Apropiado</b> ¿Es el lugar apropiado para ser visitado por peatones? <input type="button" value=""/>	<input type="button" value=""/>	<input type="button" value=""/>	<input type="button" value="NO LO SÉ"/>
<b>Seguro</b> ¿Es seguro el acceso peatonal y público a este lugar? <input type="button" value=""/>	<input type="button" value=""/>	<input type="button" value=""/>	<input type="button" value="NO LO SÉ"/>
<b>Exactitud</b> ¿Los detalles que acompañan a esta propuesta son correctos e informativos? <input type="button" value=""/>	<input type="button" value=""/>	<input type="button" value=""/>	<input type="button" value="NO LO SÉ"/>
<b>Permanente y distinto</b> ¿Es un lugar o un objeto permanente, físico, tangible e identificable que marca una zona? <input type="button" value=""/>	<input type="button" value=""/>	<input type="button" value=""/>	<input type="button" value="NO LO SÉ"/>
<b>Socializar</b> ¿Es un buen lugar para conocer gente y socializar? <input type="button" value=""/>	<input type="button" value=""/>	<input type="button" value=""/>	<input type="button" value="NO LO SÉ"/>
<b>Ejercicio</b> ¿Es un buen lugar para practicar una actividad física ligera? <input type="button" value=""/>	<input type="button" value=""/>	<input type="button" value=""/>	<input type="button" value="NO LO SÉ"/>
<b>Explorar</b> ¿Crees que esta ubicación es importante para la comunidad local? <input type="button" value=""/>	<input type="button" value=""/>	<input type="button" value=""/>	<input type="button" value="NO LO SÉ"/>

Figura 3.2: Cuestionario de una propuesta en Wayfarer

¿Es alguna de estas ubicaciones un duplicado de este Wayspot?

Mapa Satélite

Ermita De San Isidro

Combinaciones de teclas

Carta del mapa ©2024 Inst. Geogr. Nacional

Notificar un problema de Mapas

SUGERIR UNA NUEVA UBICIÓN

¿Son las categorías seleccionadas apropiadas?

Sí  No  Objeto (monumento)

Sí  No  Escultura

Sí  No  Estatua

Añadir nuevas categorías

Accidente geográfico >

Arte >

Bar >

Campo de atletismo >

Cancha deportiva >

Comercial >

Figura 3.3: Mapa y clasificación de una propuesta de Wayfarer

Para este proyecto, las principales utilizadas son TensorFlow y Scikit-learn. La primera de ellas fue creada por Google y permite construir de manera sencilla modelos de aprendizaje automático, contando con las im-

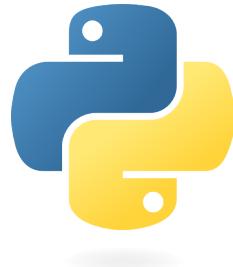
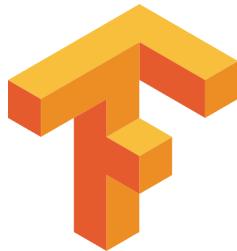


Figura 3.4: Python

plementaciones de los principales elementos que los forman, así como algoritmos de entrenamiento, datasets de ejemplo, y funciones auxiliares de utilidad para trabajar con los datos empleados durante el proceso. La segunda de ellas, además de contar también con diversas implementaciones de modelos y algoritmos de aprendizaje automático, es bien conocida por contar con las implementaciones de diferentes métricas para modelos, lo que permite comprobar de manera rápida y visual la calidad de los modelos entrenados. Finalmente, estas bibliotecas hacen uso de otras como pueden ser Matplotlib, utilizada para generar cualquier tipo de figura desde Python.



(a) TensorFlow



(b) Scikit-learn

Figura 3.5: Bibliotecas populares de Python

Por otro lado, no todo son ventajas. Python es un lenguaje interpretado y de tipado dinámico, lo que hace que su ejecución sea mucho más lenta que la de otros lenguajes. Sin embargo, existen soluciones como NumPy, una biblioteca que permite trabajar con grandes matrices de datos en tiempos de ejecución menores que al utilizar listas, gracias a que gran parte de su codificación se realizó utilizando lenguajes de más bajo nivel como C.

### 3.1.2. CUDA

CUDA son las siglas de Compute Unified Device Architecture, y es una tecnología de la marca estadounidense NVIDIA. Esta tecnología hace referencia a un conjunto de elementos que permiten ejecutar código de manera masivamente paralela en una tarjeta gráfica de la compañía, de manera que el tiempo de ejecución es mucho menor que al ejecutarlo de manera secuencial. Entre estos elementos se encuentran el controlador, el compilador, las interfaces para los diferentes lenguajes (la más común la de C/C++), las librerías que contienen algoritmos optimizados, y los propios núcleos CUDA de la GPU[20].

El hardware de las tarjetas gráficas de NVIDIA se divide en *streaming multiprocessors* (SMs), que a su vez están compuestos de *streaming processors* (SPs) o núcleos CUDA, entre otros elementos. A nivel de software, la unidad básica paralela es el hilo, que a nivel de hardware se ejecuta en un SP. Dichos SPs, son unos núcleos capaces de hacer operaciones más sencillas que las que un núcleo de una CPU podría realizar, pero de manera más rápida. A su vez, los hilos se agrupan en bloques, teniendo en cuenta que a nivel de hardware cada bloque se ejecuta en un único SM. Finalmente, todos los elementos se agrupan en una cuadrícula (o

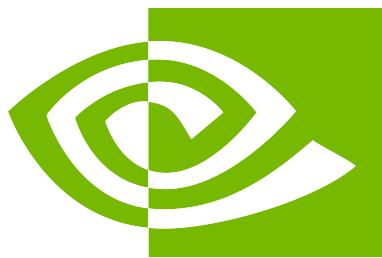


Figura 3.6: NVIDIA

*grid*) compuesta de bloques. Cada cuadrícula se ejecuta en una única GPU a nivel de hardware, y ejecuta a nivel de software un kernel[21], [22].

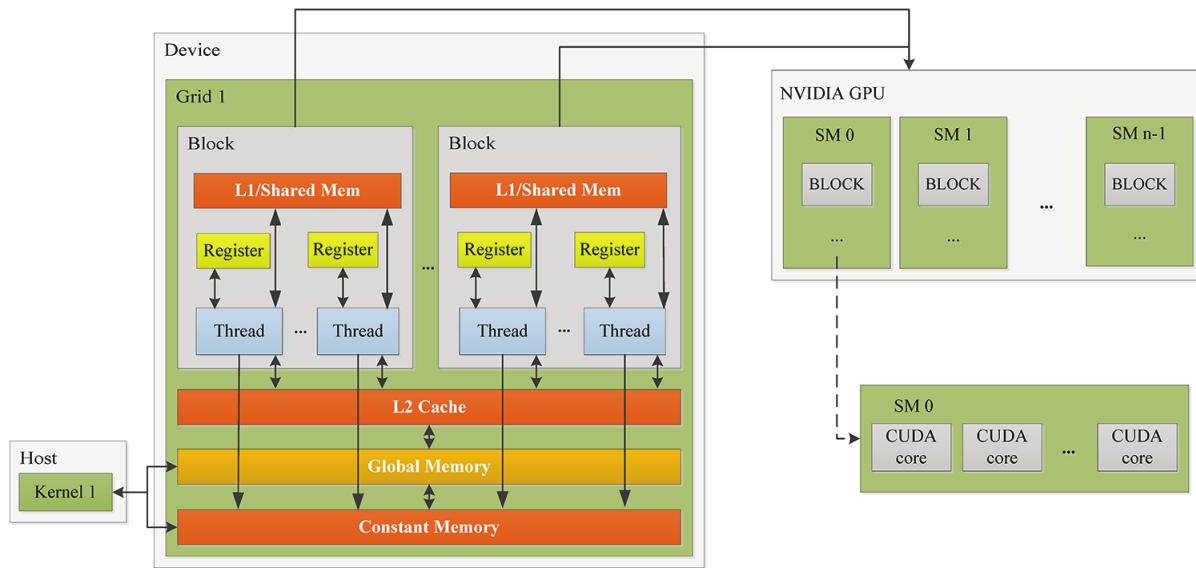


Figura 3.7: Arquitectura CUDA a nivel de hardware y software[22]

El hecho de poder realizar miles de operaciones sencillas de manera simultánea, reduce enormemente el tiempo de ejecución de algoritmos que son paralelizables, es decir, que contienen pasos sencillos que son independientes de otros y se pueden realizar al mismo tiempo, como es el caso de una convolución o multiplicación de matrices. Además, se cuenta con diferentes niveles de memoria, con los que se agilizan los tiempos de lectura de elementos que son utilizados frecuentemente.

La mayoría de algoritmos de aprendizaje automático de TensorFlow tienen implementada su versión paralela, de manera que si se cuenta con una GPU compatible con CUDA, TensorFlow la utilizará para ahorrar el mayor tiempo posible durante los entrenamientos de los modelos. En este caso se dispone de un equipo que cuenta con un procesador Intel Core i7-8700, una GPU NVIDIA GeForce GTX 1060 6GB, y 32 GB de RAM; lo que hace posible la ejecución concurrente de dichos algoritmos. Esta tarjeta gráfica cuenta con 1280 SPs o núcleos CUDA repartidos en 10 SMs.

Si bien el trabajo de indicar a TensorFlow que haga uso de CUDA es sencillo, no es tan trivial la configuración de drivers de NVIDIA, instalación de CUDA, versión de Python, versión de TensorFlow, versión de sistema operativo, etc. Con el frecuente cambio de versiones de cada componente, se presentan nuevas incompatibilidades entre ellos, haciendo que guías oficiales fallen y que el proceso de configuración inicial sea complicado en sistemas operativos como Windows 11. La manera más sencilla actualmente de realizar

una instalación de CUDA y TensorFlow es sobre el sistema operativo Ubuntu en su versión 22.04 LTS. Para instalar y configurar el resto de componentes se ha hecho uso de LambdaStack (<https://lambdalabs.com/lambda-stack-deep-learning-software>). Mediante el uso del comando de terminal

```
wget -nv -O- https://lambdalabs.com/install-lambda-stack.sh | sh -
```

automáticamente lanza LambdaStack que se encarga de instalar las versiones de Python y sus librerías (TensorFlow y PyTorch entre otras) adecuadas, los drivers de NVIDIA correctos, componentes de CUDA, realiza las configuraciones de variables de entorno, etc. En general, deja el entorno de Ubuntu configurado para poder trabajar en cualquier proyecto de inteligencia artificial en el que se quiera hacer uso de una GPU de NVIDIA. También cuentan con una imagen de Docker que ofrece una experiencia similar. El uso de este servicio es completamente gratuito, y es el que utiliza la compañía que lo gestiona en sus servicios de cloud computing (que sí son de pago).

