Universidad del País Vasco
Euskal Herriko Unibertsitatea
ZIENTZIA ETA TEKNOLOGIA FAKULTATEA
FACULTAD DE CIENCIA Y TECNOLOGÍA

Trabajo de fin de grado
Doble Grado de Física e Ingeniería Electrónica

Estudio, desarrollo y evaluación de técnicas de aprendizaje automático para el reconocimiento de símbolos matemáticos escritos a mano

Autor:
Beñat Berasategui Miguéliz
Director:
Luis Javier Rodríguez Fuentes

Leioa, 20 de junio de 2024

ÍNDICE GENERAL

ACRÓNIMOS

Hay varios casos de términos en inglés cuyos acrónimos están muy extendidos en el campo del aprendizaje automático. Se resumen a continuación, a modo de referencia.

ML	Aprendizaje automático (*Machine Learning*)
SVM	Máquina de vectores soporte (*Support Vector Machine*)
ANN	Redes neuronales artificiales (*Artificial Neural Network*)
TLU	Unidad lógica de umbral (*Threshold Logic Unit*)
MLP	Perceptrón multicapa (*Multilayer Perceptron*)
DNN	Red neuronal profunda (*Deep Neural Network*)
ReLU	Unidad lineal rectificada (*Rectified Linear Unit*)
CNN	Red neuronal convolucional (*Convolutional Neural Network*)
MER	Exactitud combinada (*Merged Accuracy*)

1 INTRODUCCIÓN

En 1968 Donald Knuth publicó el primer volumen de su obra *El arte de programar ordenadores*. Se imprimió utilizando la monotipia, una técnica de composición de texto inventada en 1887, que dotaba a su trabajo de un estilo tipográfico clásico. Para los volúmenes y ediciones posteriores, se empleó una técnica más moderna llamada *fotocomposición*. Los resultados que se obtenían de esta manera no tenían un aspecto tan elegante, cosa que disgustó profundamente a Knuth.

Indignado con la calidad cada vez menor de la tipografía de sus libros, decidió diseñar su propio lenguaje de tipografía: fue un proyecto para el que necesitó casi una década. De esta manera nació TEX, que se publicó en 1978 por primera vez. Según se explica en el manual oficial *The TeXbook*, el nombre proviene de la raíz griega de palabras como "técnica" o "tecnología", $\tau\varepsilon\chi\nu\eta$, que significa también "arte" [1]. Es una referencia a que la técnica no tiene por qué estar reñida con el arte o la presentación elegante[1].

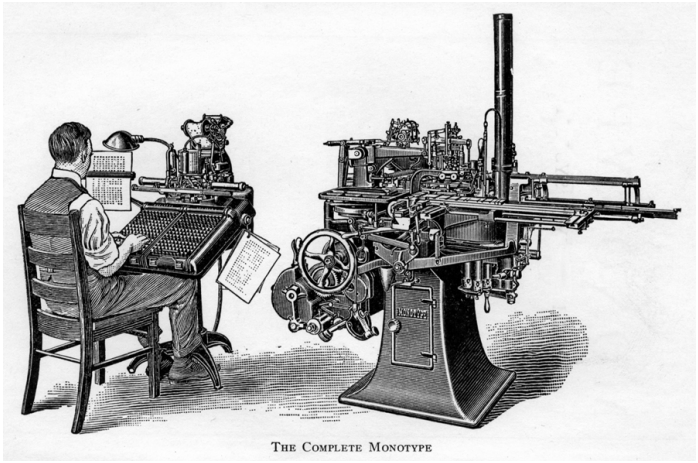


Figura 1.1: La máquina de composición Monotipo. El teclista producía una tira de papel perforada con un código de agujeros, utilizando un teclado parecido al de una máquina de escribir. Después, se introducía el papel en el fundidor, que lo interpretaba. Utilizando moldes adecuados, creaba piezas de metal para cada carácter. Tras ordenarlos adecuadamente, se mojaban con tinta y se imprimía el texto. Fuente: https://letterpresscommons.com/monotype/.

A su vez, Leslie Lamport creó LATEX en 1984, una colección de macros para facilitar la tarea de aquellos que desean utilizar TEX [2]. Este sistema permite que cualquiera con un ordenador produzca textos de una belleza y complejidad con la que ni los mejores tipógrafos del siglo XX podrían haber soñado. TEX y sus versiones posteriores han supuesto una revolución tecnológica.

Si bien es cierto que LATEX es muy popular en el entorno académico, presenta varias dificultades cuando se empieza a utilizar por primera vez. Uno de los problemas a los que se enfrenta el usuario principiante es el de encontrar el comando que corresponde al símbolo que quiere escribir, especialmente a la hora de crear ecuaciones.

1 Según Knuth, el nombre "TEX" está compuesto por las letras griegas τ, ε y χ en mayúsculas, lo que quiere decir que se debería pronunciar "tej" y no "tecs".

Existe la *Comprehensive LaTeX Symbol List*, un documento de más de 400 páginas en el que se recogen los comandos de los de símbolos que LaTeX reconoce [3]. A pesar de ser una referencia muy completa, no es demasiado útil para encontrar de forma rápida el símbolo que se busca.

Para sobrepasar esta dificultad, las herramientas como Detexify suponen una gran ayuda. Esta en particular es una página web en la que el usuario puede encontrar el comando del símbolo en cuestión dibujándolo directamente con el ratón del ordenador [4, 5].

El objetivo principal de este proyecto ha sido crear un sistema parecido de principio a fin, para entender el funcionamiento de este tipo de herramientas. Para ello, ha sido imprescindible encontrar una gran cantidad de imágenes de símbolos matemáticos escritos a mano, con las que entrenar diferentes modelos de aprendizaje automático. Sería impensable programar explícitamente un ordenador para que fuera capaz de distinguir entre cientos de símbolos, pero gracias a las técnicas de aprendizaje automático (o *machine learning*) un ordenador puede "aprender" a reconocer patrones de manera relativamente sencilla.

En el primer capítulo de este trabajo se explicará con mayor precisión la tarea que se ha elegido, tras lo cual se describirán los aspectos teóricos de los modelos de aprendizaje automático que se han utilizado. En el siguiente capítulo se explorarán las características de la base de datos que se ha empleado, se comentará brevemente cómo se han implementado los modelos y se definirán las métricas con las que se han evaluado sus rendimientos. Finalmente, se resumirán los resultados obtenidos. Se ha añadido además un apéndice que reúne los 369 símbolos diferentes con los que se ha trabajado.

El proyecto se ha desarrollado utilizando el lenguaje de programación Python, concretamente la versión `3.8.10`. Los paquetes Matplotlib, NumPy y Pandas han resultado de gran ayuda para trabajar con datos y visualizarlos. Se han usado tanto Scikit Learn como TensorFlow y Keras para entrenar los modelos de aprendizaje automático. Finalmente, Ipywidgets, Ipycanvas y Voila se han utilizado para construir una demo partiendo de un *notebook* de Jupyter.

Paquete	Versión
matplotlib	3.6.3
numpy	1.23.5
pandas	1.5.3
scikit-learn	1.2.0
tensorflow	2.12.0
keras	2.11.0
ipywidgets	8.0.4
ipycanvas	0.13.2
voila	0.5.7

2

TÉCNICAS DE APRENDIZAJE AUTOMÁTICO

2.1 DESCRIPCIÓN DE LA TAREA

La tarea que se ha escogido para este trabajo ha sido la de construir varios sistemas de clasificación que sean capaces de identificar símbolos matemáticos escritos a mano. Para ello, se ha utilizado la base de datos HASY, que contiene 168.233 muestras de 369 clases de símbolos distintos [6, 7].

HASY está inspirada en la archiconocida base datos MNIST, que contiene únicamente imágenes de dígitos escritos a mano del 0 al 9 y consta de 70.000 muestras. Existen muchos modelos con rendimiento excelente entrenados con estos datos, debido a que solo deben diferenciar entre 10 clases, cada clase tiene 6.000 muestras y no hay errores en las etiquetas que corresponden a las imágenes.

Figura 2.1: 125 muestras de la base de datos MNIST.

En cambio, las imágenes de HASY pertenecen a 369 categorías distintas, muchas de ellas tienen menos de 100 muestras, y además, en el desarrollo de este trabajo se han detectado varias imágenes etiquetadas de manera incorrecta. Por tanto, construir modelos adecuados con esta base de datos es una tarea más complicada.

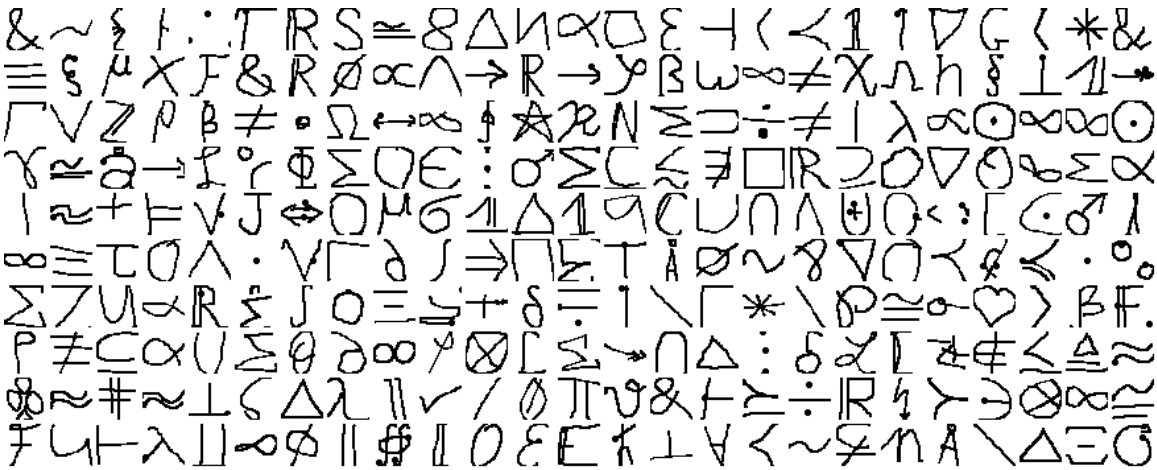


Figura 2.2: 250 muestras de la base de datos HASY.

En la Fig. 2.2 aparecen varios ejemplos de imágenes que componen la base de datos HASY. Como se puede observar, su resolución es muy baja: son imágenes en blanco y negro de 32 px × 32 px.

En este trabajo se construirán varios sistemas capaces de clasificar este tipo de imágenes y se compararán sus rendimientos, con el objetivo final de construir el prototipo de una aplicación que devuelva el comando de LaTeX que corresponde al símbolo que dibuje el usuario.

2.2 APRENDIZAJE AUTOMÁTICO

"El aprendizaje automático (ML, *Machine Learning*) es el campo de estudio que da a los ordenadores la capacidad de aprender sin ser programados de forma explícita". "Es la ciencia (y el arte) de programar los ordenadores de manera que puedan *aprender de los datos*" [8].

Un ejemplo típico es el filtro de correos electrónicos no deseados, que en esencia es un programa de ML capaz de diferenciar los correos que son *spam* de los que no, después de haber "aprendido" de unas cuantas muestras de correos de cada tipo. Los ejemplos de los que el sistema aprende forman el *conjunto de entrenamiento*; cada uno de ellos es una *muestra*. Se utilizan también otros ejemplos con los que no se entrena el sistema, que pertenecen al *conjunto de prueba*, para medir el rendimiento del mismo cuando es expuesto a muestras nuevas.

Utilizar técnicas de ML resulta ventajoso para solucionar problemas que son demasiado complejos para los enfoques tradicionales, o bien no tienen un algoritmo conocido.

Hay muchos tipos de aprendizaje automático, por lo que se pueden clasificar en varias categorías. Dependiendo de si se hace bajo la supervisión de humanos, se distinguen el aprendizaje *supervisado* y el *no supervisado*. Dependiendo de si puede pueden aprender sobre la marcha e incluir datos nuevos sin tener que reiniciar el entrenamiento, se habla de aprendizaje *online* y *aprendizaje por lotes*.

El aprendizaje no supervisado trabaja con datos sin etiquetar y es capaz de agrupar los que tienen características similares o detectar anomalías. Por el contrario, el aprendizaje supervisado necesita muestras *etiquetadas*. La clasificación es una tarea típica de aprendizaje supervisado.

El otro criterio que se ha mencionado depende de si el sistema puede aprender de manera gradual a partir de datos nuevos que se le proporcionan. En el aprendizaje por lotes el sistema es incapaz de hacer esto, los modelos se entrenan con todos los datos disponibles; si se obtienen nuevos datos que el sistema debe conocer, todo el entrenamiento debe volver a empezarse desde cero. Siendo un proceso que requiere mucho tiempo, suele hacerse *offline*. En el aprendizaje *online*, es posible entrenar el sistema de manera gradual, introduciendo instancias de datos de manera secuencial, lo que es idóneo para sistemas que reciben un flujo de datos nuevos constantemente.

En este trabajo se ha optado por utilizar modelos de aprendizaje *supervisado*, a los que se les proporcionarán las imágenes de los símbolos con el comando de LaTeX correspondiente. El entrenamiento se hará *offline*, utilizando todas las muestras disponibles.

2.3 MODELOS UTILIZADOS

2.3.1 *Máquinas de vectores soporte*

Las máquinas de vectores soporte (SVM, *Support Vector Machines*) son uno de los modelos más populares de aprendizaje automático. Son especialmente eficaces en bases de datos pequeñas o medianas.

2.3.1.1 *Clasificación SVM lineal*

Considérense dos clases que se pueden separar fácilmente con una línea recta, es decir, que son *separables linealmente*. Las SVM separan las dos clases utilizando el margen más ancho posible. En el ejemplo de la Fig. 2.3, la línea continua representa el límite de decisión del modelo y las líneas discontinuas la región más ancha posible entre las clases.

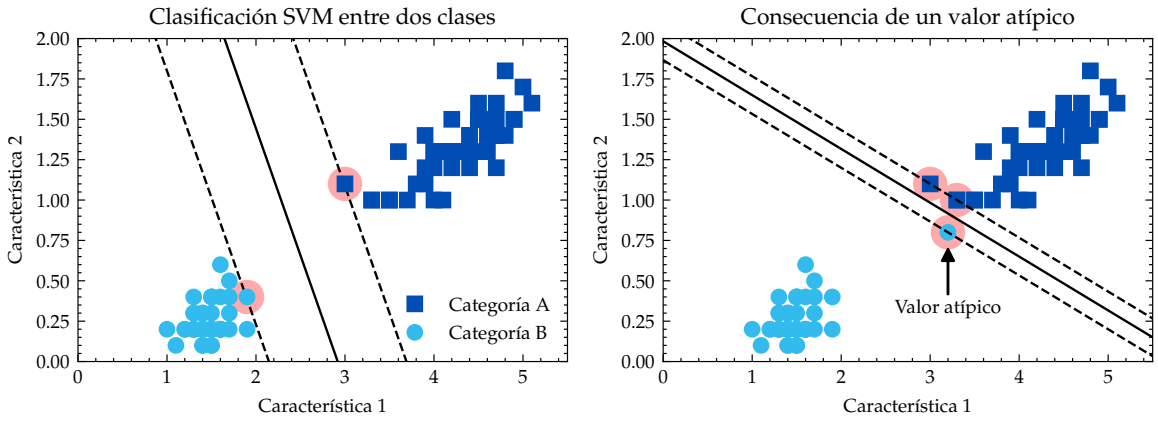


Figura 2.3: Ejemplo de clasificación SVM lineal entre dos clases con dos características. Los vectores soporte se indican en rojo. Estos modelos son sensibles a la escala de las características. Adaptado de [8].

Los puntos redondeados en rojo que determinan o *soportan* las líneas discontinuas son los *vectores soporte*. Son los únicos que afectan el límite de decisión, cualquier instancia que se añada fuera del margen representado no supondrá ningún cambio.

En casos de mayor dimensionalidad, es decir, con datos que tienen más características, las SVM tratan de encontrar no la línea, sino el *hiperplano* que separe las clases con el margen más ancho posible.

Si se impone que todas las instancias deben quedar fuera de la *calle* que limitan las líneas discontinuas, se dice que se está utilizando una *clasificación de margen duro*. No obstante, este método funciona únicamente con datos separables linealmente como los de la Fig. 2.3 y depende sensiblemente de cualquier instancia que por algún motivo se acerque al límite de decisión más que alguno de los vectores soporte *óptimos*.

Por lo tanto, suele ser conveniente utilizar modelos más flexibles que encuentran un equilibrio entre mantener el margen más ancho posible y tener cierta tolerancia a las violaciones del margen, esto es, a las instancias que están en el medio de la separación o incluso en el lado equivocado. Habitualmente es necesario aceptar algunas de estas excepciones para que el modelo generalice mejor. Esto se denomina *clasificación de margen blando*. El equilibrio se puede encontrar ajustando los *hiperparámetros* del modelo.