Para probar la eficacia del uso de CUDA para proyectos de esta índole, se ha seleccionado un modelo de los que se presentan en la Sección 3.2.2 (entrenamiento de una red convolucional desde cero), y se ha lanzado midiendo el tiempo, el mismo entrenamiento pero con diferente número de épocas sobre CPU y GPU. La Figura 3.8 evidencia cómo el entrenamiento en la GPU llega a ser hasta cinco veces más rápido que en la CPU. Todos los entrenamientos de la Sección 3.2.2 han sido realizados sobre la GPU.

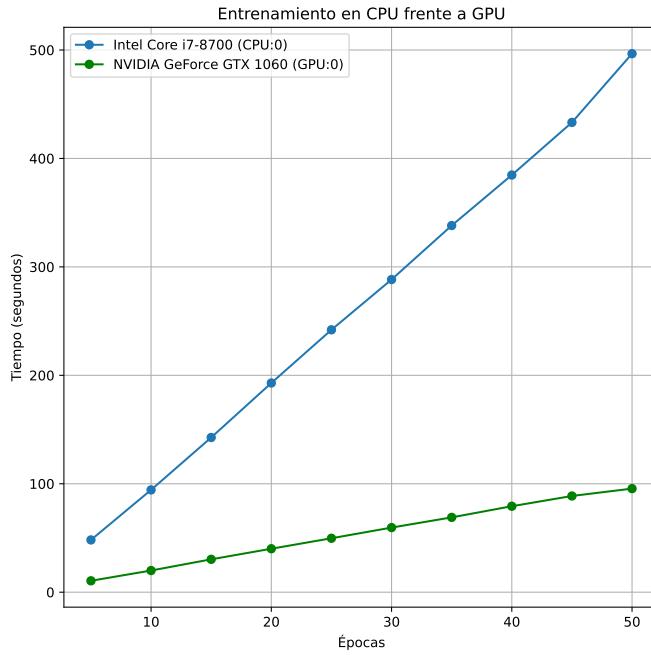


Figura 3.8: Comparativa de entrenamiento en CPU frente a en GPU

## 3.2. Clasificación de imágenes mediante redes convolucionales

Una primera aproximación a agilizar el proceso de valoración comentado, sería detectar el objeto que se muestra en la imagen de la propuesta, para que en caso de que sea algo aceptable proseguir con el proceso de valoración, o en caso contrario, rechazar directamente la propuesta. En general, en función del objeto que aparezca en la imagen, se tienen directamente las respuestas a las preguntas de la Figura 3.2.

Para comenzar este caso práctico, lo primero a realizar es el proceso conocido como ETL, que consiste en la extracción, transformación, y carga de los datos; para posteriormente poder trabajar con ellos y proporcionárselos al modelo. Comenzando por la extracción de datos, se presenta el primero de los problemas.

La idea es utilizar imágenes que realmente hayan pasado por este proceso de valoración para poder hacer el proyecto lo más realista posible, sin embargo, ni Wayfarer ni ninguno de los juegos poseen alguna API (al menos de manera pública) que permita recolectar de manera programática las imágenes utilizadas o información relativa a ellas.

### 3.2.1. Proceso ETL

La solución adoptada para este proceso, ha sido recolectar manualmente imágenes que han superado el proceso de valoración, pudiendo consultar algunas de ellas desde el mapa de uno de los juegos (<https://intel.ingress.com/>). En este caso, se estarían clasificando entre las  $n$  clases de objetos aceptables las imágenes recibidas, cosa que en un primer momento parece carecer de sentido pues se conoce el resultado de la valoración. Sin embargo, al no tener acceso a propuestas rechazadas, no se pueden recolectar estos datos para entrenar los modelos, pero si de la empresa responsable se tratase, se dispondría de una enorme cantidad de imágenes válidas y no válidas etiquetadas (gracias a la parte del proceso de valoración que muestra la Figura 3.3), y que se podrían cargar de manera automática. En resumen, cambiando simplemente los datos que se cargarían y su fuente, se podrían tomar las siguientes decisiones sin necesidad de modificar el resto del proyecto.

- Si  $I \in C_i, 0 \leq i < m$ , rechazar la propuesta
- Si  $I \in C_j, m \leq j < n$ , continuar evaluando la propuesta

Continuando con la extracción de los datos, y preparándolos para la carga, se ha creado una carpeta `tfg_dataset` que representa el conjunto de las imágenes que se utilizan durante el proyecto. Dentro de esta, se ubicarán dos subcarpetas, `train` y `test`, que hacen referencia a las imágenes que se utilizarán para entrenar los modelos, y las que se utilizarán para evaluar su rendimiento. Para generar dichas carpetas partiendo de una que contiene las imágenes separadas por clases, se ha codificado la función `split_tt` que permite hacer la división del conjunto en los de entrenamiento y test con el porcentaje especificado. Previamente a la ejecución de esta función, se recomienda ejecutar otra que se ha codificado llamada `renombrar_imagenes`, que dada la ruta raíz donde se encuentran las carpetas de cada clase, renombra las imágenes de cada carpeta de manera que queden enumeradas. Esta es la manera de etiquetar el conjunto de datos y prepararlo para la carga. Se puede visualizar un breve esquema de esta estructura en la Figura 3.9.

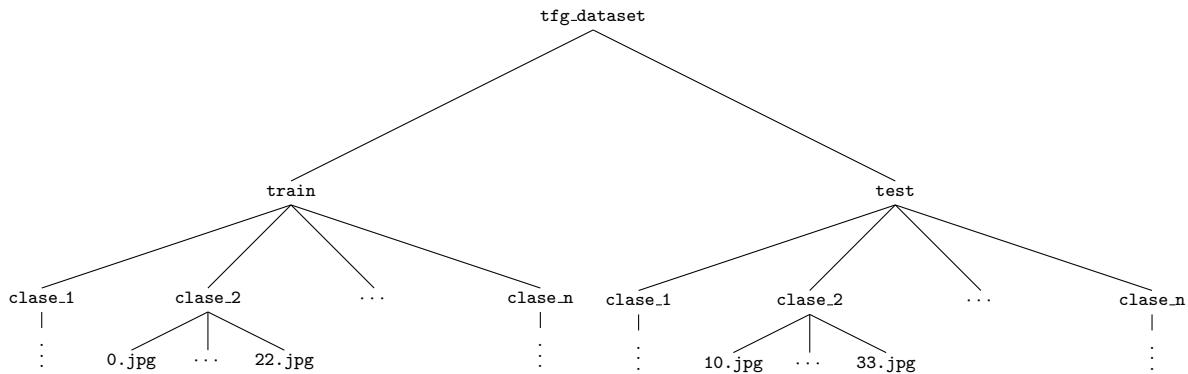


Figura 3.9: Árbol de carpetas del dataset

Para cargar el dataset en TensorFlow directamente desde la estructura de carpetas creadas, se hará uso de la función `image_dataset_from_directory` de `utils`. Esta contiene una serie de parámetros interesantes a comentar.

- `directory`: es el directorio raíz del dataset, en este caso `tfg_dataset`.
- `image_size`: es una tupla de dos elementos con las dimensiones en píxeles que deberán tener las imágenes del dataset.



Figura 3.10: Visualización de ejemplo de un dataset creado

- **labels**: mediante el valor `inferred` las etiquetas toman el mismo valor que el nombre de las carpetas.
- **label\_mode**: hace referencia a la forma de codificar las etiquetas. Se empleará el valor `categorical` para codificar las etiquetas utilizando one-hot-encoding, es decir, si se tienen por ejemplo cuatro clases y un elemento pertenece a la cuarta, dicha etiqueta queda codificada como 0001.
- **batch\_size**: hace referencia al tamaño de batch o lote que será utilizado durante el entrenamiento. Si en la función de entrenamiento se elige un tamaño diferente, se utilizará el menor de los valores.
- **validation\_split**: hace referencia al porcentaje de los datos de entrenamiento que se reservan para validar el modelo durante el entrenamiento, es decir, permiten calcular el error del modelo al hacer una predicción de datos que nunca ha visto mientras entrena. Se reservarán un 20 % de los datos para validar.
- **seed**: hace referencia a la semilla que se utiliza para ordenar las imágenes de manera aleatoria.
- **subset**: seleccionando el valor `both` permite devolver el conjunto de entrenamiento y validación con la llamada a la función, es decir, `x_train, x_val = image_dataset_from_directory(...)`.

Como se puede observar, esta función que está haciendo principalmente el trabajo de la carga de datos, también hace parte del proceso de transformación, ya que permite modificar las dimensiones de la imagen (en este caso se utilizará un valor de  $224 \times 224$  píxeles, se justificará más adelante), y también permite modificar la codificación de las etiquetas. Además, cada píxel (en cada canal de color) toma un valor entre 0 y 255. Es muy importante normalizar estos valores en el proceso de transformación para facilitar el trabajo a los diferentes modelos. Para ello, con ayuda de una capa de reescalado de Keras, se divide el valor de cada píxel entre 255 para obtener valores entre 0 y 1. Con motivo de verificar que todos estos pasos se han realizado correctamente, se codificó una función llamada `sample_ds_dfd` que permite visualizar nueve ejemplos aleatorios de un dataset creado con la función de TensorFlow mencionada, obteniendo como resultado la Figura 3.10.

### 3.2.2. Creación y entrenamiento de una red convolucional en TensorFlow

En esta primera aproximación se va a crear y entrenar una red convolucional desde cero con el dataset mencionado. En este caso, se han elegido cuatro tipos de objetos que frecuentemente se proponen como puntos de interés: parques infantiles, carteles informativos, marcadores de ruta, e hitos del Camino de Santiago. Mediante la clase `Sequential` de Keras, se puede proporcionar una lista de las capas que conforman el modelo. La primera de ellas será una capa de convolución, con la particularidad de que se debe indicar el tamaño de entrada, siendo este un tensor de las dimensiones indicadas en la creación del dataset. Se alterna cada capa de convolución ReLU de stride  $3 \times 3$  y 32 o 64 filtros, con una capa de maxpooling con stride  $2 \times 2$ . Estos valores han sido elegidos de manera arbitraria. Esta arquitectura se puede visualizar en la Figura 3.11.

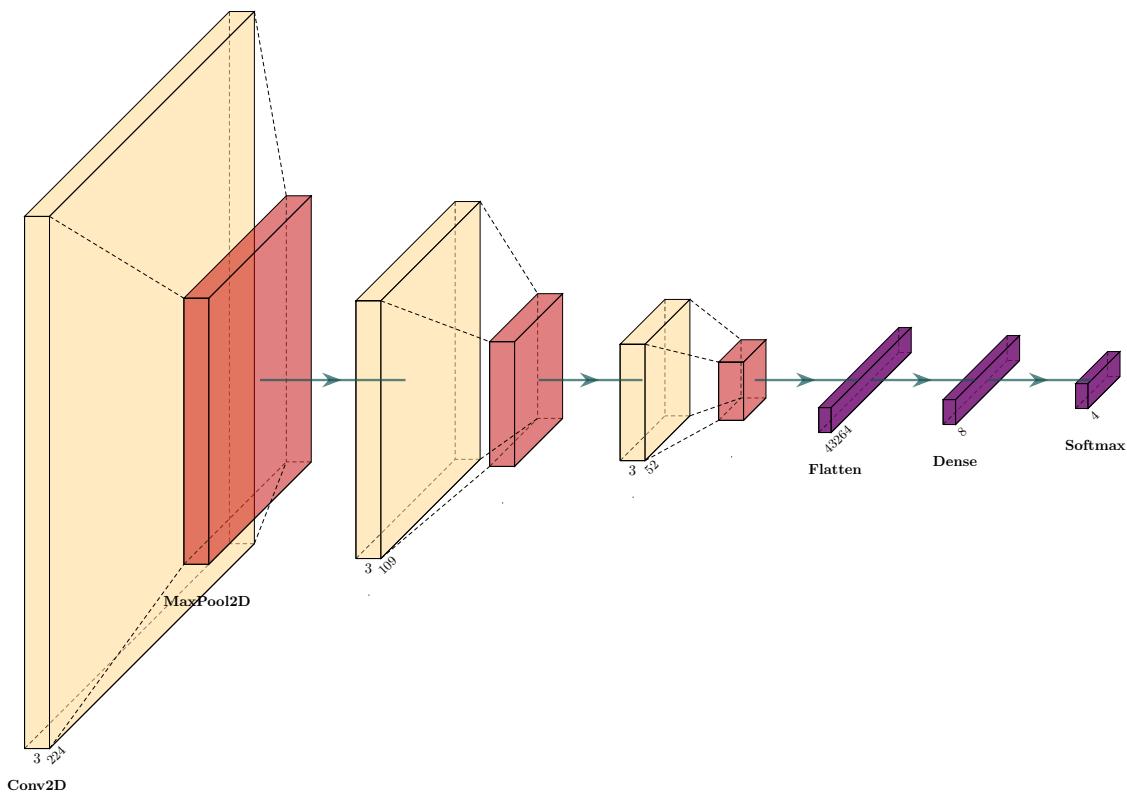


Figura 3.11: Arquitectura de la red convolucional

Mediante estas capas, se supone que la red debe extraer características de las imágenes, como por ejemplo

- Aparece un objeto rectangular
- Tiene dos patas
- No tiene formas curvas

y de ahí con ayuda de una red clásica, ser capaz de deducir que en la imagen aparece un cartel. En realidad, las capas de convolución no obtienen características tan claras, pero sí resultan con una matriz de detalles en la imagen que pueden conducir a la misma conclusión. Para ello se utiliza la capa `Flatten` que transforma dicha matriz en un vector que sirve de entrada a la siguiente y última capa de la red, una capa oculta con tantas neuronas como clases diferentes. Como función de activación se utiliza softmax para obtener una

distribución de probabilidad en la que se observe cuál es claramente la clase a la que pertenece el objeto, y con qué confianza lo es. Finalmente, mediante la función `summary` se puede observar un resumen del modelo.

Layer (type)	Output Shape	Param #
<hr/>		
conv2d (Conv2D)	(None, 222, 222, 32)	896
max_pooling2d (MaxPooling2D)	(None, 111, 111, 32)	0
conv2d_1 (Conv2D)	(None, 109, 109, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 54, 54, 64)	0
conv2d_2 (Conv2D)	(None, 52, 52, 64)	36928
max_pooling2d_2 (MaxPooling2D)	(None, 26, 26, 64)	0
flatten (Flatten)	(None, 43264)	0
dense_2 (Dense)	(None, 8)	346120
dense_3 (Dense)	(None, 4)	36
<hr/>		
Total params:	402476 (1.54 MB)	
Trainable params:	402476 (1.54 MB)	
Non-trainable params:	0 (0.00 Byte)	

---

Con la función `compile` se termina de crear el objeto que representa al modelo, pudiendo especificar un optimizador, la función de pérdida, y las métricas que se muestran durante el entrenamiento. Para este proyecto se utilizarán Adam, entropía cruzada, y precisión respectivamente. El modelo ya está en condiciones de ser entrenado, y esto se logra mediante la función `fit`, a la que se le proporcionan los datos de entrenamiento (`x_train`), los datos de validación (`x_val`), el número de épocas (en este caso se han elegido 25), y la ruta de los callbacks. Esta sirve para ir almacenando metadatos del entrenamiento, que mediante una utilidad con la que cuenta TensorFlow llamada TensorBoard, permite monitorizar de manera muy visual la calidad del entrenamiento, tal y como se muestra en la Figura 3.12.

La situación que describen las gráficas, es negativa y sería la típica no deseada, pues se observa como con el paso de las iteraciones el error crece mientras que la precisión se estanca en valores no deseados. En concreto, en la Figura 3.12c se observa cómo el error sobre el conjunto de validación crece, mientras que sobre el conjunto de entrenamiento tiende a cero. Esta es una situación denominada como sobreaprendizaje o *overfitting*[23]. La red no tiene capacidad de aprender ni generalizar, y lo que está haciendo es memorizar los datos de ejemplo que se le presentan, cometiendo entonces errores cuando recibe datos que no ha visto nunca.

### 3.2.3. Transfer learning en TensorFlow

Como se ha podido observar durante la sección anterior, el resultado del entrenamiento no ha sido adecuado, pues la red tenía la memoria de los datos que se le presentaban sin mostrar una buena capacidad de generalización. Esto puede deberse a múltiples factores, como por ejemplo, que al tener un conjunto de datos

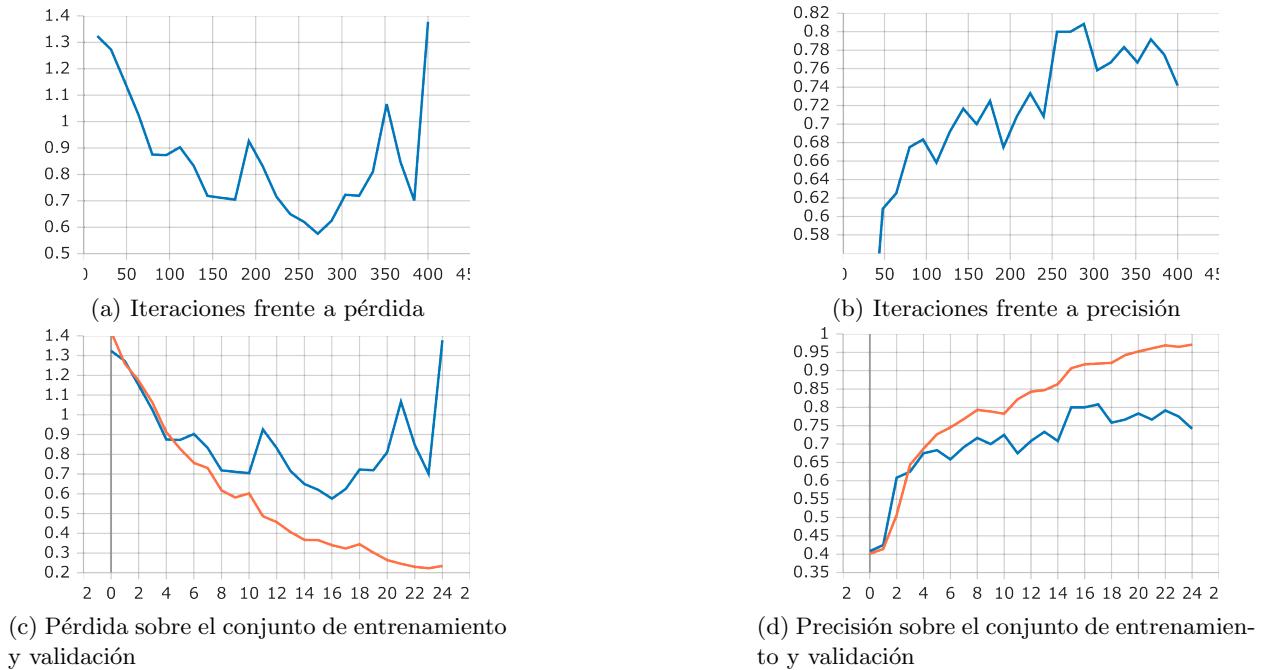


Figura 3.12: Pérdida y precisión durante el entrenamiento de la CNN

de entrenamiento tan reducido no sea capaz de extraer correctamente las características que determinan a los objetos de cada clase. Es decir, en vez de hacerle llegar a las capas densas hechos como “tiene un objeto rectangular”, “tiene dos patas”, etc; podría estar haciéndole llegar, “hay dos árboles en el fondo”. Observando la estructura del modelo, se está tratando de optimizar cerca un millón de parámetros con poco más de 500 observaciones, algo que es muy desproporcionado; la red no es capaz de encontrar relaciones con tan pocos datos.