Cabe destacar que a diferencia de otros modelos, la salida de las SVM no es una probabilidad de que la muestra pertenezca a una de las clases. El sistema únicamente devuelve la predicción de la clase a la que pertenece la muestra.

2.3.1.2 *Clasificación SVM no lineal*

En caso de que las bases de datos no sean separables linealmente, es necesaria una nueva forma de clasificación. Una manera de afrontar estas situaciones es *fabricar* nuevas características que consigan que los datos pasen a ser separables linealmente. En el ejemplo

de la Fig. 2.4 se ve el efecto de añadir una segunda característica $x_2 = (x_1)^2$ a un conjunto de datos que en principio no es separable linealmente.

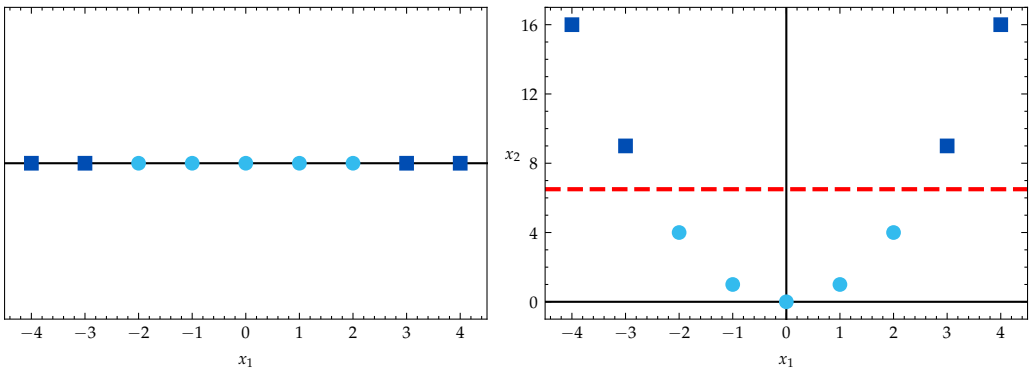


Figura 2.4: Ejemplo de adición de características para hacer que un conjunto de datos sea separable linealmente. Adaptado de [8].

La idea de incorporar características polinomiales es intuitiva y produce buenos resultados, pero en ocasiones será necesario añadir muchas de ellas, y esto puede hacer que el algoritmo se vuelva demasiado lento.

Afortunadamente, existe lo que se llama el *truco kernel*, un atajo matemático que permite obtener los resultados de añadir características polinomiales sin hacerlo realmente, evitando el problema de la ralentización del sistema.

2.3.1.3 *Funcionamiento de las SVM*

El modelo clasificador SVM lineal predice la clase de una nueva instancia $\mathbf{x}$ calculando la función de decisión

$$h(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + b = w_1x_1 + \cdots + w_nx_n + b, \tag{2.1}$$

donde $\mathbf{w}$ es el vector de pesos (*weights*) de las características y b es el término de sesgo (*bias*). Según el signo del valor obtenido se decide la clase $\hat{y}$ a la que la muestra pertenece.

$$\hat{y} = \begin{cases} A & \text{si } \mathbf{w}^T\mathbf{x} + b \geq 0, \\ B & \text{si } \mathbf{w}^T\mathbf{x} + b < 0. \end{cases} \tag{2.2}$$

El límite de decisión es el lugar geométrico de todos los puntos en los que $\mathbf{w}^T\mathbf{x} + b = 0$, mientras que las líneas discontinuas que delimitan el margen están compuestas por el conjunto de puntos en los que $h = \pm 1$ (ver la Fig. 2.5). Entrenar el modelo consiste en encontrar los valores de $\mathbf{w}$ y b que maximizan la anchura de este margen evitando o limitando los valores atípicos.

Siguiendo con el esquema de la Fig. 2.5, en los límites del margen se tiene que $\mathbf{w}^T\mathbf{x}_A + b = 1$ y $\mathbf{w}^T\mathbf{x}_B + b = -1$. Por lo tanto, restando ambas expresiones,

$$\mathbf{w}^T(\mathbf{x}_A - \mathbf{x}_B) = 2 \tag{2.3}$$

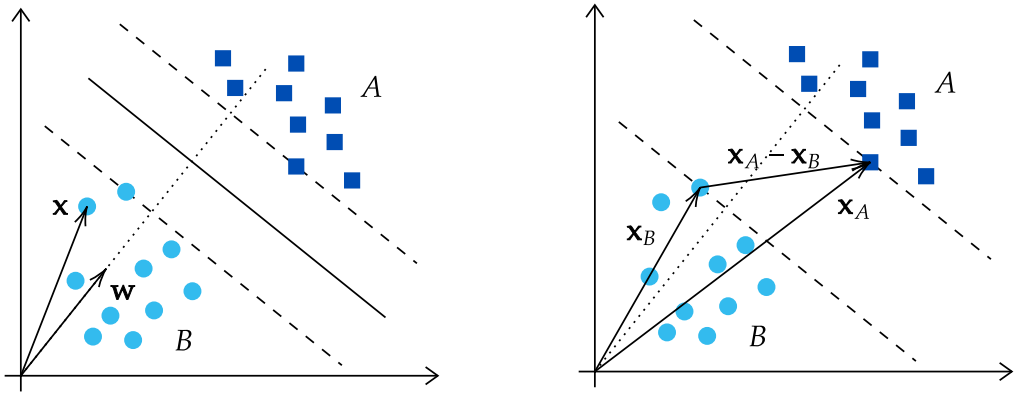


Figura 2.5: Esquema que representa los componentes de la función de decisión de una SVM. Adaptado de [9].

Por otro lado, viendo que $\mathbf{w}$ es perpendicular a las líneas discontinuas, se tiene que la distancia entre ambas es la proyección del vector $\mathbf{x}_A - \mathbf{x}_B$ en la dirección de $\mathbf{w}$:

$$d(A, B) = \frac{\mathbf{w}^T}{||\mathbf{w}||}(\mathbf{x}_A - \mathbf{x}_B) \tag{2.4}$$

Sustituyendo la Ec. (2.3), el ancho del camino $d(A, B)$ viene dado por:

$$d(A, B) = \frac{2}{||\mathbf{w}||}. \tag{2.5}$$

Esto quiere decir que para conseguir la máxima separación se debe minimizar $||\mathbf{w}||$. En la práctica se minimiza $\frac{1}{2}||\mathbf{w}||^2$, pues resulta matemáticamente más conveniente. Eso sí, se deben respetar las restricciones que impone que la función de decisión sea $h \geq 1$ para todas las instancias de la clase A y $h \leq 1$ para las de la clase B. Definiendo

$$t^{(i)} = \begin{cases} 1 & \text{si } y^{(i)} = A, \\ -1 & \text{si } y^{(i)} = B, \end{cases} \tag{2.6}$$

la restricción para todas las instancias se puede expresar como:

$$t^{(i)} \left(\mathbf{w}^T \mathbf{x}^{(i)} + b \right) \geq 1. \tag{2.7}$$

Esta es la condición de margen duro. Si se necesita un margen blando, se introduce la variable de holgura $\zeta^{(i)} \geq 0$ para cada instancia, que mide cuánto se le permite violar el margen a la i^a instancia. Se tienen dos objetivos: maximizar el ancho del camino y reducir las violaciones de margen. Para encontrar un equilibrio se utiliza el hiperparámetro C, que define la compensación entre los dos objetivos. De esta manera, se busca minimizar

$$\frac{1}{2}\mathbf{w}^T\mathbf{w} + C\sum_{i=1}^{m} \zeta^{(i)} \tag{2.8}$$

respecto a $\mathbf{w}$, b y ζ, respetando las restricciones

$$t^{(i)} \left(\mathbf{w}^T \mathbf{x}^{(i)} + b \right) \geq 1 - \zeta^{(i)} \quad \text{y} \quad \zeta^{(i)} \geq 0, \qquad i = 1, 2, \cdots, m. \tag{2.9}$$

Este es un "problema de optimización cuadrática convexa con restricciones lineales" [8]. En vez de estudiar este problema, puede tomarse el *problema dual* en el que se minimiza respecto a α la siguiente expresión:

$$\frac{1}{2}\sum_{i=1}^{m}\sum_{j=1}^{m}\alpha^{(i)}\alpha^{(j)}t^{(i)}t^{(j)}\mathbf{x}^{(i)T}\mathbf{x}^{(j)} - \sum_{i=1}^{m}\alpha^{(i)} \tag{2.10}$$

sujeto a las restricciones

$$\alpha^{(i)} \geq 0, \qquad i = 1, 2, \cdots, m. \tag{2.11}$$

Si la función del problema original que se quiere minimizar es convexa y las restricciones vienen dadas por funciones convexas y diferenciables de forma continua, se puede demostrar que el problema original y el dual tienen la misma solución [8].

Una vez que se encuentra el vector $\widehat{\boldsymbol{\alpha}}$ que minimiza la Ec. (2.10) del problema dual, se obtienen las soluciones del problema original utilizando las transformaciones:

$$\begin{aligned}\widehat{\mathbf{w}} &= \sum_{i=1}^{m}\hat{\alpha}^{(i)}t^{(i)}\mathbf{x}^{(i)},\\ \hat{b} &= \frac{1}{n_s}\sum_{\substack{i=1\\ \alpha^{(i)}>0}}^{m}\left(t^{(i)} - \widehat{\mathbf{w}}^T\mathbf{x}^{(i)}\right),\end{aligned} \tag{2.12}$$

donde n_s es el número de vectores soporte y $\alpha^{(i)} > 0$ implica sumar solo sobre los puntos que son vectores soporte.

El problema dual se resuelve más rápido que el original para los casos en los que el número de instancias m del conjunto de datos es menor que el número de características de cada una de ellas. Sin embargo, lo más importante del problema dual es que permite utilizar lo que se llama *truco kernel*, mientras que el problema original no.

Para entender el fundamento de esta estrategia, supóngase que se quiere obtener la representación de los datos en un espacio de mayor dimensión, aplicándoles la función $\phi : U \to V$,

$$\mathbf{z} = \phi(\mathbf{x}). \tag{2.13}$$

Se toma como ejemplo la aplicación polinómica de segundo grado $\phi : \mathbb{R}^2 \to \mathbb{R}^3$,

$$\phi(\mathbf{x}) = \phi\left(\begin{pmatrix}x_1\\ x_2\end{pmatrix}\right) = \begin{pmatrix}x_1^2\\ \sqrt{2}x_1x_2\\ x_2^2\end{pmatrix}. \tag{2.14}$$

Al resolver el problema dual, en ningún momento aparecen términos $\phi(\mathbf{x})$ independientemente. Se puede observar que en las ecuaciones anteriores (2.10, 2.12) solo aparecen en productos escalares $\phi(\mathbf{x}^{(i)})^T\phi(\mathbf{x}^{(j)})$. Incluso al calcular $\hat{b}$, sustituyendo la expresión de $\widehat{\mathbf{w}}$, se tiene que

$$\hat{b} = \frac{1}{n_s}\sum_{\substack{i=1\\ \alpha^{(i)}>0}}^{m}\sum_{j=1}^{m}\left(t^{(i)} - \hat{\alpha}^{(j)}t^{(j)}\mathbf{x}^{(j)T}\mathbf{x}^{(i)}\right). \tag{2.15}$$

Solo aparecen los productos escalares. Teniendo $\mathbf{a}$ y $\mathbf{b}$ no es necesario calcular $\phi(\mathbf{a})$ y $\phi(\mathbf{b})$ independientemente, basta con obtener el valor de $\phi(\mathbf{a})^T\phi(\mathbf{b})$, que a menudo es un

proceso más sencillo. En el caso del ejemplo de la aplicación polinómica de segundo grado, se puede calcular que

$$\phi(\mathbf{a})^T\phi(\mathbf{b}) = (\mathbf{a}^T\mathbf{b})^2. \tag{2.16}$$

La clave es que si se le aplica la transformación ϕ a todas las instancias, el problema dual contendrá únicamente los productos escalares $K(\mathbf{x}^{(i)}, \mathbf{x}^{(j)}) = \phi(\mathbf{x}^{(i)})^T\phi(\mathbf{x}^{(j)})$. Si se conoce K, no es necesario tener la expresión de ϕ. En ML a la función K se le llama *kernel*, y sirve para calcular el producto escalar $\phi(\mathbf{x}^{(i)})^T\phi(\mathbf{x}^{(j)})$ sin tener que calcular explícitamente (ni siquiera conocer) la transformación ϕ, a partir de los vectores $\mathbf{x}^{(i)}$ y $\mathbf{x}^{(j)}$ originales.

Los *kernels* utilizados habitualmente son:

$$\begin{aligned}
\text{Lineal: } K(\mathbf{a}, \mathbf{b}) &= \mathbf{a}^T\mathbf{b}. \\
\text{Polinomial: } K(\mathbf{a}, \mathbf{b}) &= \left(\gamma\mathbf{a}^T\mathbf{b} + r\right)^d. \\
\text{Función de base radial gaussiana: } K(\mathbf{a}, \mathbf{b}) &= \exp\left(-\gamma\|\mathbf{a} - \mathbf{b}\|^2\right). \\
\text{Sigmoide: } K(\mathbf{a}, \mathbf{b}) &= \tanh\left(\gamma\mathbf{a}^T\mathbf{b} + r\right)
\end{aligned} \tag{2.17}$$

2.3.2 *Árboles de decisión y random forests*

Al igual que las SVM, los *árboles de decisión* son métodos de aprendizaje automático versátiles capaces de adaptarse a bases de datos complejas. Son los componentes fundamentales de los bosques aleatorios (*random forests*) que se verán más adelante.

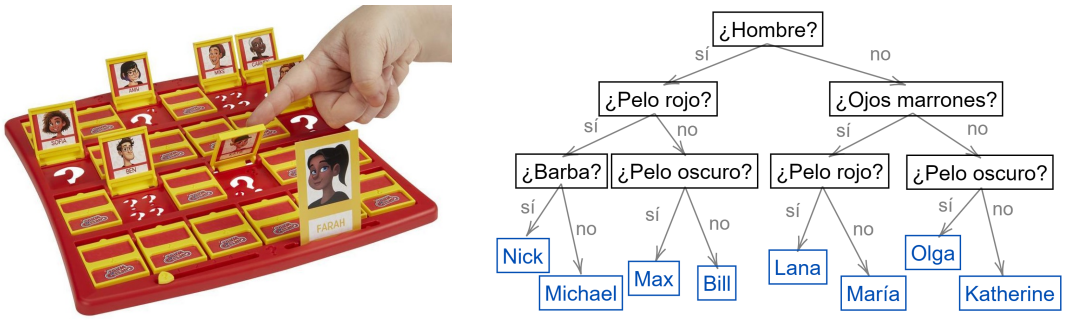


Figura 2.6: Ejemplo de árbol de decisión para el juego *¿quién es quién?* (*Guess Who? Classic Game* de Hasbro: `https://shop.hasbro.com/`).

Un buen ejemplo para comprender el funcionamiento de los árboles de decisión es el juego *¿quién es quién?*, en el que uno de los jugadores elige uno de los posibles personajes, y el otro jugador debe hacer preguntas de respuesta *sí* o *no* para adivinar de quién se trata. Cada pregunta corresponde a un *nodo*, y dependiendo de la respuesta se construyen nodos *hijos*. El proceso de tomar la decisión, partiendo del *nodo inicial* hasta llegar a los *nodos finales* da lugar a una estructura en forma de árbol, de ahí el nombre del método. A los nodos finales se les llama también *hojas*.

Una de las ventajas de estos modelos es que, a diferencia de las SVM, no importa cómo estén escaladas las características y casi no necesitan preprocesamiento de los datos.

Para construir un árbol de decisión adecuado, la clave es elegir qué preguntas hacer y en qué orden. Lo ideal sería elegir las preguntas que dan más información sobre lo que se quiere predecir. Este contenido de información está relacionado con la *pureza*: se dice que un conjunto es *puro* si todas las instancias que lo componen pertenecen a la misma clase.