En proyectos similares del mundo real, y en los que además se dispone de pocos datos (como es el caso), es muy poco frecuente crear y entrenar un modelo desde cero tal y como se ha realizado en la Sección 3.2.2. En vez de realizar esto, es muy común utilizar una técnica conocida como transfer learning. Tal y como se menciona en [24], es una técnica muy aplicada en casos en los que se dispone de pocos datos y los modelos de deep learning no son capaces de encontrar relaciones, y también en casos en los que se dispone de equipos con pocos recursos. Consiste en aplicar el conocimiento obtenido de otro dataset, no necesariamente relacionado, para facilitar el proceso de obtener el conocimiento deseado del dataset actual.

En el problema que se está tratando, se va a aplicar esta técnica de la siguiente manera. Como previamente se ha mencionado, las redes convolucionales, pueden verse como un conjunto de capas capaces de extraer características de las imágenes, junto con una red neuronal clásica que es capaz de clasificar en función de dichas características. Teniendo en cuenta que el principal problema del modelo anterior podría ser que no fuera capaz de obtener las características adecuadas que definen a cada clase, la idea es utilizar una serie de capas que sí sean capaces de obtener de manera correcta las características de una imagen, para después poder entrenar la red neuronal clásica con las características adecuadas.

Con ayuda de la plataforma TensorHub que trabaja junto con TensorFlow, se pueden importar modelos ya entrenados. En este caso para aplicar la técnica de transfer learning, se ha decidido utilizar la red MobileNet de Google que se observa en la Figura 3.13. El modelo se puede importar como si de una capa se tratase al crear un modelo secuencial de Keras. Para lograr la técnica, será clave declarar que los parámetros de dicha capa no se deberán entrenar (pues al importar el modelo ya vienen con unos valores adecuados). A continuación se coloca una capa densa, encargada de clasificar las características que la red de Google

extraiga, y que deberá ser entrenada. Esta vez, el número de parámetros a optimizar sí es más adecuado al número de observaciones disponibles.

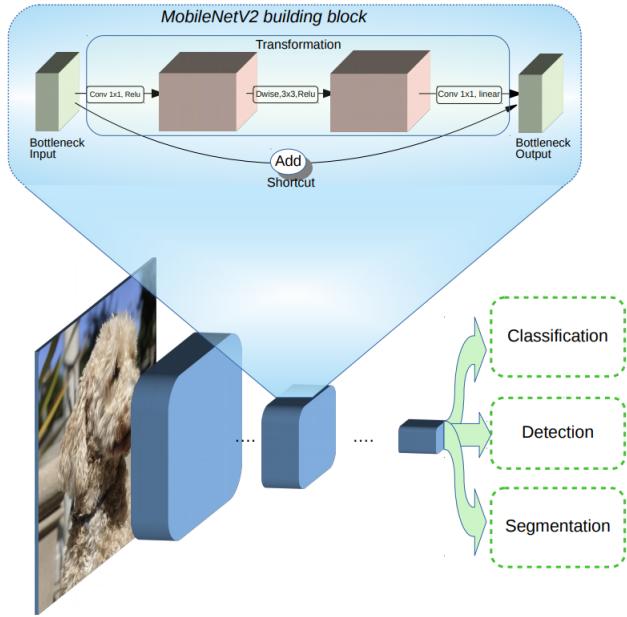


Figura 3.13: MobileNetV2

Layer (type)	Output Shape	Param #
<hr/>		
mobilenetv2_1.00_224 (Functional)	(None, 7, 7, 1280)	2257984
global_average_pooling2d (GlobalAveragePooling2D)	(None, 1280)	0
dense_2 (Dense)	(None, 8)	10248
dense_3 (Dense)	(None, 4)	36
<hr/>		
Total params: 2268268 (8.65 MB)		
Trainable params: 10284 (40.17 KB)		
Non-trainable params: 2257984 (8.61 MB)		
<hr/>		

La manera de lanzar el entrenamiento es la misma que en el anterior, sin embargo, esta vez los resultados obtenidos son totalmente distintos, tal y como muestran las gráficas de TensorBoard en la Figura 3.14.

Estas gráficas representan una situación cercana a la ideal. En las Figuras 3.14a y 3.14b se observa cómo con el paso de las iteraciones, el error disminuye y la precisión aumenta hasta llegar a valores adecuados. Además, en las Figuras 3.14c y 3.14d se muestra claramente que tanto el error como la precisión se comportan de manera similar sobre los conjuntos de entrenamiento y validación respectivamente, tomando además valores adecuados.

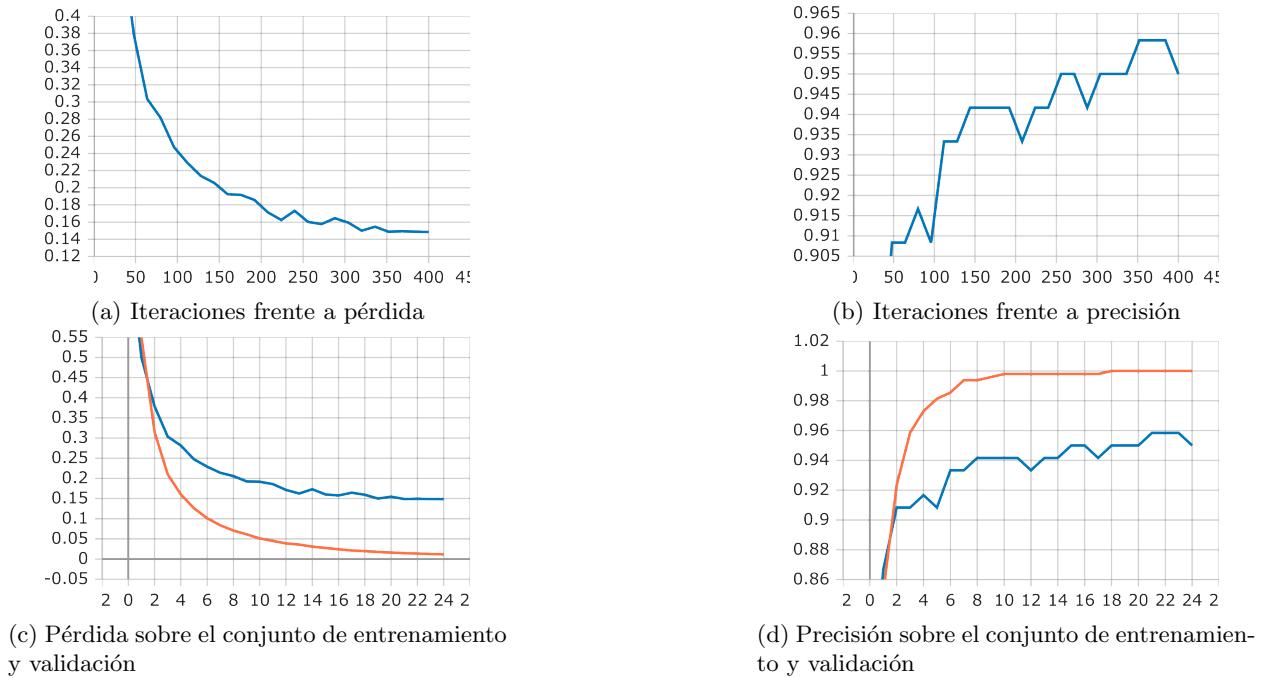


Figura 3.14: Pérdida y precisión durante el entrenamiento aplicando transfer learning

### 3.2.4. Aumento de datos

Como se ha visto a lo largo de las secciones anteriores, uno de los problemas que se está teniendo en este caso práctico, es la falta de datos. Una técnica utilizada frecuentemente en estos casos es conocida como aumento de datos. Consiste en modificar las imágenes del dataset de entrenamiento para que el modelo disponga de más ejemplos diferentes[25]. Podría entenderse como una parte del proceso de transformación ETL.

Una vez más, es posible aplicar esta técnica mediante funciones de TensorFlow. Para ello se crea un objeto de la clase `ImageDataGenerator`. Mediante su constructor se pueden declarar los valores que toman los atributos que modificarán las imágenes, siendo algunos de los más destacables:

- `rotation_range`: gira la imagen.
- `width_shift_range` y `height_shift_range`: desplazamiento horizontal y vertical de la imagen.
- `shear_range`: estira la imagen.
- `zoom_range`: hace zoom a la imagen.
- `rescale`: indica el factor por el que se reescaliza la imagen, en este caso  $1/255$  para facilitar el entrenamiento a la red, como en casos anteriores.
- `validation_split`: porcentaje de los datos reservados a validación.

Una vez se tiene declarado el generador, se utiliza su método `flow_from_directory` para que dada la ruta que contiene las imágenes, el tamaño deseado de imagen, y el tamaño de los lotes, se cree el objeto que representa al dataset. De manera similar a la anterior, se ha programado una función `sample_ds_ffd` que permite visualizar nueve elementos del dataset creado, tal y como muestra la Figura 3.15. Esta es una de la manera más común de crear datos nuevos, siendo alguna de las más novedosas las arquitecturas GAN. Están formadas por dos redes, una capaz de crear imágenes y otra capaz de distinguir entre imágenes reales y generadas, y transmitir dicho conocimiento a la primera red, de manera que una intente “engaños” a la otra[26].