El algoritmo con el que Scikit-Learn entrena los árboles de decisión se llama *Classification and Regression Trees* (CART). Produce árboles binarios, es decir, cada nodo tiene dos *hijos*. Primero, divide el conjunto de entrenamiento en dos, según una única característica k y un umbral t_k. Aunque en el ejemplo de *quién es quién* las respuestas son binarias, también pueden tomarse condiciones como "estatura $\leq 170\,cm$". Se busca la pareja (k, t_k) que produzca los subconjuntos más puros. La función de coste que el algoritmo trata de minimizar es:

$$J\left(k, t_k\right) = \frac{m_{\text{izq}}}{m} G_{\text{izq}} + \frac{m_{\text{dch}}}{m} G_{\text{dch}} \tag{2.18}$$

donde G_{izq} y G_{dch} miden la pureza de los subconjuntos de la izquierda/derecha, y m_{izq} y m_{dch} son el número de muestras de cada uno de ellos y $m = m_{\text{izq}} + m_{\text{dch}}$. Una vez que se termina este paso, se vuelven a dividir los subconjuntos siguiendo la misma lógica, después los sub-subconjuntos... de manera sucesiva. El proceso se para cuando se llega a la profundidad máxima establecida, o cuando no se pueda encontrar una partición que reduzca más la impureza. Este tipo de algoritmos produce una solución razonablemente buena, pero no garantiza que sea la solución óptima.

Dos maneras de medir la impureza G son la *impureza Gini* y la *entropía*. La expresión de la primera es

$$G_i = 1 - \sum_{k=1}^{n} p_{i,k}^2 \tag{2.19}$$

en la que $p_{i,k}$ es la razón entre el número de instancias de la clase k y el número total de instancias de entrenamiento del nodo i-ésimo.

En la teoría de la información, la *entropía* mide el contenido de información medio de un mensaje. Cuando un conjunto contiene instancias de una única clase, la entropía es nula. La entropía del nodo i-ésimo se calcula de esta manera:

$$H_i = - \sum_{\substack{k=1 \\ p_{i,k} \neq 0}}^{n} p_{i,k} \log_2\left(p_{i,k}\right). \tag{2.20}$$

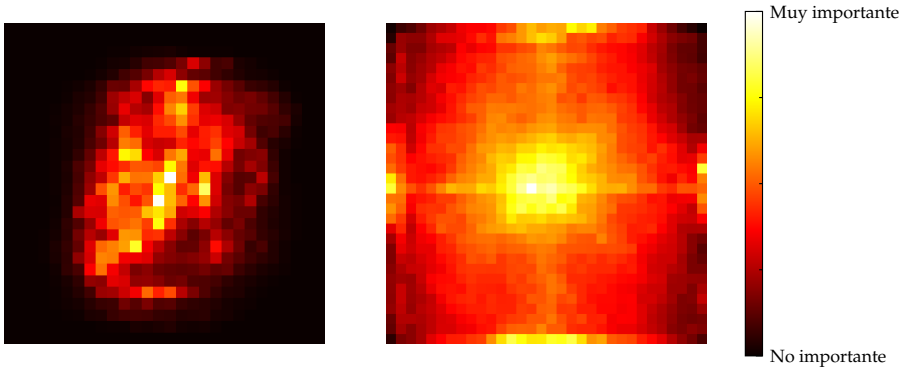
Generalmente se obtienen resultados parecidos utilizando cualquiera de las dos opciones. La impureza Gini se calcula de manera ligeramente más rápida, mientras que la entropía suele producir árboles algo más equilibrados.

Aunque el entrenamiento de los árboles de decisión pueda requerir tiempo, las predicciones se suelen hacer muy rápidamente, puesto que generalmente atravesar un árbol hasta llegar al nodo final requiere recorrer del orden de $\mathcal{O}\left(\log_2(m)\right)$ nodos.

Se dice que los árboles de decisión son *cajas blancas*, a diferencia de otros modelos que son *cajas negras* (como las redes neuronales), puesto que son intuitivos y sus decisiones son fáciles de interpretar. Se pueden revisar las reglas que utilizan, e incluso aplicar manualmente. De esta manera, se podría obtener una lista de las preguntas que pueden llevar a la victoria más rápidamente en el juego de *¿quién es quién?* (Fig. 2.6). En la Fig. 2.7 se muestra la importancia que le da un conjunto de árboles de decisión a cada píxel de las muestras de las bases de datos MNIST y HASY.

2.3.2.1 *Limitaciones de los árboles de decisión*

A pesar de tener muchas ventajas, también tienen alguna limitación. Se ha mencionado que estos modelos son capaces de adaptarse a conjuntos de datos complejos, y esto puede llegar a ser un problema. Si no se le impone ningún límite, la estructura del árbol se ajustará



(a) MNIST **(b)** HASY

Figura 2.7: Importancia de cada píxel de las muestras de las bases de datos MNIST y HASY, según un clasificador *random forest*, que es un conjunto de árboles de decisión.

completamente a los datos de entrenamiento, y lo más probable es que se *sobreajuste*. Es decir, que no sea capaz de generalizar a nuevos datos de manera eficaz.

Para evitar este problema hay que restringir la libertad del árbol de decisión. A este proceso se le llama *regularización*. Se utilizan diversos hiperparámetros con este fin, siendo el más común la propia *profundidad* del árbol de decisión, el número de niveles del nodo inicial a las hojas. Otras maneras de restringir el entrenamiento es fijar el número mínimo de muestras que un nodo debe tener para poder dividirse, el número máximo de hojas o la cantidad máxima de características que se evalúan en cada nodo.

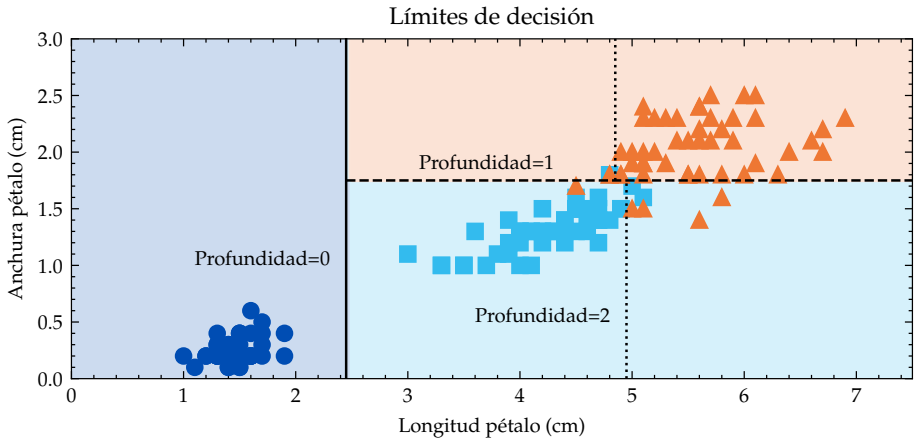


Figura 2.8: Límites de decisión de un árbol de decisión. En este ejemplo se intenta distinguir entre tres tipos de flores utilizando la anchura y la longitud de sus pétalos. La línea continua es la frontera que impone el nodo inicial. La parte izquierda es pura, por lo que no se divide más. El siguiente nodo marca una división para la anchura $= 1.75$ cm (línea discontinua). Los siguientes dos nodos generan las fronteras representada con la línea punteada. Adaptado de [8].

Otra de las limitaciones es que los límites de decisión son ortogonales, como se puede notar en las Figs. 2.8 y 2.9. Todas las divisiones se hacen de manera perpendicular a los ejes. Esto hace que sean sensibles a las rotaciones del conjunto de datos. También son sensibles a pequeñas variaciones en el conjunto de entrenamiento, y es más, puesto que el algoritmo que Scikit-Learn utiliza en su entrenamiento es estocástico, se pueden obtener modelos muy distintos partiendo de los mismos datos.

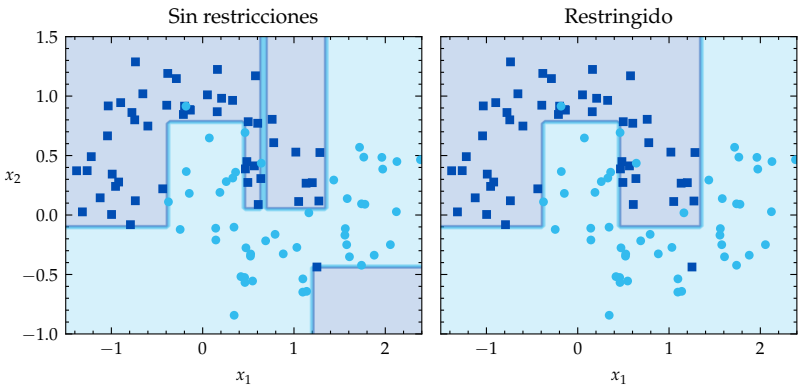


Figura 2.9: Dos árboles de decisión. El primero se ha entrenado sin restricciones, con los valores por defecto de los hiperparámetros. En el segundo se evita el sobreajuste utilizando restricciones. Adaptado de [8].

Como se verá a continuación, los *random forests* pueden limitar esta inestabilidad calculando la media entre las predicciones de varios árboles de decisión.

2.3.2.2 *Random forests*

Si se reúnen las predicciones de un grupo de predictores, a menudo se obtienen mejores resultados que con el mejor de los modelos individuales. Esta estrategia se denomina *ensamblaje de modelos*.

Como ejemplo de esta técnica se puede entrenar un grupo de árboles de decisión, cada uno en un subconjunto aleatorio del conjunto de entrenamiento. Tras obtener las predicciones individuales, la predicción del sistema será la que más *votos* tenga. Al ensamble de árboles de decisión se le llama bosque aleatorio (*random forest*), y es uno de los algoritmos de ML más poderosos disponibles hoy en día.

Generalmente esta combinación de modelos se hace una vez que se han construido modelos individuales adecuados, hacia el final del proyecto, para obtener un modelo combinado todavía mejor. Sin embargo, en este trabajo se ha puesto la atención en el rendimiento de los modelos individuales, por lo que el único ejemplo de ensamble de modelos será el de los *random forests*.

Conviene que los modelos individuales sean tan independientes entre ellos como sea posible, puesto que de esta manera será más probable que cada uno falle de manera distinta, y que al combinarlos estos errores se compensen. Una manera de conseguir esto es utilizar algoritmos que sean muy distintos. Otra, utilizar el mismo algoritmo, pero entrenar cada predictor en un subconjunto distinto de los datos de entrenamiento, elegido aleatoriamente.

Cuando se realiza un muestreo *con* reemplazo, este método se denomina *bagging*, que es la abreviatura de *bootstrap aggregating*, es decir, agregación *bootstrap*[1]. Si se muestrea *sin* reemplazo, se denomina *pasting*. Ambos métodos permiten que las mismas instancias de entrenamiento sean vistas por múltiples predictores, pero solo el *bagging* permite que el mismo predictor vea varias veces la misma instancia.

Una vez que se obtienen las predicciones individuales, el ensamble suele devolver la *moda* estadística para la clasificación o la media si es una tarea de regresión.

Cada árbol individual tendrá mayor sesgo que si se hubiera entrenado en el conjunto de entrenamiento original, pero el ensamblaje reduce tanto el sesgo como la varianza, dando

1 En estadística, el muestreo con reemplazo se llama *bootstrapping*.

como resultado un predictor de sesgo similar, pero con una varianza más baja que la de un único árbol entrenado en todo el conjunto.

Cada árbol del ensamble se puede entrenar en paralelo, y las predicciones también se pueden hacer de esta manera, por lo que los *random forests* toleran bien las bases de datos grandes.

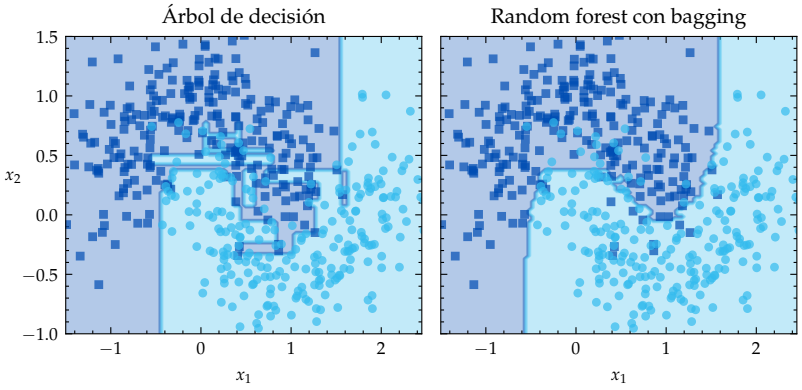


Figura 2.10: Árbol de decisión individual a la izquierda, ensamblaje con *bagging* de 500 árboles a la derecha. Se puede ver que las predicciones generalizarán mejor en el segundo caso. El sesgo de los dos modelos es parecido, pero la varianza del segundo es menor, por lo que el límite de decisión es menos irregular. Adaptado de [8].

2.3.3 *Perceptrones multicapa*

Las redes neuronales artificiales (ANN, *Artificial Neural Networks*) son modelos de ML inspirados en las estructuras neuronales de los cerebros. Al menos originalmente, ya que paulatinamente se han ido diferenciando de sus análogos biológicos, y hoy en día la analogía ha dejado de funcionar.

Las redes neuronales son el núcleo del llamado *aprendizaje profundo*. Son modelos versátiles que sirven para tareas tan complejas como la clasificación de imágenes o el reconocimiento de voz.

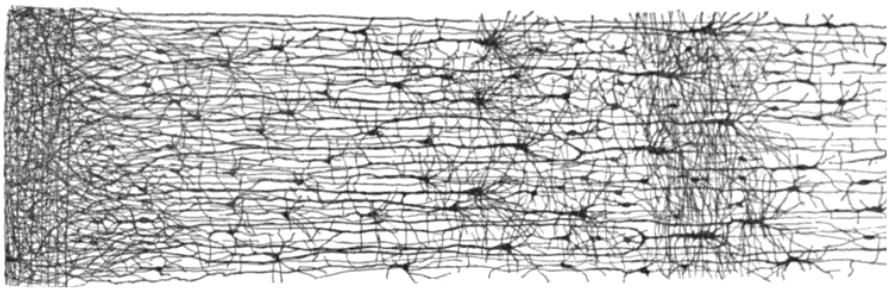


Figura 2.11: Red neuronal biológica del córtex cerebral humano. Dibujo de S. Ramón y Cajal, en *Comparative study of the sensory areas of the human cortex* (1899).

Las ANN no son modelos nuevos, aparecieron por primera vez en 1943 en el artículo "A Logical Calculus of Ideas Immanent in Nervous Activity" [10]. Los autores presentaron un modelo simplificado de cómo las neuronas biológicas podrían hacer cálculos complejos utilizando la *lógica proposicional*. A pesar de las expectativas iniciales, los recursos de aquella época no eran suficientes para crear redes neuronales artificiales robustas, por lo que se perdió el interés por ellas.

Hoy en día la situación es muy distinta: existen cantidades de datos inmensas con las que entrenar las ANN, el poder computacional ha crecido enormemente y los algoritmos de entrenamiento han mejorado, de manera que es posible entrenar grandes redes neuronales en un tiempo razonable. Por todo ello, las ANN actualmente gozan de bastante popularidad y están experimentando muchos progresos.

El *perceptrón* es una de las arquitecturas de ANN más simples, creada por F. Rosenblatt en 1957 [11]. La neurona utilizada se llama unidad lógica de umbral (TLU, *Threshold Logic Unit*). Sus entradas y salidas son números, y cada *conexión* de entrada tiene asociado un peso. La TLU primero calcula una suma ponderada de sus entradas ($z = \sum_i w_i x_i$), después le aplica la función escalón a la suma y finalmente produce ese resultado como salida:

$$h_{\mathbf{w}}(\mathbf{x}) = u(\mathbf{x}^T\mathbf{w}) = u(z), \qquad z = \mathbf{x}^T\mathbf{w}, \tag{2.21}$$

donde $u(z)$ es la función escalón de Heaviside. En este contexto, a $u(z)$ se le llama *función de activación*. Entrenar una TLU significa encontrar los valores apropiados de w_i.

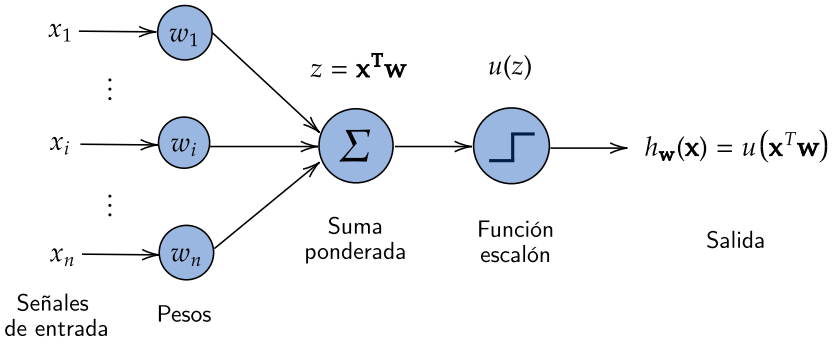


Figura 2.12: Una unidad lógica de umbral (TLU) es una neurona artificial que calcula una suma ponderada de sus entradas y después le aplica la función escalón al resultado.