Figura 3.15: Visualización de ejemplo de un dataset aplicando aumento de datos

Contando ahora con el dataset original y el modificado, la técnica del aumento de datos se puede aplicar de dos formas. La primera de ellas es crear un modelo y entrenar únicamente con el dataset modificado. Esto puede dar una mayor capacidad de generalización al modelo al mostrar imágenes más diferentes de objetos similares. La segunda de ellas consiste en entrenar el modelo con el dataset original, y después con el dataset modificado (sin cambiar qué imágenes pertenecen a los conjuntos de entrenamiento, validación, y test en cada dataset). Esta puede dar mejores resultados, pues tiene más observaciones y el modelo puede de ver el mismo objeto “de diferentes formas”. Sin embargo hay que tener cuidado con esta segunda opción, pues podría conducir a situaciones de sobreajuste.

### 3.2.5. Evaluación de los modelos

Si bien gracias a las gráficas que TensorBoard proporciona es posible hacerse una idea de la calidad que tendrán las predicciones de un modelo, no son suficientes. Será entonces el momento de presentar al modelo una serie de imágenes que nunca haya visto para poder evaluar la calidad de sus predicciones mediante una serie de métricas y valores estadísticos. Este conjunto de imágenes que el modelo no ha recibido hasta el momento, es el denominado conjunto de test.

Para calcular estas métricas en Python, se va a utilizar la librería Scikit-learn, ya que contiene un conjunto de métodos con los que calcular y visualizar la mayoría de métricas empleadas en machine learning. Además, las predicciones sobre los conjuntos de test quedarán representadas mediante dos matrices, ***Y\_matrix*** y ***Y\_score***. La primera de ellas es una matriz  $n \times 3$  que contiene para cada observación su etiqueta real, la predicha, y con qué probabilidad se le asigna. La segunda, contiene las probabilidades de cada observación de pertenecer a cada clase, por tanto es de dimensiones  $n \times c$ .

$$Y_m = \begin{pmatrix} 0 & 0 & 0,75 \\ 2 & 2 & 0,7 \\ 3 & 1 & 0,97 \\ \vdots & \vdots & \vdots \\ 1 & 1 & 0,87 \end{pmatrix} \quad Y_s = \begin{pmatrix} 0,75 & 0,07 & 0,01 & 0,17 \\ 0,15 & 0,01 & 0,7 & 0,14 \\ 0,02 & 0,97 & 0,01 & 0,01 \\ \vdots & \vdots & \vdots & \vdots \\ 0,05 & 0,87 & 0,01 & 0,08 \end{pmatrix}$$

Dependiendo si el problema en cuestión es de regresión o clasificación, se deben utilizar unas métricas u otras. Debido a que se desea evaluar la calidad de una clasificación, se han elegido las siguientes.

### Matriz de confusión

La matriz de confusión muestra en el caso de clasificación binaria, los verdaderos positivos, verdaderos negativos, falsos positivos, y falsos negativos, mientras que en el caso de clasificación no binaria, en general la relación entre las etiquetas reales de cada observación y las que el modelo le ha asignado[27]. De manera muy visual se puede observar la cantidad de ejemplos de cada clase (pudiendo ver si se encuentran desbalanceadas), cuántos han sido clasificados correctamente, cuántos no, y en general entre qué clases suele confundirse más el modelo.

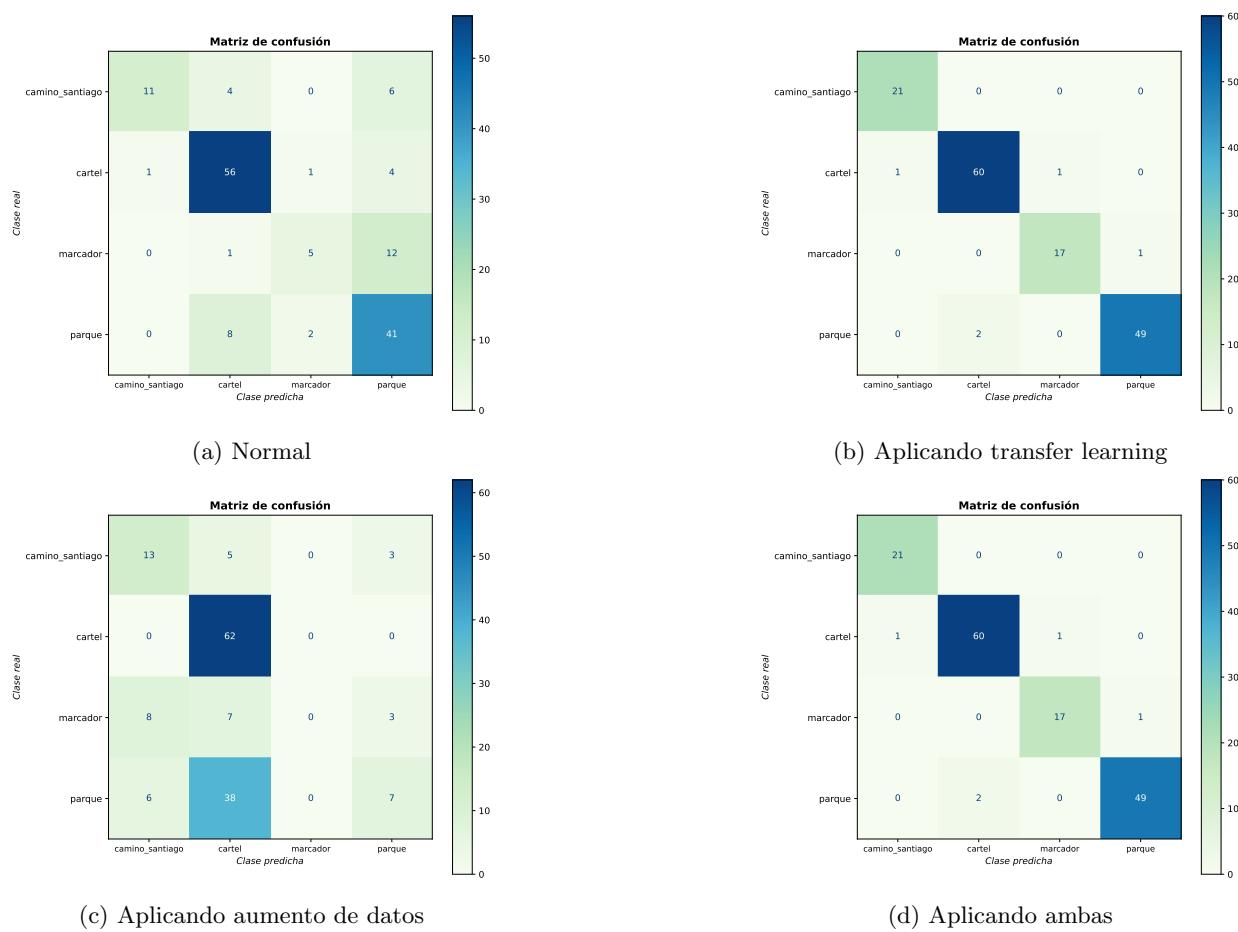


Figura 3.16: Matrices de confusión

En la Figura 3.16 se observan las matrices de confusión de las cuatro situaciones descritas anteriormente. El elemento  $a_{ij}$  representa la cantidad de observaciones de la clase  $C_i$  que han sido clasificadas como  $C_j$  (a partir de ahora,  $\hat{C}_j$ ). La situación ideal se da cuando en el caso de que  $i \neq j$ , entonces  $a_{ij} = 0$ , o lo que es lo mismo, que solo haya valores en la diagonal principal, pues a todos se les estaría asignando la clase correcta. Además, en esta representación gráfica sería deseable que todos los elementos de dicha diagonal tuvieran el mismo color, pues la situación actual describe un desbalanceo entre las clases. Se ve claramente que al aplicar transfer learning (Figuras 3.16b y 3.16d) se obtienen resultados mucho mejores que al entrenar el modelo desde cero, tal y como las curvas de TensorBoard parecían indicar, pues la cantidad de malas clasificaciones es muy pequeña. La diferencia en dicho caso entre usar aumento de datos o no, no es muy significativa.

Para poder calcular la matriz de confusión, se ha utilizado la función `confusion_matrix` de Scikit-learn, que recibe como parámetros un vector con las clases reales de cada observación, y otro con las que el modelo les ha asignado. Mediante la función `ConfusionMatrixDisplay` se puede ver de manera gráfica la matriz (Figura 3.16). Con diferentes parámetros se pueden cambiar los colores e incrustar código L<sup>A</sup>T<sub>E</sub>X para modificar los textos. Se ha codificado la función `matriz_confusion` que recibe el dataset y  $Y_m$ , y muestra el gráfico con la configuración personalizada.

### Precisión, sensibilidad, y $F_1$ -score

Si bien la matriz de confusión permite mostrar de manera visual la calidad de las clasificaciones, es conveniente calcular algunos valores [28] a partir de esta matriz que permitan visualizar de manera numérica dichas calidades. La primera de ellas conocida como **precisión** ( $\mathcal{P}$ ), y puede entenderse como la probabilidad de que un elemento que etiquetado como  $C_i$ , realmente pertenezca a dicha clase.

$$\mathcal{P} = P(C_i|\hat{C}_i) = \frac{P(C_i \cap \hat{C}_i)}{P(\hat{C}_i)}$$

La siguiente se denomina **sensibilidad** (o recuerdo,  $\mathcal{R}$ ) e indica la probabilidad de que un elemento de la clase  $C_i$ , se clasifique como tal.

$$\mathcal{R} = P(\hat{C}_i|C_i) = \frac{P(\hat{C}_i \cap C_i)}{P(C_i)}$$

En resumen y en términos de probabilidad condicionada,  $\mathcal{P}$  puede entenderse como una probabilidad a posteriori (cómo de bien ha clasificado el modelo), mientras que  $\mathcal{R}$  puede entenderse como una probabilidad a priori (cómo de bien clasificaría el modelo). Es frecuente calcular la media armónica de estos dos valores para obtener un único valor que evalúe la calidad de clasificación del modelo, comúnmente llamada  $F_1$ .

$$F_1 = \frac{2}{\mathcal{P}^{-1} + \mathcal{R}^{-1}}$$

Para calcular estos valores, se ha hecho uso de la función `classification_report` de Scikit-learn, que muestra en una tabla estos valores para cada clase, junto con las medias de dichos valores. En situaciones de clases desbalanceadas como es esta, es conveniente mirar las medias ponderadas (`weighted avg`) para que malos resultados en clases con pocos elementos contribuyan de manera proporcional al resultado final.

$$\bar{\mathcal{P}} = \sum_{i=1}^n P(C_i)P(C_i|\hat{C}_i) \quad \bar{\mathcal{R}} = \sum_{i=1}^n P(C_i)P(\hat{C}_i|C_i)$$

En las tablas obtenidas en la Figura 3.17, puede verse como los valores obtenidos corresponden con las situaciones que mostraban las matrices de confusión, siendo los mejores en aquellas que se aplica transfer learning al tener valores mayores, y que a pesar de tener clases desbalanceadas (se puede observar en la columna de `support`), los valores entre clases son similares (en dichos casos), obteniendo medias ponderadas similares a las calculadas sin ponderar. En los casos más negativos, se observan valores más bajos y comportamientos diferentes dependiendo de la clase.

### Curvas ROC y AUC

Otra manera de evaluar la calidad de un clasificador binario es mediante las conocidas como curvas ROC (*Receiver Operating Characteristic*), que representan en el espacio  $[0, 1] \times [0, 1]$  los falsos positivos frente a los verdaderos positivos[29]. Es más sencillo de entender y representar en términos de probabilidad condicionada, representando  $P(\hat{C}_i|\neg C_i)$  frente a  $P(\hat{C}_i|C_i)$ . Para determinar entre dos curvas cuál describe un mejor clasificador, lo que se hace es elegir aquella con un AUC (Area Under Curve) mayor. Si se toma la curva ROC como una función  $r : [0, 1] \longrightarrow [0, 1]$ , entonces el AUC se puede entender como

$$\mathcal{A} = \int_0^1 r(x) dx,$$

	precision	recall	f1-score	support		precision	recall	f1-score	support
camino_santiago	0.917	0.524	0.667	21	camino_santiago	0.955	1.000	0.977	21
cartel	0.812	0.903	0.855	62	cartel	0.968	0.968	0.968	62
marcador	0.625	0.278	0.385	18	marcador	0.944	0.944	0.944	18
parque	0.651	0.804	0.719	51	parque	0.980	0.961	0.970	51
accuracy			0.743	152	accuracy			0.967	152
macro avg	0.751	0.627	0.656	152	macro avg	0.962	0.968	0.965	152
weighted avg	0.750	0.743	0.728	152	weighted avg	0.967	0.967	0.967	152
(a) Normal					(b) Aplicando transfer learning				
	precision	recall	f1-score	support		precision	recall	f1-score	support
camino_santiago	0.481	0.619	0.542	21	camino_santiago	0.955	1.000	0.977	21
cartel	0.554	1.000	0.713	62	cartel	0.968	0.968	0.968	62
marcador	0.000	0.000	0.000	18	marcador	0.944	0.944	0.944	18
parque	0.538	0.137	0.219	51	parque	0.980	0.961	0.970	51
accuracy			0.539	152	accuracy			0.967	152
macro avg	0.393	0.439	0.368	152	macro avg	0.962	0.968	0.965	152
weighted avg	0.473	0.539	0.439	152	weighted avg	0.967	0.967	0.967	152
(c) Aplicando aumento de datos					(d) Aplicando ambas				

Figura 3.17: Precisión, sensibilidad, y  $F_1$ 

siendo  $\mathcal{A} = 1$  el mejor de los valores posibles. En general, cuanto mayores sean los valores del eje  $y$  para valores muy pequeños del eje  $x$ , mejor será el clasificador, mientras que cuando la curva ROC se aproxime a la función  $f(x) = x$ , más parecido será el modelo a hacer las clasificaciones al azar. En los casos que no son de clasificación binaria como este, no existe como tal una clase positiva y una negativa. Lo que se hace en su lugar es una de las siguientes aproximaciones[30].

- OVO (*one versus one*): para cada clase  $C_i$  y  $C_j$  con  $i \neq j$ , se toma una de ellas como clase positiva y la otra como negativa, y se calcula la curva ROC y AUC asociado, disponiendo de un total de  $\binom{n}{2}$  o  $n(n - 1)$  curvas según el autor o la librería utilizada (depende de considerar o no cada elemento de la pareja como positiva y negativa, y viceversa, teniendo que  $2\binom{n}{2} = n(n - 1)$ ).

$$\mathcal{A}_{\text{OVO}} = \frac{1}{n(n - 1)} \sum_{i=1}^n \sum_{j \neq i}^n P(C_i) \mathcal{A}(C_i, C_j)$$

- OVR (*one versus rest*): para cada clase  $C_i$  se toma como clase positiva, y el resto de las clases como una única clase negativa. Se calcula la correspondiente curva ROC y AUC asociado, obteniendo un total de  $n$  curvas.

$$\mathcal{A}_{\text{OVR}} = \sum_{i=1}^n P(C_i) \mathcal{A}(C_i, \neg C_i)$$

Es muy útil visualizar en una misma gráfica las diferentes curvas ROC de una de estas técnicas, sin embargo Scikit-learn no cuenta con ninguna función que haga esto directamente, sólo muestra el valor final. Por ello, se ha codificado una función `roc_auc_ovr` que recibe  $Y_m$ ,  $Y_s$ , y el nombre de las clases, y devuelve un gráfico en el que se ven las curvas ROC OVR de cada clase, los AUC de cada una, y el global. Para ello se ha hecho uso de la función `roc_curve`, que calcula la curva, `auc`, que calcula el AUC, y `RocCurveDisplay` que muestra un gráfico de la curva ROC. En la Figura 3.18 se muestran los cuatro resultados para cada uno de los casos, teniendo de nuevo que para las variantes de las Figuras 3.18b y 3.18d se observan los mejores resultados sin haber casi diferencia entre ambos.

Como resumen de esta parte, se puede concluir que el entrenamiento de una red convolucional desde cero es un proceso complejo y que requiere de una cantidad enorme de datos, que en los casos en los que no se

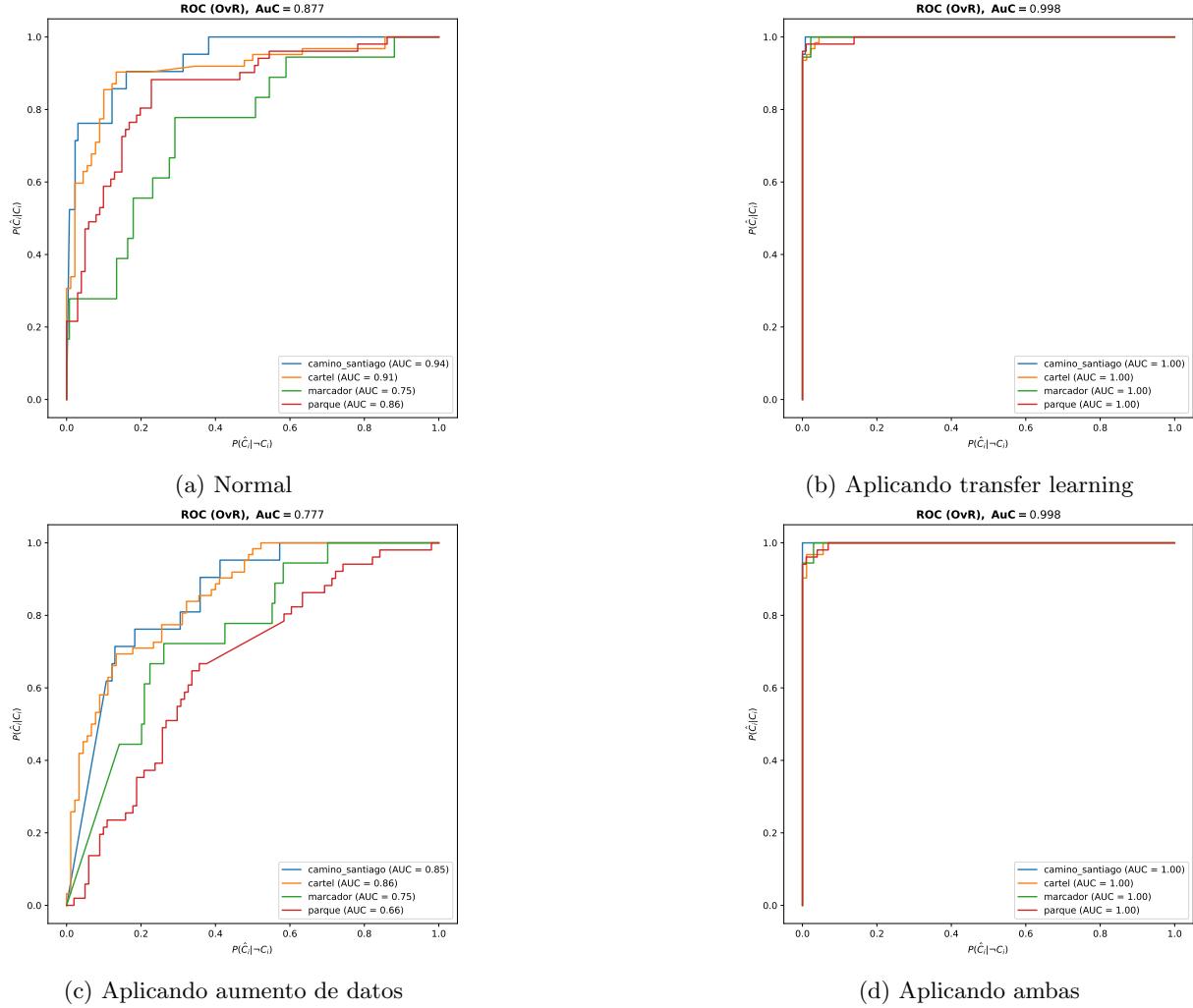


Figura 3.18: Curvas ROC

dispone de ellos, conduce a resultados negativos, siendo necesario aplicar la técnica de transfer learning. A pesar de haber visto su gran eficacia, se va a realizar una prueba más.