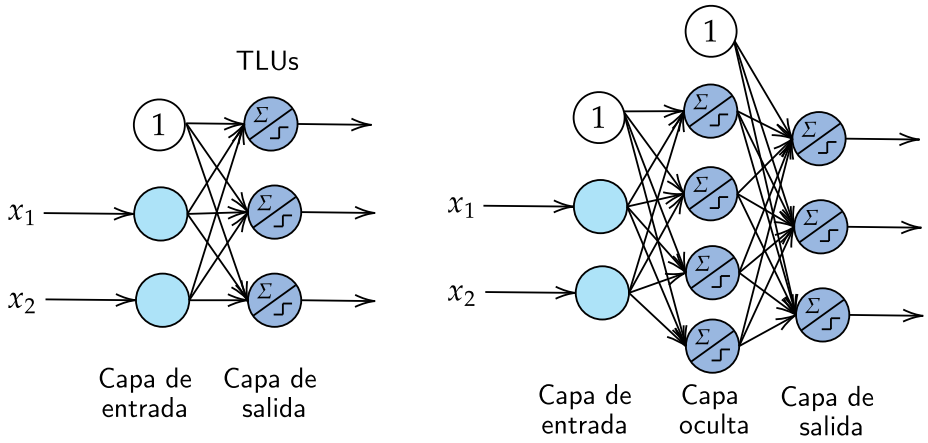
Un *perceptrón* está compuesto de una única *capa* de TLUs, en la que cada TLU está conectada a todas las entradas. Cuando todas las neuronas de una capa están conectadas a todas las de la capa anterior, se dice que dicha capa es *densa* o *completamente conectada*. Se utiliza también una capa de entrada, que introduce las señales de entrada a la red, y se incluye además una *neurona de sesgo* que añade una característica adicional, $x_0 = 1$. Las salidas de una capa de neuronas artificiales se calcula de esta forma:

$$h_{\mathbf{W},\mathbf{b}}(\mathbf{X}) = \phi(\mathbf{X}\mathbf{W} + \mathbf{b}), \tag{2.22}$$

donde $\mathbf{X}$ representa la matriz de las entradas, con una fila por instancia y una columna por característica y $\mathbf{W}$ es la matriz que contiene todos los pesos de las conexiones (excepto los de las neuronas de sesgo, que están contenidos en el vector $\mathbf{b}$). $\mathbf{W}$ tiene una fila por cada neurona de ingreso, y una columna por cada neurona de la capa. $\mathbf{b}$ tiene un elemento por cada neurona artificial. Como ya se ha mencionado antes, la función ϕ se denomina *función de activación*, que en el caso de las TLUs es la función escalón.

El entrenamiento de un perceptrón considera el error que comete la red al hacer una predicción y refuerza las conexiones entre neuronas que ayudan a reducirlo. El ajuste de pesos se hace de la siguiente manera:

$$w_{i,j}^{(\text{siguiente paso})} = w_{i,j} + \eta(y_j - \hat{y}_j)x_i. \tag{2.23}$$



(a) Perceptrón. **(b)** Perceptrón multicapa.

Figura 2.13: Comparación entre un perceptrón y un perceptrón multicapa. Ambas estructuras tienen dos entradas y tres salidas. Los círculos blancos representan las neuronas de sesgo, que siempre producen 1 como salida. Adaptado de [8].

En esta expresión, $w_{i,j}$ es el peso de la conexión entre la i^{a} neurona de entrada y la j^{a} neurona de salida, x_i es el i^{o} valor de entrada de la instancia de entrenamiento actual, $\hat{y}_j$ es la salida de la j^{a} neurona de salida para esa instancia, y y_j la salida objetivo de la j^{a} neurona. Por último, η es la *tasa de aprendizaje*, que condiciona el efecto de cada paso.

Puesto que los límites de decisión de cada neurona de la capa de salida son lineales, los perceptrones no pueden aprender patrones complejos. Eso sí, Rosenblatt demostró el *teorema de convergencia del perceptron*, que dice que si el problema es separable linealmente, el algoritmo convergerá a una solución.

Resulta que esta debilidad de los perceptrones puede solucionarse apilando varios perceptrones. La ANN que se crea de esta manera se denomina perceptrón multicapa (MLP, *Multilayer Perceptron*).

Como se puede observar en la Fig. 2.13b, un MLP se compone de una capa de entrada, una o más *capas ocultas* compuestas de TLUs, y una capa de TLUs final, llamada *capa de salida*. Todas excepto la última capa incluyen una neurona de sesgo y están completamente conectadas. Cuando una ANN contiene varias capas ocultas, se denomina red neuronal profunda (DNN, *Deep Neural Network*).

Encontrar una manera de entrenar los perceptrones multicapa fue un reto hasta que en 1986 llego el algoritmo de *retropropagación*, que hoy en día sigue utilizándose [12]. En solo dos pasadas a través de la red, una hacia delante y otra hacia atrás, este algoritmo puede calcular el gradiente de error de la red respecto a cada parámetro del modelo. Es decir, averigua cómo se debe ajustar cada peso y cada término de sesgo para reducir el error.

Este algoritmo hace primero una predicción para cada instancia de entrenamiento y mide el error. Después, vuelve por cada capa en sentido contrario, midiendo la contribución al error de cada conexión, y finalmente ajusta los pesos de las conexiones para reducir el error [8].

Para que el algoritmo funcione correctamente, es necesario cambiar la función de activación de las neuronas: en vez de la función escalón, es conveniente utilizar la función *sigmoide* o *logística*:

$$\sigma(z) = \frac{1}{1 + e^{-z}}. \tag{2.24}$$

Esto se debe a que la función escalón solo contiene segmentos planos, de derivada nula, y el algoritmo de retropropagación necesita derivadas distintas de cero para progresar correctamente. Otras funciones de activación que funcionan de manera satisfactoria son la tangente hiperbólica y la función de unidad lineal rectificada (ReLU, *Rectified Linear Unit*):

$$\begin{aligned} \tanh(z) &= \frac{e^z - e^{-z}}{e^z + e^{-z}} = 2\sigma(2z) - 1, \\ \mathrm{ReLU}(z) &= \max(0, z). \end{aligned} \tag{2.25}$$

La función ReLU es continua pero no es diferenciable en $z = 0$, y su derivada es nula para $z < 0$, lo que puede suponer un problema para el algoritmo de minimización del error. Sin embargo, en la práctica funciona muy bien y tiene la ventaja de poder calcularse muy rápido, por lo que es la función de activación que se utiliza por defecto.

Cabe destacar que el uso de funciones de activación es lo que le da a los MLP la capacidad de adaptarse a datos que no son separables linealmente. Si no se añade ninguna no-linealidad entre capas, por muchas transformaciones lineales que se combinen, el resultado será también una función lineal de las entradas. Estas, cómo se ha visto, no permiten resolver problemas demasiado complejos.

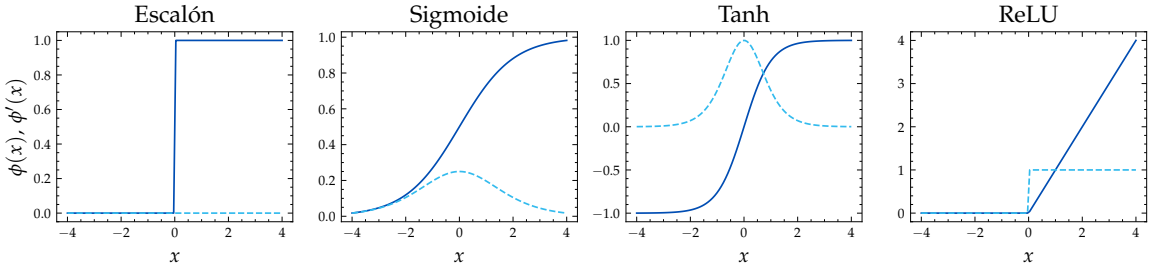


Figura 2.14: Funciones de activación mencionadas en el texto. Las derivadas de cada una se representan con línea discontinua.

Utilizando la función sigmoide en la capa de salida solo se pueden resolver problemas de clasificación binaria. La salida será un número entre 0 y 1, que puede interpretarse como la probabilidad estimada de una de las dos clases. Para adaptar estos modelos a situaciones con más clases, se utiliza la función *softmax*:

$$\hat{p}_k = \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}}, \qquad z_k = \mathbf{x}^T \mathbf{w}_k, \tag{2.26}$$

donde K es el número de clases y $\hat{p}_k$ la probabilidad de que una instancia dada pertenezca a la clase k.

Si cada instancia pertenece únicamente a una clase entre K, es necesario que la capa de salida tenga una neurona por clase, y utilizar la función de activación softmax para toda la capa. Esta función garantiza que todas las probabilidades estimadas tomarán valores entre 0 y 1, y que la suma de todas sea unitaria. Esto se conoce como *clasificación multiclase*.

2.3.4 *Redes neuronales convolucionales*

Las idea de las redes neuronales convolucionales (CNN, *Convolutional Neural Networks*) surgió del estudio de córtex visual del cerebro. Estos modelos se han utilizado en tareas de reconocimiento de imágenes desde la década de 1980. En los últimos años han superado el

rendimiento humano en varias tareas complejas, gracias al aumento de datos disponibles y del poder de cómputo. Sirven también para el reconocimiento de voz y el procesamiento del lenguaje natural.

D. Hubel y T. Wiesel recibieron el premio Nobel de Medicina de 1981, por su trabajo sobre el procesamiento de información en el sistema visual. Tras experimentar con gatos y monos, dieron a conocer que muchas neuronas del córtex visual reaccionan únicamente a estímulos de una región limitada del campo visual. Es más, mostraron que unas neuronas son sensibles únicamente a las líneas horizontales, y otras a líneas con otras orientaciones, y es de la combinación de todas ellas de la que se obtiene el campo visual entero: las neuronas con campos receptivos mayores reaccionan a patrones que son combinaciones de los patrones de nivel inferior.

Estos estudios inspiraron el *neocognitrón* en 1980, el predecesor de lo que hoy en día se conoce como *redes neuronales convolucionales* [13]. Esta idea fue mejorada por Yann LeCun *et al.* en 1998, introduciendo la arquitectura LeNet-5, que muchos bancos utilizaron para reconocer números escritos a mano [14]. Además de las capas completamente conectadas y las funciones de activación mencionadas en la sección anterior, esta arquitectura incluye *capas convolucionales*, *filtros* y *capas de pooling*.

2.3.4.1 *Capas convolucionales*

Las capas convolucionales son el bloque fundamental de las CNN. La Fig. 2.15 representa que las neuronas de la primera capa convolucional no están conectadas a todos los píxeles de la imagen de entrada, sino únicamente a los que son parte de sus *campos receptivos*. De la misma manera, las neuronas de la segunda capa solo se conectan a una pequeña región de las de la primera. Esta arquitectura permite que la primera capa pueda *ver* los detalles más finos, y unirlos formando patrones más complejas que recibirán las siguientes capas.

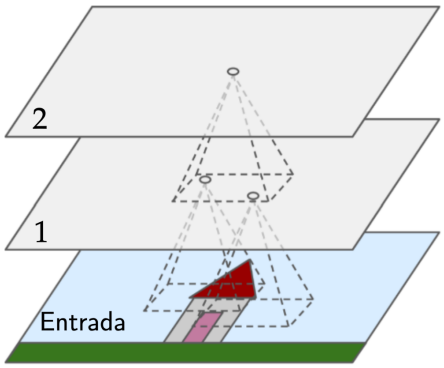
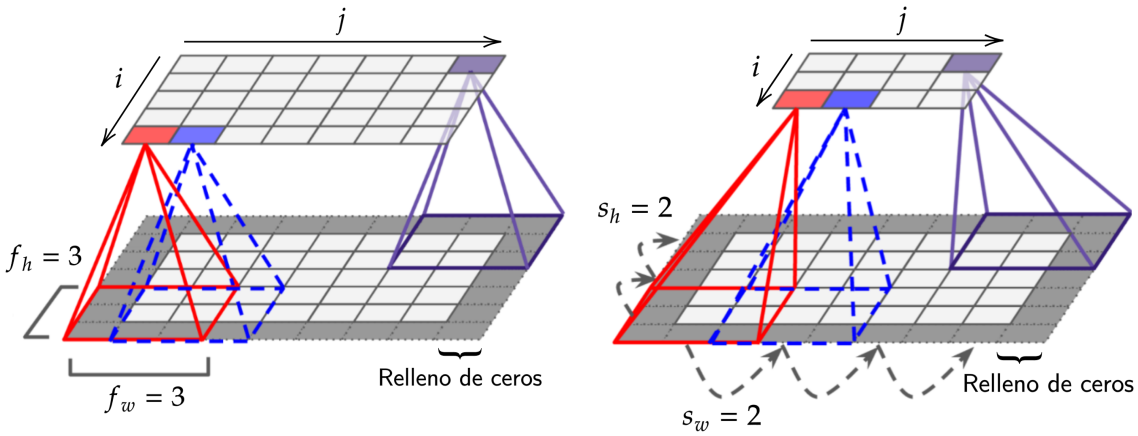


Figura 2.15: Capas convolucionales con campos receptivos rectangulares [8].

La neurona en la posición (i, j) de una capa está conectada a las salidas de las neuronas de la capa anterior en la región entre las filas i y $i + f_h - 1$ y las columnas j y $j + f_w - 1$, donde f_h y f_w son la altura y el ancho del campo receptivo (ver la Fig. 2.16a). Se suelen añadir ceros en los bordes de las entradas, para conservar las mismas dimensiones de la capa anterior. Esto se denomina *relleno de ceros*.

De manera parecida, es posible conectar una capa de entrada a otra mucho más pequeña aumentando la distancia entre los campos receptivos. Este desplazamiento de un campo al siguiente se llama *paso de avance* o *stride*. Véase la Fig. 2.16b. Una neurona en la posición (i, j) de la capa superior se conecta a las salidas de las neuronas de la capa anterior ubicadas entre las filas $i \cdot s_h$ y $i \cdot s_h + f_h - 1$ y las columnas $j \cdot s_w$ y $j \cdot j_w + f_w - 1$, donde s_h

y s_w son los *strides* horizontales y verticales. Esta operación reduce significativamente la complejidad del modelo, puesto que se reduce la cantidad de parámetros necesarios.



(a) Campos receptivos de 3×3 y paso de avance 1. **(b)** Campos receptivos de 3×3 y paso de avance 2.

Figura 2.16: Conexiones entre capas convolucionales. Adaptado de [8].

2.3.4.2 *Filtros*

Los pesos de una neurona pueden representarse con una pequeña imagen del tamaño de los campos receptivos. A continuación se muestran dos posibles conjuntos de pesos, llamados *filtros* o *kernels* convolucionales. Ambos son matrices de 7×7 rellenas de ceros, excepto las franjas blancas, que son columnas o filas de unos.

Las neuronas que utilicen el primero ignorarán todo su campo receptivo, excepto la línea vertical central, que serán las únicas entradas que no se multipliquen por cero. El segundo es un filtro horizontal; solo lo pasarán las entradas que correspondan a la línea horizontal.

(a) Imagen original. **(b)** Filtro vertical. **(c)** Filtro horizontal.

Figura 2.17: Efecto de los filtros. La imagen muestra las escaleras exteriores de la biblioteca del campus de Leioa de la UPV-EHU. Es un aumento de la esquina inferior derecha de la fotografía de la Fig. 2.18. El filtro vertical resalta las líneas verticales y el horizontal las horizontales.

Si todas las neuronas de una capa usan el mismo filtro vertical y el mismo término de sesgo, y la entrada de la red es la Fig. 2.17a, se obtendrá el resultado de la Fig. 2.17b: todo se difumina, excepto las líneas verticales. El resultado de utilizar el filtro horizontal se

muestra en la Fig. 2.17c. Por lo tanto, una capa de neuronas que utiliza el mismo filtro produce un *mapa de características*, que destaca las áreas que más activan ese filtro.

Estos filtros no se establecen manualmente. Durante el entrenamiento, la propia capa convolucional aprenderá qué filtros son los más adecuados para la tarea, y las capas superiores aprenderán a combinar estos formando patrones más complejos.

En la práctica cada capa tiene más de un filtro, y cada filtro produce un mapa de características. Todas las neuronas de un filtro comparten los mismos valores de los parámetros, por lo que se reduce drásticamente el número de parámetros requeridos. El campo receptivo de cada neurona se extiende por todos los filtros de la capa anterior.