El dataset empleado, contenía imágenes de diferentes partes de España, por lo que el modelo estaba viendo variantes de cada tipo de objeto. Lo que se va a hacer ahora es separar dicho dataset de manera que contenga únicamente en el conjunto de test imágenes de la ciudad de Guadalajara. Al ser pocas, el experimento no será muy representativo, pero permitirá probar la eficacia de la red generalizando conceptos. Para ello se ha utilizado el mismo modelo que se ha utilizado en el caso de transfer learning (entrenando de nuevo los parámetros desde cero para evitar hacer predicciones de una imagen vista durante el entrenamiento).

Como se observa en la Figura 3.19, tras haber entrenado el modelo sin usar imágenes de objetos de Guadalajara, y enseñarle uno como por ejemplo este hito del Camino de Santiago, la red está casi segura que de uno de ellos se trata, a pesar de que sea completamente distinto a los que ha visto durante el entrenamiento con el aspecto típico de Galicia.



Figura 3.19: Capacidad de generalización de la red, predicción en Galicia y Guadalajara

### 3.2.6. Detección de clases desconocidas

Si bien hasta el momento se habían supuesto las condiciones ideales de que los modelos sólo recibían imágenes que contenían uno de los cuatro elementos seleccionados, en el problema que se trata de resolver aun habiendo entrenado un modelo para un número grande de clases, una persona podría enviar un punto de interés que no perteneciese a ninguna de las clases supuestas. En dicho caso, se necesitaría detectar que la imagen no debería pertenecer a ninguna de las clases que el modelo conoce, y descartar para ser evaluada de otra manera.

Este problema es ampliamente conocido como *open set recognition*, y existen diversas formas de abordarlo. En esta sección se va a demostrar la eficacia de dos de las más populares. En ambos casos se necesitan recopilar datos de imágenes que no pertenezcan a ninguna de las clases. Para esto, también se han tomado imágenes que superaron el proceso de valoración pero que difieren de las clases seleccionadas para este proyecto, como por ejemplo, grafitis, iglesias, fuentes, pistas deportivas, etc.

La primera de las aproximaciones consiste en tratar el resto de clases desconocidas como una única clase desconocida. Al dataset utilizado en los casos anteriores se ha añadido una clase **resto** que contiene diversas imágenes mezcladas. Utilizando el mejor de los modelos anteriores (aplicando transfer learning), y teniendo en cuenta que no todas las imágenes de la misma clase guardan relación entre sí, se han obtenido resultados realmente positivos, puesto que el valor del  $F_1$  es de 0,956, casi no hay clasificaciones incorrectas en la matriz de confusión (Figura 3.20b), y las curvas ROC son casi perfectas (Figura 3.20c).

La segunda consiste en utilizar un primer modelo que permita distinguir entre imágenes de las clases conocidas y las que no. Para ello se utiliza un clasificador binario con las clases **resto** y **no\_resto**, y un modelo que sea capaz de distinguir entre las clases originales. Para realizar esto, ambos modelos siguen la arquitectura del mejor modelo de los anteriores, adaptando el número de clases. Se puede visualizar un resumen de esta situación en la Figura 3.21.

En una primera aproximación podría parecer innecesario evaluar cada imagen en ambos modelos, pudiendo evaluar todas las imágenes en el clasificador binario, y aquellas que sean etiquetadas como **no\_resto**, ser evaluadas por el segundo clasificador. Sin embargo, esto es incorrecto, pues para medir la calidad de la arquitectura en general se necesita hacer uso de una matriz  $Y_s$  (en este caso de dimensiones  $n \times 5$ ) que contenga las probabilidades de pertenecer a cada clase, siendo dichas clases las  $C_i$  y  $R$ . Si hay una imagen que se etiqueta como  $R$ , se tienen  $P(R)$  y  $P(\neg R)$ , pero no  $P(C_i)$ . Para obtener dichas probabilidades, se necesita hacer uso del segundo clasificador, teniendo en cuenta que las probabilidades que devuelve este son  $P(C_i|\neg R)$ , tal y como se describe en la Figura 3.22.

	precision	recall	f1-score	support
camino_santiago	0.875	1.000	0.933	21
cartel	0.983	0.952	0.967	62
marcador	1.000	0.944	0.971	18
parque	0.961	0.961	0.961	51
resto	0.931	0.931	0.931	29
accuracy			0.956	181
macro avg	0.950	0.958	0.953	181
weighted avg	0.958	0.956	0.956	181

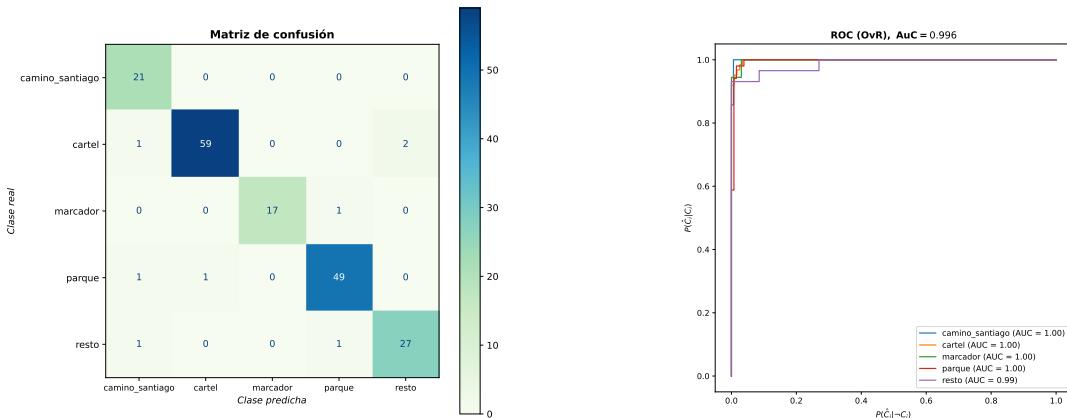
(a) Precisión, sensibilidad, y  $F_1$ 

Figura 3.20: Métricas en clasificación con clase desconocida

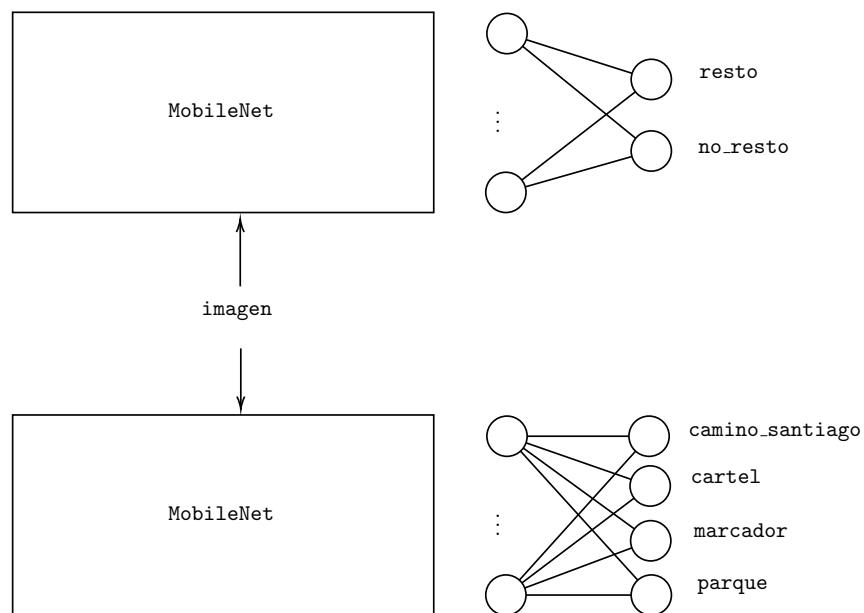


Figura 3.21: Arquitectura de clasificación con clases desconocidas

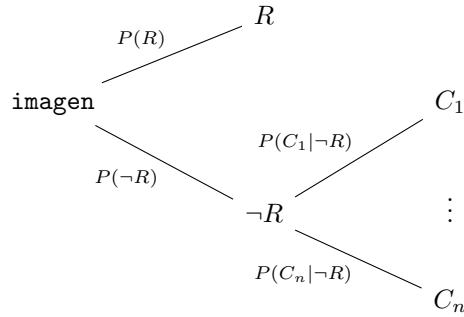


Figura 3.22: Diagrama de probabilidades

Una vez se dispone de  $P(R)$ ,  $P(\neg R)$ , y  $P(C_i|\neg R)$ , basta aplicar el Teorema de la probabilidad total para calcular los  $P(C_i)$ , obteniendo que  $P(C_i) = P(C_i|\neg R)P(\neg R)$ . Para calcular entonces la matriz  $Y_s$  necesaria, basta con tomar las matrices  $Y_s^{(1)}$  y  $Y_s^{(2)}$  (siendo cada una la obtenida por cada uno de los clasificadores), y corregir las probabilidades de la segunda matriz con la segunda columna de la primera matriz y concatenarla con la primera columna de la primera matriz, pudiéndose traducir esto a  $Y_s = \left( \text{diag} \left( Y_{s_{i1}}^{(1)} \right) Y_s^{(2)} \middle| Y_{s_{i2}}^{(1)} \right)$ , es decir,