Más precisamente, la salida $z_{i,j,k}$ de una neurona en la posición (i, j) del mapa de características k de una capa l depende de las neuronas de la capa anterior $l-1$, posicionadas entre las filas $i \cdot s_h$ y $i \cdot s_h + f_h - 1$ y las columnas $j \cdot s_w$ y $j \cdot s_w + f_w - 1$, en cada mapa de la capa $l-1$. De esta manera, todas las neuronas colocadas en la posición (i, j) de mapas consecutivos están conectadas a las salidas de las mismas neuronas de la capa anterior.

$$z_{i,j,k} = b_k + \sum_{u=0}^{f_h-1} \sum_{v=0}^{f_w-1} \sum_{k'=0}^{f_{n'}-1} x_{i',j',k'} \cdot w_{u,v,k',k} \quad \text{con} \begin{cases} i' = i \cdot s_h + u \\ j' = j \cdot s_w + v \end{cases} \tag{2.27}$$

En esta ecuación b_k es el término de sesgo del mapa k de la capa l. $x_{i',j',k'}$ es la salida de la neurona del mapa k' en la posición (i', j'). $w_{u,v,k',k}$ es el peso que corresponde a cualquier neurona del mapa k y la neurona del mapa k' de la capa $l-1$, en la posición (u, v). f_h, f_w y s_h, s_w son la altura y anchura y los pasos de avance vertical y horizontal, respectivamente. $f_{n'}$ es el número de mapas de características de la capa anterior $l-1$.

2.3.4.3 *Capas de pooling*

La finalidad de estas capas es la de disminuir la dimensión de la imagen, para reducir la memoria y el número de parámetros necesarios. La reducción del número de parámetros limita el riesgo de sobreajuste.

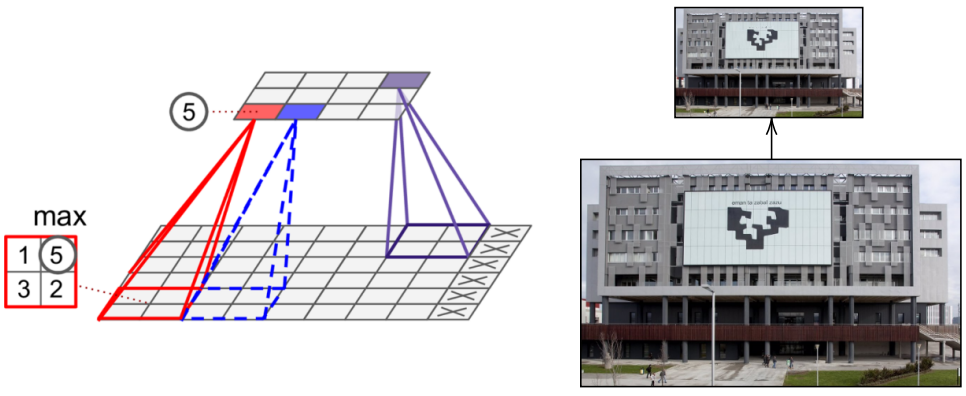


Figura 2.18: Capa de *max pooling*, con *kernel* de *pooling* de 2×2, paso de avance de 2 y sin relleno de ceros. La imagen que se obtiene a la salida tiene la mitad de la altura y la anchura que la original. Adaptado de [8].

De la misma manera que en las capas convolucionales, cada neurona de una capa de *pooling* está conectada a las salidas de un grupo limitado de neuronas de la capa anterior, que son las que pertenecen al campo receptivo. Se define su tamaño, el paso de avance y tipo de relleno de ceros igual que antes. No obstante, una neurona de este tipo no tiene pesos, pues su función es agregar los valores que recibe calculando el máximo o la media.

La Fig. 2.18 muestra una capa de *max pooling*, que son las que habitualmente se utilizan. Únicamente el valor máximo llega a la capa siguiente, las otras entradas se descartan.

2.3.4.4 *Arquitecturas de las redes neuronales convolucionales*

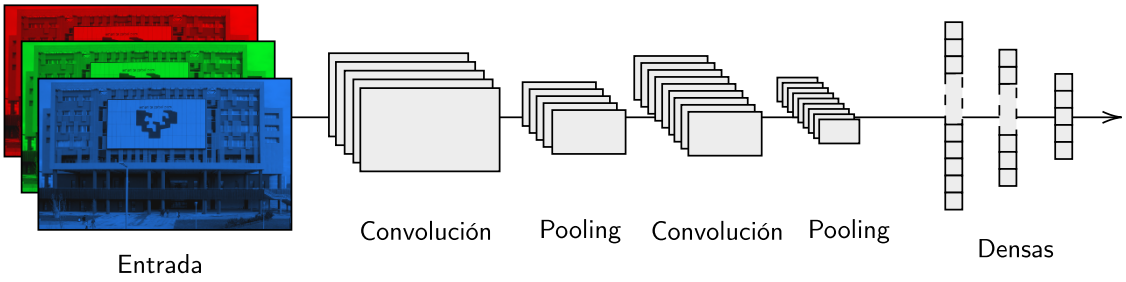


Figura 2.19: Arquitectura típica de una CNN. Adaptado de [8].

Para construir una red neuronal convolucional se apilan varias capas convolucionales, cada una de ellas seguida de una capa de ReLU y una capa *pooling*. Esto se repite de manera sucesiva, de forma que a medida que la imagen atraviesa la red va haciéndose más pequeña. Es cierto que a pesar de que su área disminuye, cada vez se vuelve más profunda, con más mapas de características, debido a las capas convolucionales.

Al final de esta secuencia de capas se añade una red compuesta por capas completamente conectadas con funciones de activación ReLU, como las que se han visto al tratar los MLP. La capa final genera como salida la predicción, utilizando en el caso de la clasificación multiclase una capa *softmax*.

3

DESARROLLO PRÁCTICO

En este capítulo se explica cómo se han llevado a la práctica los conceptos tratados en la parte anterior. Se empieza describiendo las características de la base de datos utilizada. Después, se explica el preprocesamiento que ha sido necesario dar a los datos para poder entrenar los modelos con ellos. A esto le sigue un apartado en el que se exponen las técnicas utilizadas para evaluar el funcionamiento de los modelos, y finalmente, se comentan los aspectos fundamentales a tener en cuenta a la hora de implementar los algoritmos para entrenar este tipo de sistemas.

3.1 ANÁLISIS EXPLORATORIO DE LOS DATOS

Como se ha explicado al describir la tarea elegida para este trabajo, la base de datos utilizada para entrenar todos los modelos es la base de datos HASY, creada por Martin Thoma [6, 7]. Está compuesta de 168.233 muestras de 369 clases distintas de símbolos escritos a mano, como los que componen la Fig. 2.2. Estas clases incluyen el abecedario latino, tanto en mayúsculas (A-Z) como en minúsculas (a-z), las cifras arábigas (0-9), el alfabeto griego, 32 tipos distintos de flechas, paréntesis, corchetes, llaves, etc.

Son imágenes en formato `png` en blanco y negro, de 32 px × 32 px. Por lo tanto, cada dato tiene $32 \cdot 32 = 1024$ características que pueden tomar los valores 0 (negro) o 255 (blanco). De media, solo el 16 % de los píxeles son negros, aunque depende mucho de la clase: en el símbolo "…" son el 3.7 % y en "■" el 59.2 %. Todas las imágenes están etiquetadas, como muestra la Fig. 3.1, cada una con su comando de LaTeX correspondiente.

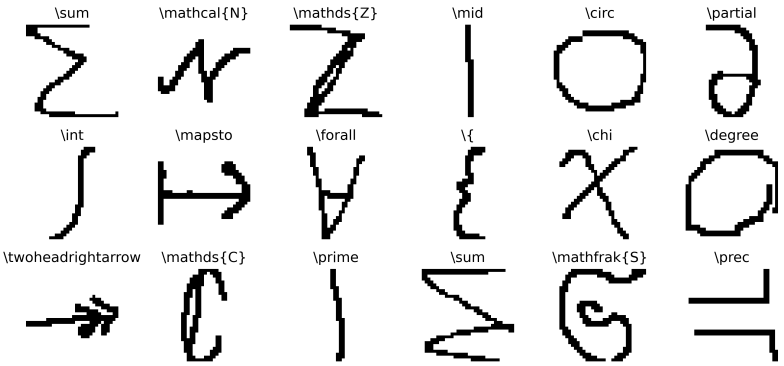


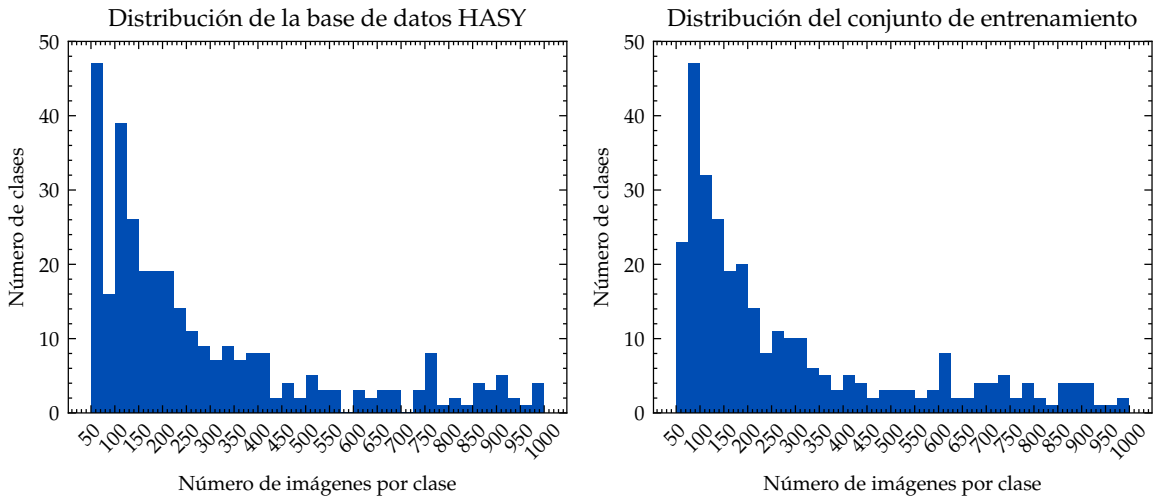
Figura 3.1: La base de datos HASY está compuesta de imágenes en blanco y negro de 32 px × 32 px, cada una etiquetada con su comando de LaTeX.

La dificultad fundamental que conlleva el uso de esta base de datos es el gran desequilibrio que existe entre la cantidad de imágenes de las distintas clases. Hay 47 clases con más de 1.000 muestras cada una. Los diez símbolos con mayor representación son:

$$\int, \sum, \infty, \alpha, \xi, \equiv, \partial, \mathbb{R}, \in, \square. \tag{3.1}$$

Hay 26.780 imágenes de estos diez símbolos, lo que supone el 16 % de la base de datos. El problema está en que, por otro lado, hay muchas otras clases que solo tienen 50 muestras.

Este hecho tendrá consecuencias drásticas en los modelos entrenados con estas imágenes. En la Fig. 3.2a se muestra la distribución de todos los datos.



(a) Distribución de la base de datos HASY.

(b) Distribución del conjunto de entrenamiento.

Figura 3.2: Distribución de los datos. Las 47 clases con más de 1.000 muestras no se han representado. Esta distribución ha sido respetada para generar el conjunto de entrenamiento.

Se han analizado distintas muestras elegidas aleatoriamente, como las de la Fig. 3.1 y las etiquetas correspondientes. De esta manera, se ha detectado que existen unos pocos errores en las etiquetas de las imágenes, que no ha sido posible corregir. Se ha observado también que todas las imágenes están procesadas de forma que los símbolos estén centrados, ocupando el mayor área posible, pero sin salirse del marco.

La base de datos contiene todas las imágenes en formato `.png` en la carpeta `hasy-data`, y las etiquetas se almacenan en el archivo `hasy-data-labels.csv`. En él se especifica la ruta de cada imagen, el comando de LaTeX que le corresponde, el identificador del usuario que dibujó esa imagen y un `symbol_ID` que identifica cada una de las 369 clases. La correspondencia entre el identificador de cada clase y el comando de LaTeX se resume en el archivo `symbols.csv`.

3.2 PREPROCESAMIENTO DE LOS DATOS

Para trabajar con todos estos datos, las funciones de Python definidas en el archivo `hasy_tools.py` han sido de mucha ayuda. Han sido escritas por el propio autor de la base de datos. Por problemas de compatibilidad con las versiones del software que este código utiliza, ha sido necesario revisarlo y actualizar las partes que se habían quedado obsoletas, e incluso se ha añadido alguna función nueva. Todo esto se recoge en el archivo `hasy_tools_updated.py`.

De esta manera, se ha dispuesto de herramientas que permiten convertir las imágenes a matrices de números, generar diccionarios que relacionan los nombres de los archivos con los comandos de los símbolos correspondientes, ordenar los datos por clases u obtener información sobre la distribución del número de imágenes por clase, necesaria para generar las imágenes de la Fig. 3.2.

Con todo esto, se ha procedido a ordenar los datos de una manera conveniente. El primer paso ha sido dividir todos los datos en dos conjuntos, el de *entrenamiento*, utilizado

para entrenar los modelos, para que "aprendan" los patrones que deben reconocer, y el de *prueba* o *test*, formado por imágenes que los modelos no "ven" mientras se entrenan, que se emplea para evaluar su rendimiento de manera más objetiva.

La base de datos HASY contiene una división de estos dos conjuntos, pero visto que no era la más conveniente para el tipo de entrenamiento que se pretendía realizar, se ha considerado oportuno utilizar los conjuntos de entrenamiento y prueba hechos *a medida*. Para ello, la función `create_stratified_train_test()` ha sido indispensable, pues ha permitido realizar la división entre los dos grupos respetando la distribución de la Fig. 3.2a. En la Fig. 3.2b se muestra la distribución que corresponde al conjunto de entrenamiento que se ha creado, que sigue la misma tendencia. Se ha utilizado el 80 % de los datos para formar este grupo, y el 20 % restante para generar el conjunto de prueba [15].

El siguiente paso ha sido estudiar la manera más conveniente de cargar las imágenes a estructuras de datos que los modelos que se quieren entrenar admiten. Tras probar con los DataFrames del paquete Pandas, se ha optado por emplear los vectores y matrices de NumPy. Se ha creado la función `load_database()` que lee el archivo `.csv` que se le proporciona y convierte las imágenes que en él se enumeran en matrices. Genera también un vector de etiquetas.

Esta función utiliza funciones secundarias que se recogen en `hasy_tools_updated.py`. En este punto ha surgido un problema, que es la ambigüedad en los identificadores de las imágenes. Las funciones originales de `hasy_tools.py` utilizan dos sistemas distintos. Por un lado, los `symbol_ID` mencionados anteriormente, que a pesar de corresponder a 369 clases, toman valores entre 32 y 1400. Se desconoce la razón de esta elección por parte del autor de la base de datos.

Por otro lado, se utiliza también un sistema de índices que se generan en el momento de leer el archivo `.csv`. Estos dependen de la posición en la que aparece cada tipo de símbolo por primera vez. Así, es posible que al leer `datos_entrenamiento.csv` el símbolo "α" aparezca el primero y se le adjudique el índice `1`, pero que en `datos_prueba.csv` sea el octavo y se le adjudique el `8`. Eso sí, en ambos casos el `symbol_ID` sería el mismo, `82`.

Con el fin de compatibilizar ambos sistemas se ha optado por producir dos diccionarios al llamar la función `load_database()`: `symbol_id2index` y `index2symbol_id`. De esta manera no ha sido necesario modificar las funciones originales de `hasy_tools.py` y se ha obtenido una manera de pasar fácilmente de un sistema a otro. Para recuperar los comandos de LaTeX, que son el objetivo final del trabajo, se genera otro diccionario por medio de la función `get_symbolid2latex()`.

Con estas herramientas, el proceso de cargar los datos para entrenar un modelo es el siguiente. Primero, se obtienen las matrices generadas a partir de las imágenes, las etiquetas (utilizando el sistema de índices) y los diccionarios mencionados anteriormente. Todo esto se hace partiendo de los archivos `.csv` que contienen las listas de imágenes pertenecientes a cada conjunto.

```
X_train,y_train,symbol_id2index_train,index2symbol_id_train=loaddatabase('train.csv')
X_test,y_test,symbol_id2index_test,index2symbol_id_test=loaddatabase('test.csv')
```

Después, se realiza la traducción de los índices a los identificadores.

```
y_train_id = np.array([index2symbol_id_train[element] for element in y_train])
y_test_id = np.array([index2symbol_id_test[element] for element in y_test])
```

Dependiendo del modelo, será más adecuado utilizar un sistema de etiquetas u otro. Por ejemplo, con los árboles de decisión utilizar los IDs resulta más cómodo, pero para entrenar redes neuronales conviene que las etiquetas sean números consecutivos, es decir, índices. Para solucionar el problema de que el mismo símbolo tenga índices distintos en

el conjunto de entrenamiento y en el de prueba, se traducen los IDs, que son iguales en ambos casos, a un único sistema de índices:

```
y_train = np.array([symbol_id2index_train[element] for element in y_train_id])
y_test = np.array([symbol_id2index_train[element] for element in y_test_id])
```

A continuación, se les aplica una transformación importante a los datos: el *escalado de características*. Salvo algunas excepciones como los *random forests*, los algoritmos de ML tienen un rendimiento peor cuando los atributos numéricos de entrada tienen escalas muy diferentes. En el contexto de este trabajo no es un factor tan crítico puesto que todas las características toman valores en el mismo rango $(0, 255)$, pero de todas maneras se ha considerado conveniente normalizar los datos para tomen valores entre 0 y 1:

```
X_train, X_test = X_train / 255., X_test/ 255.
```

La mayoría de modelos toman como entrada vectores unidimensionales de tamaño 1024. En el caso de las CNN, las entradas deben ser matrices bidimensionales, por lo que se le cambia la forma a los vectores que contienen los datos:

```
X_train = X_train.reshape(X_train.shape[0],32,32)
X_test = X_test.reshape(X_test.shape[0],32,32)
```

3.3 EVALUACIÓN DEL RENDIMIENTO

La construcción y la evaluación del rendimiento de los modelos se ha hecho en varios pasos. Primero, se han ajustado los hiperparámetros de los sistemas para optimizar el funcionamiento de cada uno, y adaptarlos a la base de datos empleada. Tras establecer los mejores valores de los hiperparámetros, se ha utilizado la *validación cruzada* para obtener estadísticas robustas del rendimiento de cada modelo. Todo esto se ha hecho en el conjunto de entrenamiento, y se ha reservado el conjunto de prueba para hacer las evaluaciones finales, al terminar todo el proceso.

3.3.1 *Ajuste de los hiperparámetros*

Los *hiperparámetros* son parámetros del algoritmo de aprendizaje. No se ven afectados por el proceso de entrenamiento: se establecen antes de que este empiece y se mantienen constantes durante el mismo [8]. Por ejemplo, el número de ramificaciones o la profundidad de un árbol de decisión, o el número de capas y la tasa de aprendizaje de una red neuronal son hiperparámetros. Pueden utilizarse para regular tanto el sobreajuste (*overfitting*) como el subajuste (*underfitting*).

La manera de seleccionar unos valores adecuados es entrenar el modelo en varios casos diferentes, y quedarse con el que mejor resultados dé. Para hacer esto de forma sistemática, se ha utilizado una *búsqueda exhaustiva*. En el caso de que un modelo tenga dos hiperparámetros a y b, este método consiste en proporcionar dos listas de valores,

$$\begin{aligned} a &: a_1, \ldots, a_n. \\ b &: b_1, \ldots, b_m. \end{aligned} \tag{3.2}$$

y después entrenar el modelo que se esté estudiando para para cada una de las $n \cdot m$ combinaciones: (a_1, b_1), (a_1, b_2), $\ldots$, (a_n, b_m). Finalmente, se selecciona la que mejor resultados da, y se pasa a la evaluación del modelo.

3.3.2 *Validación cruzada*

El conjunto de prueba no se debe utilizar hasta el final del proceso de construcción de un modelo, cuando llegue el momento en el que se han terminado todos los ajustes y se quiere obtener una evaluación objetiva, empleando datos que son nuevos para el modelo.

Es por esto que sería incorrecto utilizar el conjunto de prueba al ajustar los hiperparámetros: si estos modelos se entrenan en todo el conjunto de entrenamiento y posteriormente se elige el que mejores puntuaciones obtiene en el conjunto de prueba, esta elección estará condicionada por las características del propio conjunto de prueba. Se estarían obteniendo unos datos del rendimiento excesivamente optimistas, que luego no se cumplirían al proporcionar al modelo datos realmente nuevos.

Habitualmente el propio conjunto de entrenamiento se vuelve a dividir, en una parte que se utiliza para entrenar el modelo y otro conjunto de *validación*, que sigue la misma filosofía del conjunto de prueba y sirve para evaluar el modelo sin tener que recurrir a los datos se prueba.

En la práctica, para obtener estadísticas del rendimiento más sólidas, esta partición se hace varias veces, se entrena el modelo en cada una de ellas, y después se calculan la media y la desviación estándar de los resultados obtenidos. En este trabajo en particular, el conjunto de entrenamiento se ha dividido en cinco subconjuntos, de manera que el modelo se entrena cinco veces, utilizando como conjunto de validación un subconjunto diferente cada vez. Así, se respeta la proporción $80-20\,\%$ utilizada también anteriormente [15].

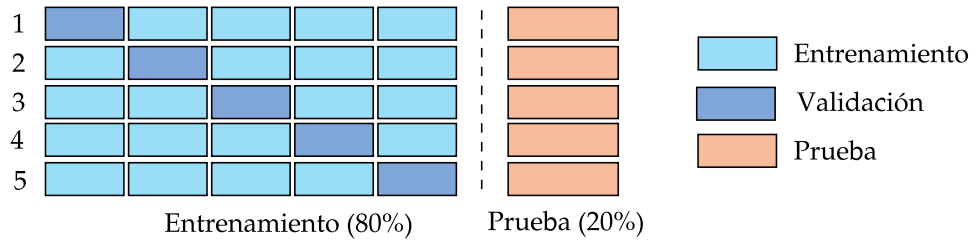


Figura 3.3: División de la base de datos. El conjunto de prueba no se utiliza hasta el final del proceso. Para evaluar los modelos se entrenan cinco veces utilizando distintas regiones del conjunto de entrenamiento y se mide su rendimiento en el conjunto de validación. Adaptado de [16].

3.3.3 *Métricas de rendimiento*

La tarea elegida en este trabajo es un problema de *clasificación*, por lo que la información más importante es el número de muestras que un modelo es capaz de clasificar correctamente de todas las que se le proporcionan. El ratio de predicciones correctas es la *exactitud* (*accuracy*). Si para n muestras con etiquetas y_i el modelo genera las predicciones $\hat{y}_i$:

$$\text{Exactitud} = \frac{1}{n}\sum_{i=1}^{n}\begin{cases} 1, & \hat{y}_i = y_i \\ 0, & \hat{y}_i \neq y_i \end{cases} \tag{3.3}$$

En los modelos que devuelven una lista de predicciones, cada una con una probabilidad, también se han calculado las veces en que la clase correcta está entre las tres y cinco primeras predicciones. Estos valores se indican con TOP3 y TOP5. Con esta notación, la exactitud se escribe TOP1.

Se debe tener en cuenta otra dificultad que el tipo de datos con los que se está trabajando supone. Hay situaciones en las que dos símbolos distintos, como `\sum` y `\Sigma` tienen el mismo glifo[1] Σ. Incluso cuando el glifo es distinto, hay símbolos que son indistinguibles al escribirse a mano, como por ejemplo `\alpha` (α), `\propto` ($\propto$) y `\varpropto` ($\varpropto$), `\Delta` (Δ) y `\triangle` ($\triangle$) o `\in` ($\in$) y `\epsilon` (ϵ).

Para realizar una evaluación más benevolente, que tenga en cuenta la imposibilidad de diferenciar entre algunas clases, se ha diseñado una función que calcula la exactitud combinada (*merged accuracy*), abreviada como MER. Para ello, se han definido 58 grupos de clases equivalentes, de manera que si el modelo predice una clase $\hat{y}_i$ que no es la real, pero pertenece a su grupo equivalente $\text{Equiv}(y_i) = \{y_i, y_i', y_i'', ...\}$, se cuente como acierto. Esta agrupación es subjetiva, se ha hecho buscando de uno en uno los símbolos que pueden crear confusión, por lo que no debe tomarse como una clasificación absoluta. Ver la tabla A.1 de símbolos equivalentes del Apéndice para más información. De esta manera,

$$\text{MER} = \frac{1}{n}\sum_{i=1}^{n}\begin{cases} 1, & \hat{y}_i \in \text{Equiv}(y_i) \\ 0, & \hat{y}_i \notin \text{Equiv}(y_i) \end{cases} \tag{3.4}$$

Cabe destacar que cuando se trabaja con bases de datos desequilibradas, la exactitud puede llevar a confusiones: en una base compuesta de 80 datos de tipo A y 20 de tipo B, un sistema que siempre prediga "A" tendrá una exactitud del 80 %, a pesar de no ser capaz de reconocer elementos del tipo B. Es más correcto utilizar la *precisión* y la *sensibilidad*:

$$\text{Precisión} = \frac{VP}{VP + FP}, \tag{3.5}$$

$$\text{Sensibilidad} = \frac{VP}{VP + FN}. \tag{3.6}$$

VP es el número de verdaderos positivos, FP el de falsos positivos y FN el de falsos negativos. La precisión es la exactitud de las predicciones positivas. En el caso de la base de datos binaria mencionada como ejemplo, la precisión indica la proporción de las veces que el sistema acierta cuando produce "A". La sensibilidad es el ratio de instancias positivas detectadas, la probabilidad de que un elemento que realmente sea de tipo A sea clasificado como "A". Ambas se suelen combinar para dar el *valor F1*, que no es más que su media harmónica.

$$\text{Valor F1} = \frac{2}{\frac{1}{\text{Precisión}} + \frac{1}{\text{Sensibilidad}}}. \tag{3.7}$$

3.4 IMPLEMENTACIÓN DE LOS ALGORITMOS

Los modelos de este trabajo y los algoritmos para entrenarlos no se han construido de cero, sino que se han importado de las librerías Scikit-learn, TensorFlow y Keras. En las próximas líneas se han reunido los modelos de ML que se han utilizado a lo largo de este proyecto. Los resultados obtenidos se exponen en el capítulo siguiente. Para más detalles, puede consultarse el código disponible en el repositorio de GitHub[2].

1 En tipografía, un *glifo* es una representación gráfica de uno o varios caracteres: a, a, *a*, a, **a**, ...
2 Enlace: https://github.com/BeBerasategi/Reconocimiento-simbolos-matematicos/.

3.4.1 *Máquinas de vectores soporte*

Se ha utilizado la clase LinearSVC de Scikit-learn. Se han hecho pruebas con otros modelos como SVC, pero al ser demasiado lentos para la base de datos de este trabajo se han tenido que descartar.

```
from sklearn.svm import LinearSVC
svm = LinearSVC(C=c)
svm.fit(X_train, y_train)
```

El efecto de variar el hiperparámetro de regularización `C` se discute en el próximo capítulo.

3.4.2 *Árboles de decisión y random forests*

```
from sklearn.ensemble import RandomForestClassifier
rf=RandomForestClassifier(n_estimators=n, max_depth=depth)
rf.fit(X_train,y_train_id)
```

En el capítulo siguiente se verán las consecuencias de variar tanto el número de árboles de decisión `n_estimators` que se ensamblan como la profundidad máxima `max_depth` de cada uno de ellos.

3.4.3 *Perceptrones multicapa*

```
from tensorflow import keras
model_nm = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[32*32]),
    keras.layers.Dense(n, activation="relu"),
    keras.layers.Dense(m, activation="relu"),
    keras.layers.Dense(369, activation="softmax")
    ])
```

La capa de entrada se compone de $32 \cdot 32 = 1024$ neuronas. Esta arquitectura está compuesta de dos capas ocultas, de `n` y `m` neuronas cada una. Como son capas densas, cada neurona se conecta a todas las de la capa anterior. La capa de salida tiene 369 neuronas, una por cada clase. En el capítulo siguiente se analizarán los cambios en el rendimiento del modelo al modificar `n` y `m`.

3.4.4 *Redes neuronales convolucionales*

```
from tensorflow import keras
CNN_MNIST = keras.models.Sequential([
        keras.layers.Conv2D(32, kernel_size=3, padding="same", activation="relu"),
        keras.layers.Conv2D(64, kernel_size=3, padding="same", activation="relu"),
        keras.layers.MaxPool2D(),
        keras.layers.Flatten(),
        keras.layers.Dropout(0.25),
        keras.layers.Dense(128, activation="relu"),
        keras.layers.Dropout(0.5),
        keras.layers.Dense(369, activation="softmax")
        ])
```

Este modelo está compuesto de capas convolucionales, capas de *pooling* para reducir la dimensionalidad, y capas densas como las del perceptrón multicapa. Se añaden también capas de *dropout*, cuya función es desactivar una proporción dada de las neuronas únicamente durante el entrenamiento. Esto se hace como técnica de regularización con el fin de evitar el sobreajuste. El `kernel_size` determina el tamaño del campo visual, `padding="same"` indica que el relleno de ceros se hace de manera que el tamaño de la capa se mantenga. Finalmente, se especifica la función de activación, ReLU para las capas ocultas y *softmax* para la capa de salida.

También se han probado modelos menos elaborados compuestos únicamente de 2, 3 y 4 capas convolucionales, que se describen a continuación.

```
CNN_2_capas = keras.models.Sequential([
        keras.layers.Conv2D(32, kernel_size=3, padding="same", activation="relu"),
        keras.layers.MaxPool2D(pool_size=(2, 2), strides=(2,2)),
        keras.layers.Flatten(),
        keras.layers.Dense(369, activation="softmax")
    ])
```

```
CNN_3_capas = keras.models.Sequential([
        keras.layers.Conv2D(32, kernel_size=3, padding="same", activation="relu"),
        keras.layers.MaxPool2D(pool_size=(2, 2), strides=(2,2)),
        keras.layers.Conv2D(64, kernel_size=3, padding="same", activation="relu"),
        keras.layers.MaxPool2D(pool_size=(2, 2), strides=(2,2)),
        keras.layers.Flatten(),
        keras.layers.Dense(369, activation="softmax")
    ])
```

```
CNN_4_capas = keras.models.Sequential([
        keras.layers.Conv2D(32, kernel_size=3, padding="same", activation="relu"),
        keras.layers.MaxPool2D(pool_size=(2, 2), strides=(2,2)),
        keras.layers.Conv2D(64, kernel_size=3, padding="same", activation="relu"),
        keras.layers.MaxPool2D(pool_size=(2, 2), strides=(2,2)),
        keras.layers.Conv2D(128, kernel_size=3, padding="same", activation="relu"),
        keras.layers.MaxPool2D(pool_size=(2, 2), strides=(2,2)),
        keras.layers.Flatten(),
        keras.layers.Dense(369, activation="softmax")
    ])
```

El argumento `pool_size` indica las dimensiones del *kernel* de *pooling*, equivalente al campo visual de la capa convolucional, y `stride` el paso de avance en cada dirección.

4

RESULTADOS

En el proceso de elegir el modelo más adecuado para la tarea de este proyecto, primero se han construido los modelos que se han descrito al final del capítulo anterior. Después se han buscado los hiperparámetros que producen los mejores resultados, y se han evaluado los modelos utilizando la validación cruzada. En un punto, ha sido necesario solucionar el problema del desequilibrio del número de muestras entre clases. Finalmente, se ha construido una demo para probar el sistema que mejor rendimiento tiene.

4.1 OPTIMIZACIÓN DE HIPERPARÁMETROS

En este proceso se ha utilizado una sola partición del conjunto de entrenamiento (ver Fig. 3.3). Se buscan los hiperparámetros que permiten obtener la mayor exactitud en el conjunto de validación.

4.1.1 *Máquinas de vectores soporte*

En el caso de las máquinas de vectores soporte, se han entrenado varios modelos cambiando el hiperparámetro `C`, que condiciona la compensación entre obtener el mayor margen entre las clases y permitir las violaciones de margen. La Fig. 4.1 muestra que el mejor resultado en el conjunto de validación se logra con $C = 0.01$.