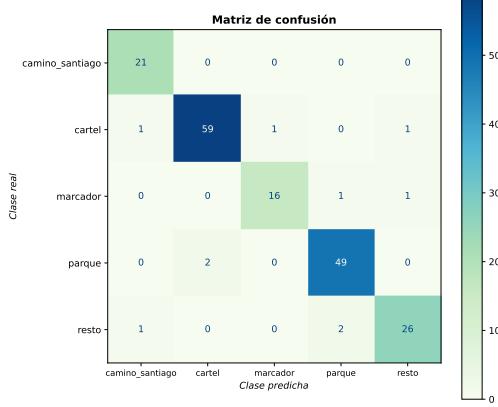
$$Y_s^{(3)} = \begin{pmatrix} P_1(\neg R) & 0 & \cdots & 0 \\ 0 & P_2(\neg R) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & P_n(\neg R) \end{pmatrix} \begin{pmatrix} P_1(C_1) & P_1(C_2) & \cdots & P_1(C_4) \\ P_2(C_1) & P_2(C_2) & \cdots & P_2(C_4) \\ \vdots & \vdots & \ddots & \vdots \\ P_n(C_1) & P_n(C_2) & \cdots & P_n(C_4) \end{pmatrix}$$

$$Y_s = \left( Y_s^{(3)} \middle| Y_{s_{i2}}^{(1)} \right).$$

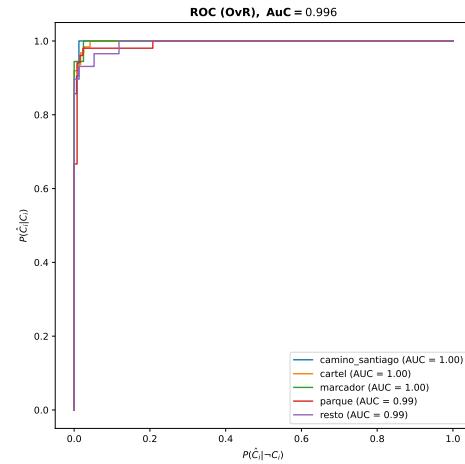
Por otro lado, ha de tenerse en cuenta que el primer clasificador da las etiquetas 0 y 1, y el segundo del 0 al 3, por lo que también se crea una matriz  $Y_m$  en función de las matrices  $Y_m^{(1)}$  y  $Y_m^{(2)}$  para tener etiquetas del 0 al 4. Con ambas matrices construidas se pueden evaluar los resultados en la Figura 3.23, pudiendo observar que también son muy positivos e incluso similares a los obtenidos por la arquitectura anterior. Finalmente añadir, que es importante que ambos modelos se entrenen con las mismas imágenes de manera que el rendimiento entre ambos sea similar, pues si por ejemplo el primer clasificador es malo distinguiendo entre  $R$  y  $\neg R$ , entonces dará igual cómo de bueno sea el segundo clasificador separando entre las otras cuatro clases, y viceversa. Como conclusión, podría elegirse cualquiera de las dos, aunque a efectos prácticos la primera es más interesante al ser más simple y necesitar un único entrenamiento.

	precision	recall	f1-score	support
camino_santiago	0.913	1.000	0.955	21
cartel	0.967	0.952	0.959	62
marcador	0.941	0.889	0.914	18
parque	0.942	0.961	0.951	51
resto	0.929	0.897	0.912	29
accuracy			0.945	181
macro avg	0.938	0.940	0.938	181
weighted avg	0.945	0.945	0.945	181

(a) Precisión, sensibilidad, y  $F_1$



(b) Matriz de confusión



(c) Curvas ROC y AUC

Figura 3.23: Métricas en clasificación con clase desconocida (doble modelo)

# Bibliografía

- [1] A. Abeliuk y C. Gutiérrez, *Historia y evolución de la inteligencia artificial*.
- [2] R. Szeliski, *Computer Vision: Algorithms and Applications*, 2nd ed. Springer International Publishing, 2022, ISBN: 978-3-030-34371-2. DOI: 10.1007/978-3-030-34372-9. dirección: <https://szeliski.org/Book/>.
- [3] López y H. I, “Método alternativo para calcular la convolución de señales en tiempo continuo,” 2009.
- [4] A. Lucía, “FFT: Transformada Rápida de Fourier,”
- [5] I. Goodfellow, Y. Bengio y A. Courville, *Deep Learning*. MIT Press, 2016, págs. 326-330. dirección: <http://www.deeplearningbook.org>.
- [6] W. Gao, L. Yang, X. Zhang y H. Liu, “An improved Sobel edge detection,” *Proceedings - 2010 3rd IEEE International Conference on Computer Science and Information Technology, ICCSIT 2010*, vol. 5, págs. 67-71, 2010. DOI: 10.1109/ICCSIT.2010.5563693.
- [7] *OpenCV: Image Filtering*. dirección: [https://docs.opencv.org/4.x/d4/d86/group\\_\\_imgproc\\_\\_filter.html](https://docs.opencv.org/4.x/d4/d86/group__imgproc__filter.html).
- [8] J. Wu, “Introduction to Convolutional Neural Networks,” 2017.
- [9] M. Minsky y S. A. Papert, *Perceptrons: An Introduction to Computational Geometry*. The MIT Press, ene. de 2017, ISBN: 9780262343930. DOI: 10.7551/MITPRESS/11301.001.0001.
- [10] H. Demuth y B. D. Jesús, *Neural Network Design 2nd Edition*. dirección: <https://hagan.okstate.edu/NNDesign.pdf>.
- [11] G. Cybenko, “Approximation by superpositions of a sigmoidal function,” *Mathematics of Control, Signals, and Systems*, vol. 2, págs. 303-314, 4 dic. de 1989, ISSN: 09324194. DOI: 10.1007/BF02551274/METRICS. dirección: <https://link.springer.com/article/10.1007/BF02551274>.
- [12] Q. Wang, Y. Ma, K. Zhao e Y. Tian, “A Comprehensive Survey of Loss Functions in Machine Learning,” *Annals of Data Science*, vol. 9, págs. 187-212, 2 abr. de 2022, ISSN: 21985812. DOI: 10.1007/S40745-020-00253-5/TABLES/8. dirección: <https://link.springer.com/article/10.1007/s40745-020-00253-5>.
- [13] S. Ruder, “An overview of gradient descent optimization algorithms,”
- [14] B. Ding, H. Qian y J. Zhou, “Activation functions and their characteristics in deep neural networks,” *Proceedings of the 30th Chinese Control and Decision Conference, CCDC 2018*, págs. 1836-1841, jul. de 2018. DOI: 10.1109/CCDC.2018.8407425.
- [15] T. Kurbiel, *Derivative of the Softmax Function and the Categorical Cross-Entropy Loss*. dirección: <https://towardsdatascience.com/derivative-of-the-softmax-function-and-the-categorical-cross-entropy-loss-ffceefc081d1>.
- [16] C. Kevin, *Feature Maps*. dirección: [https://medium.com/@chriskevin\\_80184/feature-maps-ee8e11a71f9e](https://medium.com/@chriskevin_80184/feature-maps-ee8e11a71f9e).
- [17] D. Katsaros, E. Fragkou y D. Papakostas, “Backpropagation for Convolutional Neural Networks,”
- [18] A. Vaswani, G. Brain, N. Shazeer et al., “Attention Is All You Need,” 2017.
- [19] J. L. Ba, J. R. Kiros y G. E. Hinton, “Layer Normalization,” jul. de 2016. dirección: <https://arxiv.org/abs/1607.06450v1>.

- [20] NVIDIA, ¿Qué es CUDA? Dirección: <https://support.nvidia.eu/hc/es/articles/4850516229266--Qu%C3%A9-es-CUDA>.
- [21] J. Ghorpade, J. Parande, M. Kulkarni y A. Bawaskar, “GPGPU PROCESSING IN CUDA ARCHITECTURE,” *Advanced Computing: An International Journal ( ACIJ )*, vol. 3, 1 2012. DOI: 10.5121/acij.2012.3109.
- [22] Q. Kemao y T. Wang, “GPU Acceleration for Optical Measurement,” *GPU Acceleration for Optical Measurement*, dic. de 2017. DOI: 10.1117/3.2314949.
- [23] M. Decuyper, M. Stockhoff, S. Vandenbergh, al - y X. Ying, “An Overview of Overfitting and its Solutions,” *Journal of Physics: Conference Series*, vol. 1168, pág. 022 022, 2 feb. de 2019, ISSN: 1742-6596. DOI: 10.1088/1742-6596/1168/2/022022. dirección: <https://iopscience.iop.org/article/10.1088/1742-6596/1168/2/022022%20https://iopscience.iop.org/article/10.1088/1742-6596/1168/2/022022/meta>.
- [24] M. Iman, H. R. Arabnia y K. Rasheed, “A Review of Deep Transfer Learning and Recent Advancements,” *Technologies 2023, Vol. 11, Page 40*, vol. 11, pág. 40, 2 mar. de 2023, ISSN: 2227-7080. DOI: 10.3390/TECHNOLOGIES11020040. dirección: <https://www.mdpi.com/2227-7080/11/2/40/htm%20https://www.mdpi.com/2227-7080/11/2/40>.
- [25] A. Mikolajczyk y M. Grochowski, “Data augmentation for improving deep learning in image classification problem,” *2018 International Interdisciplinary PhD Workshop, IIPHDW 2018*, págs. 117-122, jun. de 2018. DOI: 10.1109/IIPHDW.2018.8388338.
- [26] A. Bissoto, E. Valle y S. Avila, “GAN-Based Data Augmentation and Anonymization for Skin-Lesion Analysis: A Critical Review,” págs. 1847-1856, 2021.
- [27] M. Heydarian, T. E. Doyle y R. Samavi, “MLCM: Multi-Label Confusion Matrix,” *IEEE Access*, vol. 10, págs. 19 083-19 095, 2022, ISSN: 21693536. DOI: 10.1109/ACCESS.2022.3151048.
- [28] C. Goutte y E. Gaussier, “A Probabilistic Interpretation of Precision, Recall and F-Score, with Implication for Evaluation,” *Lecture Notes in Computer Science*, vol. 3408, págs. 345-359, 2005, ISSN: 03029743. DOI: 10.1007/978-3-540-31865-1\_25/COVER. dirección: [https://link.springer.com/chapter/10.1007/978-3-540-31865-1\\_25](https://link.springer.com/chapter/10.1007/978-3-540-31865-1_25).
- [29] T. Fawcett, “An introduction to ROC analysis,” *Pattern Recognition Letters*, vol. 27, págs. 861-874, 8 jun. de 2006, ISSN: 0167-8655. DOI: 10.1016/J.PATREC.2005.10.010.
- [30] C. Ferri, J. Hernández-Orallo y R. Modroiu, “An experimental comparison of performance measures for classification,” *Pattern Recognition Letters*, vol. 30, págs. 27-38, 2008. DOI: 10.1016/j.patrec.2008.08.010.





Universidad de Alcalá  
Escuela Politécnica Superior



ESCUELA POLITECNICA  
SUPERIOR