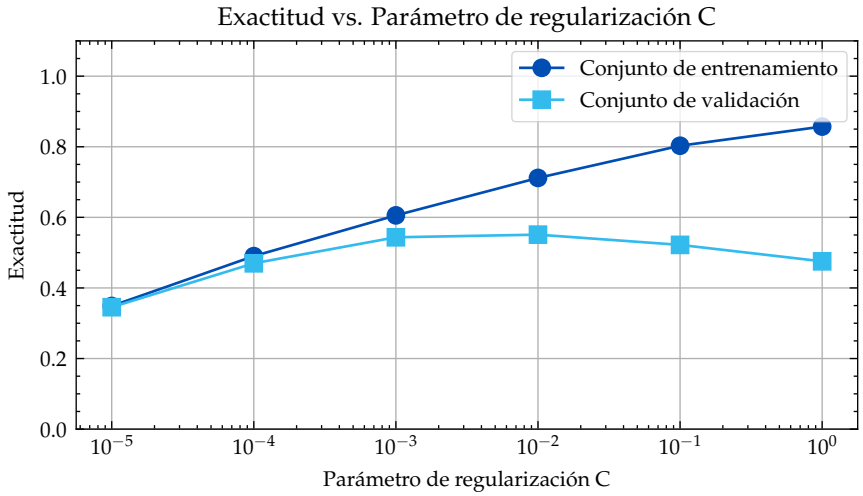
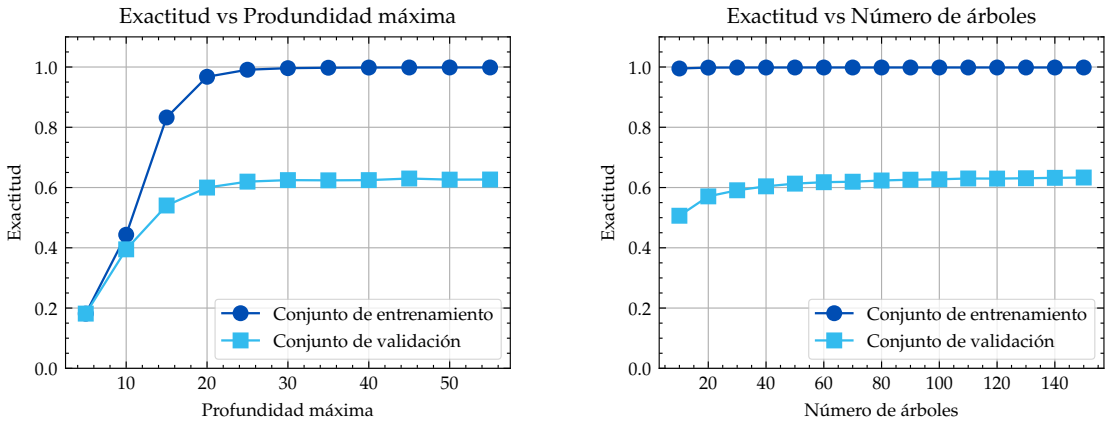


Figura 4.1: Ajuste del hiperparámetro `C` de una máquina de vectores soporte.

4.1.2 *Árboles de decisión y random forests*

En el caso de los *random forests*, primero se ha analizado el efecto de variar la profundidad máxima de los árboles de decisión, utilizando `n=100` de estos estimadores. Tras ver que se obtiene la mayor exactitud para una profundidad mayor de 45, se ha variado el número de

árboles que forman el *random forest*. Se ha concluido que 100 es un número apropiado. Los resultados de ambas pruebas se han representado en la Fig. 4.2.



(a) Efecto de cambiar la profundidad máxima de los árboles de decisión.

(b) Efecto de cambiar el número de árboles de decisión que componen el *random forest*.

Figura 4.2: Ajuste de hiperparámetros de un *random forest*.

4.1.3 *Perceptrones multicapa*

En el caso de los perceptrones multicapa, con la arquitectura vista en la sección anterior, se han hecho pruebas cambiando el número de neuronas n y m de la primera y segunda capas ocultas. Se han entrenado 4 modelos distintos, con valores de $(n, m) = (800, 500)$, $(500, 500)$, $(800, 800)$, $(500, 200)$. Cada modelo se ha entrenado 4 veces, utilizando una tasa de aprendizaje distinta cada vez: $0.005, 0.01, 0.02$ y 0.04. Esto quiere decir que en total se han entrenado 16 MLPs. El mejor resultado se ha obtenido utilizando el tercer modelo, con $n = m = 800$ y una tasa de aprendizaje de 0.02.

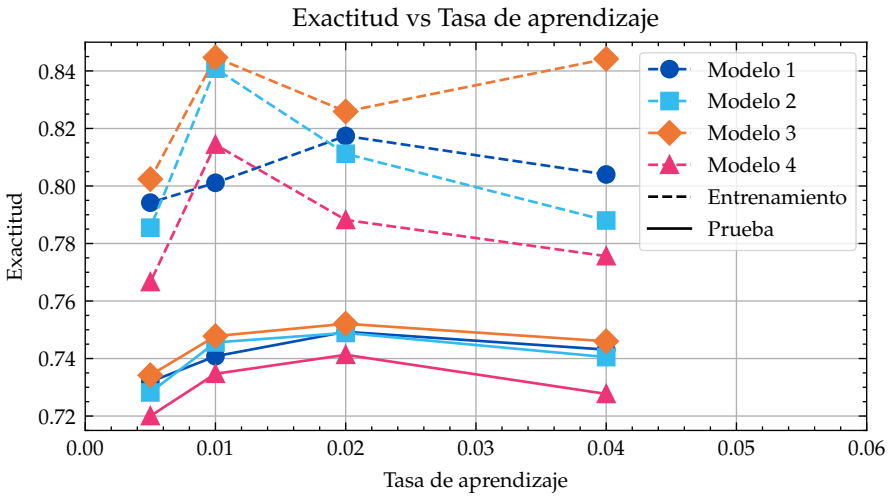


Figura 4.3: Exactitud de cuatro modelos de perceptrones multicapa, entrenados con distintos valores de la tasa de aprendizaje.

4.1.4 *Redes neuronales convolucionales*

En el caso de las redes neuronales convolucionales, no se ha hecho un ajuste de hiperparámetros de este tipo, sino que directamente se ha utilizado la validación cruzada para evaluar las cuatro arquitecturas descritas al final del capítulo anterior. Los resultados se exponen en la próxima sección.

4.2 VALIDACIÓN CRUZADA

Se ha empleado la validación cruzada de la manera que se ha representado en la Fig. 3.3, utilizando 5 particiones distintas del conjunto de entrenamiento, y entrenando los modelos en cada una de ellas. Esto permite conseguir una estimación más robusta de la exactitud de cada uno.

Como se ha explicado en el capítulo anterior, la exactitud (TOP1) no ha sido la única métrica utilizada, también se han medido la TOP3 y la TOP5. Además, para considerar las clases equivalentes, se ha calculado la métrica MER. Los resultados se resumen en las tablas 4.1 y 4.2.

Tabla 4.1: Resultados de la validación cruzada para las máquinas de vectores soporte, *random forests* y perceptrones multicapa.

	TOP1	TOP3	TOP5	MER
SVM	$54.7 \pm 0.2\,\%$	–	–	$61.0 \pm 0.2\,\%$
RF	$62.6 \pm 0.1\,\%$	$79.2 \pm 0.1\,\%$	$83.6 \pm 0.2\,\%$	$69.7 \pm 0.1\,\%$
MLP	$74.6 \pm 0.3\,\%$	$91.0 \pm 0.1\,\%$	$93.9 \pm 0.1\,\%$	$84.0 \pm 0.2\,\%$

Tabla 4.2: Resultados para CNNs con diferentes configuraciones de capas y el modelo MNIST.

	TOP1	TOP3	TOP5	MER
2 capas	$73.8 \pm 0.3\,\%$	$90.2 \pm 0.1\,\%$	$93.1 \pm 0.1\,\%$	$82.9 \pm 0.2\,\%$
3 capas	$78.3 \pm 0.5\,\%$	$93.2 \pm 0.1\,\%$	$95.5 \pm 0.1\,\%$	$87.6 \pm 0.1\,\%$
4 capas	$80.4 \pm 0.4\,\%$	$94.7 \pm 0.2\,\%$	$96.7 \pm 0.2\,\%$	$89.6 \pm 0.2\,\%$
Modelo MNIST	$80.3 \pm 0.2\,\%$	$94.6 \pm 0.2\,\%$	$96.6 \pm 0.2\,\%$	$88.8 \pm 0.1\,\%$

En la Tab. 4.1 la segunda y tercera columna de las SVM se han dejado vacías, dado que el modelo que se ha utilizado no produce probabilidades de que una imagen pertenezca a una clase, sino que devuelve una única predicción. En el resto de los casos sí que ha sido posible calcular el TOP3 y el TOP5.

Las SVM son los primeros modelos que se han utilizado, que para esta tarea no han dado buenos resultados. Además, han sido muy lentas de entrenar, por lo que se han descartado casi inmediatamente. Los *random forests* permiten obtener mejores resultados, y tienen la ventaja de ser rápidos de entrenar, pero no pueden competir con las redes neuronales. De los modelos analizados en la Tab. 4.1, los MLP son claramente los que mejor rendimiento presentan.

En la Tab. 4.2 se compara el funcionamiento de cuatro redes neuronales convolucionales, con las arquitecturas descritas al final del capítulo anterior. Al "modelo MNIST" se le ha

dado ese nombre porque es un modelo que se ha adaptado de [8], que originalmente se utilizaba para clasificar muestras de la base de datos MNIST.

Se puede observar que en los primeros tres modelos el rendimiento mejora al aumentar el número de capas ocultas. Dejando de lado la métrica MER, se ve que la exactitud del modelo con 4 capas y el modelo MNIST es prácticamente la misma.

La diferencia está en que el modelo de 4 capas es más sencillo y se entrena más rápido, mientras que las capas adicionales que tiene el modelo MNIST le permiten ser más resistente al sobreajuste.

Comparando las tablas 4.1 y 4.2, la superioridad de las CNN es evidente. En lo que sigue, se utilizará únicamente el modelo MNIST. Es junto con la CNN de cuatro capas el que mejores resultados ha dado.

Tras volver a entrenar el modelo, esta vez usando el conjunto de entrenamiento entero, se evalúa su rendimiento en el conjunto de prueba, que hasta este momento no se ha utilizado. Los resultados que se obtienen se resumen en la Tab. 4.3.

Tabla 4.3: Resultados de la evaluación en el conjunto de prueba del modelo MNIST entrenado en el conjunto de entrenamiento entero.

	TOP1	TOP3	TOP5	MER
Modelo MNIST	80.8 %	95.1 %	97.0 %	89.5 %

4.3 EQUILIBRIO DE DATOS DESEQUILIBRADOS

En este punto, se ha diseñado un prototipo de una aplicación en la que el usuario puede probar el sistema con símbolos dibujados por él mismo. Se ha implementado en un *notebook* de Jupyter, utilizando el *widget* `DrawingWidget` del paquete `ipycanvas_drawing`. La Fig. 4.4 muestra una de las primeras versiones de este prototipo.

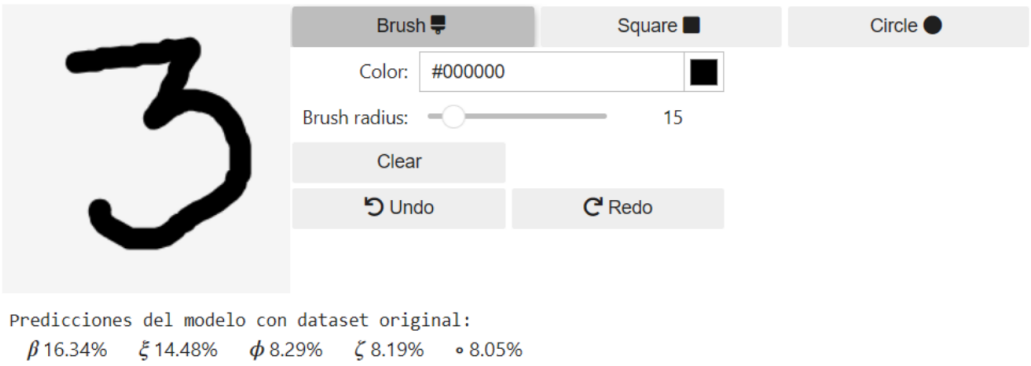


Figura 4.4: Prototipo de una posible aplicación. Se muestran las cinco opciones más probables, según el modelo MNIST.

En las pruebas que se han hecho de esta manera, ha quedado claro que por muy buenas que sean las exactitudes medidas en el apartado anterior, el sistema está sesgado debido al desequilibrio entre clases que se ha mencionado al describir la base de datos utilizada.

Consecuentemente, el modelo reconoce casi a la perfección los símbolos de los que más muestras ha recibido, pero se equivoca con aquellos que tienen menos representación, como se puede ver en la Fig. 4.4. Esto es un problema grave, ya que entorpece el funcionamiento del sistema. Refleja la importancia de tener una base de datos adecuada.

Para seguir avanzando ha sido necesario modificar la base de datos, intentando compensar ese desequilibrio. Las dos opciones principales para hacer esto son obtener más datos de las clases con menos representación y descartar muestras de aquellas que están más pobladas [17]. En este trabajo se ha utilizado una combinación de ambas técnicas.

Una manera cada vez más extendida de aumentar el número de datos es generarlos de manera sintética. Este método se denomina SMOTE, que es el acrónimo de *Synthetic Minority Oversampling Technique* [18]. Es un método popular, que produce buenos resultados al trabajar con bases de datos numéricas desequilibradas. Sin embargo, el procesamiento de imágenes supone ciertas dificultades adicionales, por este método no está tan desarrollado para este tipo de datos.

Se ha encontrado un artículo que describe el método DeepSMOTE para imágenes [19], más adecuado para lo que este proyecto requiere. No obstante, las pruebas que se han realizado no han sido satisfactorias, ya que el código que acompaña al artículo es algo difícil de comprender, y específico para imágenes de 28×28 píxeles. Es por esto que este método se ha tenido que dejar de lado.

Como alternativa se ha utilizado un método menos ambicioso, pero que ha sido capaz de producir buenos resultados: el *aumento de datos*. Consiste en desplazar, rotar, invertir y dimensionar cada imagen del conjunto de entrenamiento varias veces y añadir las imágenes resultantes a ese conjunto [8]. De esta manera se obtienen modelos más tolerantes respecto a las variaciones en la posición, la orientación y el tamaño de las imágenes.

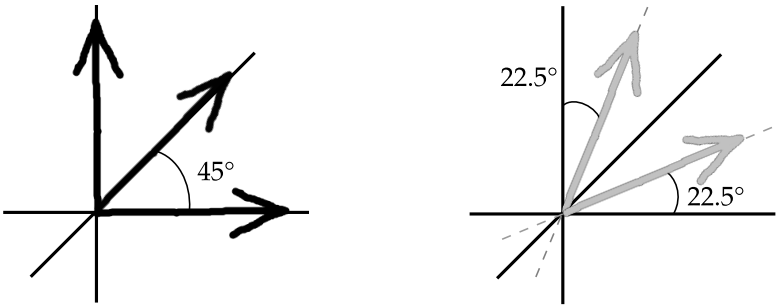


Figura 4.5: Las rotaciones tienen una restricción importante. Tomando el ejemplo de las flechas, se puede ver que una rotación de más de 22.5° entorpecería la capacidad de distinguir entre clases.

En el caso de los símbolos estas transformaciones se han tenido que elegir con atención. Las inversiones no son adecuadas, puesto que hay que distinguir símbolos como → y ←, ↻ y ↺ o ↑ y ↓. Las rotaciones se deben hacer teniendo en cuenta que el hecho que ↑, ↗ y → sean símbolos diferentes impone una restricción. Como la Fig. 4.5 demuestra, una rotación positiva o negativa de más de 22.5° llevaría a confusiones entre clases. Por precaución, el valor máximo de las rotaciones hechas en este trabajo ha sido de 10°. Finalmente, el cambio de escala de las imágenes se ha hecho solo de manera que su tamaño disminuyera, ya que las muestras de la base de datos HASY están hechas de forma que los símbolos ocupen el mayor área posible, y aumentar su tamaño significaría dejar algunas partes fuera del encuadre.

El submuestreo de las clases más densas se ha implementado de la manera más sencilla posible. Tras establecer el número de muestras n que todas las clases deben tener, en los casos de las clases que originalmente tenían más, simplemente se han seleccionado n muestras de manera aleatoria y sin reemplazo.

El símbolo con más muestras de la base de datos tiene 2811 imágenes. Se han generado cuatro bases de datos en los que todas las clases tienen 400, 600, 1000 y 2811 muestras respectivamente. Tras entrenar el modelo MNIST en ellas, se ha evaluado el modelo en el

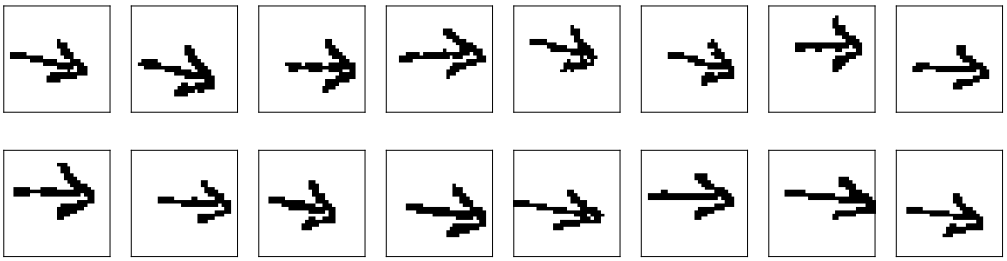


Figura 4.6: Se han generado muestras nuevas combinando aleatoriamente las transformaciones descritas. Rotaciones de $\pm 10°$, reducciones de escala entre el 10 % y el 60 % y desplazamientos verticales y horizontales del 15 %. Se han evitado las inversiones. Esta figura muestra el ejemplo de las muestras generadas a partir de una sola imagen.

conjunto de prueba de la base de datos original, el mismo que se ha utilizado para obtener la Tab. 4.3. Los resultados se pueden encontrar en la Tab. 4.4.

Tabla 4.4: Resultados de entrenar el modelo MNIST en bases de datos aumentadas de distinto tamaño. Puntuaciones obtenidas en el conjunto de prueba de la base de datos original.

Muestras	TOP1	TOP3	TOP5	MER
400	77.3 %	94.1 %	96.4 %	88.3 %
600	79.4 %	94.7 %	96.7 %	88.9 %
1000	79.5 %	94.9 %	97.0 %	89.3 %
2811	75.1 %	93.4 %	96.6 %	86.9 %

Finalmente, se han calculado también la precisión, la sensibilidad y el valor F1 de los modelos MNIST entrenados en la base de datos original y en la base de datos aumentada con 1000 muestras por clase. Para calcular estas métricas, primero se han calculado la precisión y sensibilidad de cada clase, utilizando las ecuaciones (3.5,3.6,3.7), y después se ha obtenido la media aritmética de cada una. Como ya se ha mencionado anteriormente, el valor F1 es la media armónica de estas dos cantidades.

Tabla 4.5: Métricas del modelo MNIST entrenado en la base de datos desequilibrada y en la base de datos aumentada de 1000 muestras por clase. El conjunto de prueba utilizado ha sido el mismo en ambos casos.

	Exactitud	Precisión	Sensibilidad	Valor F1
Modelo MNIST original	80.8 %	72.3 %	67.1 %	68.0 %
Modelo MNIST 1000	79.5 %	68.7 %	70.8 %	68.8 %

4.4 DISEÑO DE LA DEMO

Una vez identificados los modelos que mejor rendimiento tienen, se ha querido construir una interfaz web del estilo de Detexify [4]. Esto se ha hecho con el objetivo de poder experimentar de manera más interactiva con los sistemas obtenidos. Se ha decidido incluir los dos modelos de la Tab. 4.5, que tienen la misma arquitectura, pero se han entrenado en bases de datos distintas. Se espera que de esta manera uno pueda ver el efecto de tener una base de datos desequilibrada, de forma más intuitiva.

El elemento principal es el lienzo gris en el que el usuario puede dibujar un símbolo de los 369 que componen las bases de datos, haciendo clic izquierdo y moviendo el ratón, manteniéndolo presionado hasta que se termine de dibujar la imagen. Después, se pulsa el botón *predecir*, y la interfaz web devuelve las cinco primeras predicciones más probables de cada sistema, el TOP5. Con el botón *borrar* el lienzo se limpia y se puede volver a dibujar un símbolo nuevo.

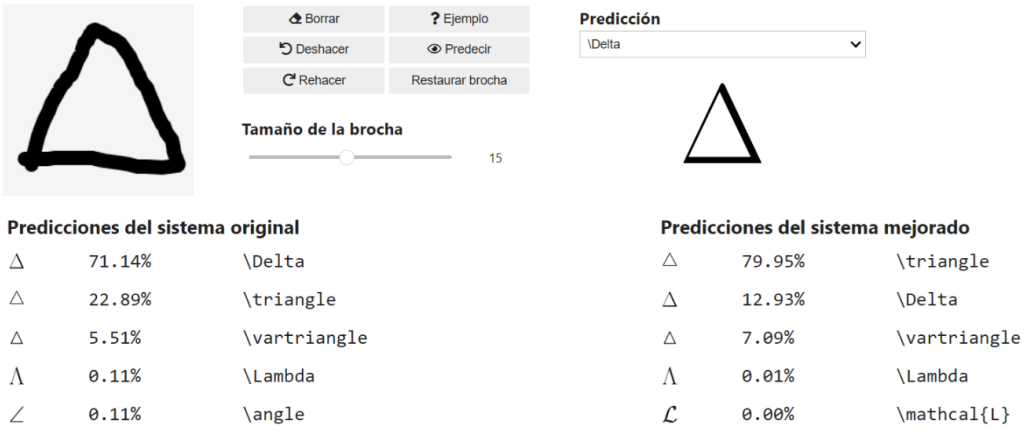


Figura 4.7: Captura de pantalla de la demo. El usuario dibuja un símbolo en el lienzo gris, utilizando el ratón. Para clasificar la imagen, se emplea la red neuronal convolucional MNIST entrenada en la base de datos original y en la aumentada. De esta manera, se han obtenido el *sistema original* y el *sistema mejorado*, que permiten ver el efecto de la base de datos desequilibrada.

Tras hacer varias pruebas, se ve que el sistema original, el que se ha entrenado en la base de datos desequilibrada, tiene tendencia a predecir ciertos símbolos. Resulta bastante evidente qué clases tienen menos muestras, pues el funcionamiento del sistema empeora considerablemente cuando la entrada es una imagen de esta categoría. En cambio, el sistema mejorado se ha entrenado utilizando la base de datos equilibrada. En consecuencia, su comportamiento es más homogéneo, no hay clases que sea capaz de acertar con mayor probabilidad, sino que en todas acierta o falla de manera parecida.

Hay que tener en cuenta que las imágenes que reciben los modelos de este sistema tienen características distintas a las de la base de datos original. Después de que el usuario dibuje el símbolo, se reduce la imagen de 200 px × 200 px a 32 px × 32 px. Haciendo varias pruebas se ha visto que cuando el tamaño de la brocha con la que el usuario dibuja es de 15 px, las imágenes que se obtienen tras la reducción de dimensiones son más parecidas a las de la base de datos (ver Fig. 4.8). De todas maneras, se ha añadido la opción de variar el tamaño de la brocha, para ver el efecto que esto tiene en el funcionamiento del sistema.

La predicción principal, que se encuentra en la parte superior derecha, es la primera predicción que hace el sistema mejorado. Se ha implementado un menú desplegable que aparece cuando la clase predicha pertenece a uno de los grupos de símbolos equivalentes que se han definido al describir la métrica MER. La lista de estos grupos se puede encontrar en la tabla A.1 del apéndice. Estos conjuntos están formados por símbolos de clases distintas que son muy difíciles o incluso imposibles de distinguir cuando se escriben a mano. Por eso, se ha decidido dar al usuario la opción de visualizar el símbolo de LaTeX que cada uno de estos comandos produce, utilizando el menú desplegable.

Una de las mayores dificultades a la hora de preparar esta demo ha sido la de obtener la imagen de cada símbolo, compilada con LaTeX. Se han hecho pruebas utilizando el propio *notebook* de Jupyter para generar el símbolo a partir del comando de LaTeX, pero

(a) Muestra de la base de datos. **(b)** Entrada dibujada. **(c)** Entrada reducida.

Figura 4.8: Se ha configurado la interfaz web para obtener unas imágenes lo más parecidas posibles a las de la base de datos.

los resultados no han sido buenos. Los comandos utilizados se encuentran en muchos paquetes de LaTeX distintos, y Jupyter contiene solo los básicos. Por eso, en muchos casos había errores de visualización.

Se ha decidido que la alternativa más segura es crear manualmente una imagen por cada símbolo. Ha sido una tarea que ha requerido tiempo, puesto que a pesar de que se ha intentado automatizar utilizando tanto Python como LaTeX, el hecho de tener que importar tantos paquetes ha creado incompatibilidades que se han tenido que resolver de una en una.

Las imágenes se han generado utilizando pdfLaTeX, en formato .pdf. Para que la interfaz web pueda mostrarlos correctamente, deben estar en formato .png. Se ha optado por convertirlas utilizando Inkscape, ya que el resto de alternativas que se han probado generaban imágenes de muy baja calidad. Ha habido alguna que ha quedado pixelada, por algún motivo que no se ha logrado comprender.

Otro contratiempo ha sido el hecho de que el sistema operativo Windows no diferencie las letras mayúsculas y minúsculas en los nombres de los archivos. Los nombres de las imágenes se han establecido según los comandos, de manera que `rightarrow.png` y `Rightarrow.png` corresponden a símbolos distintos: "$\rightarrow$" y "$\Rightarrow$". Desafortunadamente Windows trata los dos nombres como equivalentes y sobreescribe los archivos de este tipo. Por lo tanto, ha sido necesario utilizar Ubuntu, que sí diferencia entre archivos de este tipo, y que ha permitido evitar la sobreescritura.

El lienzo se ha implementado adaptando el código del *widget* `DrawingWidget` del paquete `ipycanvas_drawing` [20]. El resto del programa se ha escrito en un *notebook* de Jupyter. Finalmente, para darle aspecto de aplicación web, se ha empleado el software llamado `voila`, que se encarga precisamente de esto [21].

Todos los programas e imágenes de los símbolos se han subido a GitHub, de manera que mediante Binder cualquiera pueda probar la demo descrita en esta sección[1]. Ver el apéndice para más información sobre el repositorio.

1 Enlace: https://github.com/BeBerasategi/Reconocimiento-simbolos-matematicos.

5

CONCLUSIONES

En este trabajo se ha llevado a cabo el proceso de construir un modelo de aprendizaje automático de principio a fin, desde la selección de los datos hasta la evaluación de los modelos y la implementación de una interfaz web.

Se han estudiado las máquinas de vectores soporte (SVM), los árboles de decisión y los *random forests*, los perceptrones multicapa (MLP) y las redes neuronales convolucionales. Los resultados han sido claros: en tareas de clasificación de imágenes la superioridad de las redes neuronales convolucionales es evidente. Es precisamente por esto que son un elemento clave de la visión artificial (*computer vision*).

Las consecuencias de que la base de datos utilizada originalmente estuviese desequilibrada también han sido relevantes. El hecho de que hubiera tanta diferencia entre la cantidad de muestras por clase hace que los modelos entrenados con estos datos estén condicionados, que se especialicen en reconocer las clases con mayor número de muestras e ignoren las demás.

El aumento de datos ha sido decisivo para corregir este problema. Se ha utilizado una técnica bastante básica, que realmente no introduce información nueva al sistema, sino que lo expone a pequeños cambios en los datos que ya conoce. A pesar de la simplicidad de esta idea, los resultados han sido satisfactorios. En un futuro podrían probarse técnicas más complicadas de generación de datos como las que se han mencionado en el texto (SMOTE y DeepSMOTE), pues podrían mejorar la calidad de las muestras adicionales con las que se entrenan los modelos.

La comparación entre el rendimiento del modelo MNIST entrenado con distintos datos que se hace en la Tab. 4.5 no es especialmente esclarecedora. La exactitud no es una métrica demasiado fiable cuando se trabaja con bases de datos desequilibradas, como se ha mencionado al definirla. Se ha empleado con el resto de modelos puesto que todos se entrenaban con los mismos datos y era válida como indicador de mejoría. Sin embargo, la Tab. 4.5 compara modelos entrenados en bases de datos diferentes.

Se puede observar un claro balance entre la precisión y la sensibilidad. El modelo entrenado en la base de datos original es más preciso, pero el que se ha entrenado con la base de datos aumentada es más sensible, cosa que en esta situación es deseable. El valor F1 considera ambas cantidades, y se puede observar una mejora global en el segundo modelo, por muy pequeña que sea.

Cabe destacar que esta evaluación se ha realizado con el conjunto de prueba de la base de datos original, para que los resultados obtenidos sean comparables. No obstante, este conjunto de datos está también desequilibrado, por lo que el modelo original, especializado en predecir las clases más frecuentes, obtendrá mejores resultados con mayor facilidad. Sería interesante construir un conjunto de datos de prueba equilibrado, con el mismo número de muestras por clase y realizar la misma comparación. Seguramente en ese caso el modelo entrenado en la base de datos equilibrada obtendría una puntuación mayor.

Además de este análisis, otro trabajo futuro que podría hacerse es estudiar cómo generalizar los sistemas construidos para que sean capaces de reconocer no solo símbolos individuales, sino fórmulas matemáticas enteras. Esta tarea resulta más compleja, puesto que no solo es necesario identificar los símbolos, sino que también se debe enseñar al

modelo a comprender la relación entre símbolos consecutivos: los significados de x^2, $x \cdot 2$ y x_2 son distintos, según la posición del segundo símbolo.

Para ello, sería necesario utilizar *procesamiento del lenguaje natural* además de la visión por ordenador [22, 23]. Hoy en día existen productos comerciales que son capaces de llevar a cabo la tarea que se ha descrito, como por ejemplo, Mathpix [24].

En este trabajo se ha dado únicamente un primer paso en el estudio de este campo, que ha servido para familiarizarse con los conceptos básicos e interiorizar la importancia de varios aspectos, como el de tener una base de datos adecuada para la tarea que se quiere realizar o el de dividir los datos en distintos conjuntos según su finalidad.

Hoy en día el aprendizaje automático y la inteligencia artificial están en boca de todos. Poseer conocimientos sobre este tema, aunque sean básicos, puede ayudar a tener una perspectiva más realista del potencial de esta tecnología.

REFERENCIAS

[1] D. Knuth, *The TeXbook* (Addison-Wesley, 1984).

[2] L. Lamport, *The LATEX Document Preparation System* (Addison-Wesley, 1986).

[3] S. Pakin, *The Comprehensive LATEX Symbol List*, (Consultado el 09/05/2024), (ene. de 2024) https://www.ctan.org/tex-archive/info/symbols/comprehensive/.

[4] D. Kirsch, *Detexify: LaTeX handwritten symbol recognition*, (Consultado el 09/05/2024), https://detexify.kirelabs.org/classify.html.

[5] D. Kirsch, *Detexify data*, (Consultado el 09/05/2024), https://github.com/kirel/detexify-data.

[6] M. Thoma, "The HASYv2 dataset", arXiv (2017).

[7] M. Thoma, "HASYv2 - Handwritten Symbol database", Zenodo (2017).

[8] A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*, 2.ª ed. (O'Reilly Media, sep. de 2019).

[9] MIT Open Course Ware, *Artificial Intelligence*, (Consultado el 14/12/2023), (2010) https://ocw.mit.edu/courses/6-034-artificial-intelligence-fall-2010/.

[10] W. S. McCulloch y W. Pitts, "A logical calculus of the ideas immanent in nervous activity", The bulletin of mathematical biophysics **5**, 115-133 (1943).

[11] F. Rosenblatt, "The perceptron: a probabilistic model for information storage and organization in the brain", Psychological review **65**, 386 (1958).

[12] D. E. Rumelhart, G. E. Hinton y R. J. Williams, "Learning Internal Representations by Error Propagation", en *Parallel Distributed Processing*, vol. 1 (The MIT Press, jul. de 1986) cap. 8, págs. 319-362.

[13] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position", Biological cybernetics **36**, 193-202 (1980).

[14] Y. LeCun et al., "Gradient-based learning applied to document recognition", Proceedings of the IEEE **86**, 2278-2324 (1998).

[15] A. Gholamy, V. Kreinovich y O. Kosheleva, "Why 70/30 or 80/20 relation between training and testing sets: A pedagogical explanation", Int. J. Intell. Technol. Appl. Stat **11**, 105-111 (2018).

[16] A. C. Müller y S. Guido, *Introduction to Machine Learning with Python*, 1.ª ed. (O'Reilly Media, 2017).

[17] J. Brownlee, *Imbalanced classification with Python: better metrics, balance skewed classes, cost-sensitive learning* (Machine Learning Mastery, 2020).

[18] N. V. Chawla et al., "SMOTE: synthetic minority over-sampling technique", Journal of artificial intelligence research **16**, 321-357 (2002).

[19] D. Dablain, B. Krawczyk y N. V. Chawla, "DeepSMOTE: Fusing deep learning and SMOTE for imbalanced data", IEEE Transactions on Neural Networks and Learning Systems (2022).

[20] R. Wiersma, *ipycanvas-drawing*, ver. 0.0.5, (2023) https://pypi.org/project/ipycanvas-drawing/.

[21] Voila Development Team, *Voila*, ver. 0.5.7, (2024) https://voila.readthedocs.io/.

[22] G. Genthial y R. Sauvestre, "Image to Latex", CS231n: Deep Learning for Computer Vision (2016).

[23] Y. Deng, A. Kanervisto y A. M. Rush, "What you get is what you see: A visual markup decompiler", arXiv (2016).

[24] Mathpix Inc., *Mathpix*, ver. 03.00.0114, (2024) https://mathpix.com/.

[25] M. Thoma et al., "On-line Recognition of Handwritten Mathematical Symbols", arXiv (2015).

[26] B. Milde, *Shapecatcher: Unicode Character Recognition*, (Consultado el 09/05/2024), https://shapecatcher.com/.