Sección de Matemáticas
Universidad de La Laguna

Daniel Rodríguez Espinosa

# *Sobre el uso de técnicas de Machine Learning para clasificar imágenes astronómicas*

## On the use of Machine Learning techniques to discriminate astronomical images

Trabajo Fin de Grado
Grado en Matemáticas
La Laguna, Septiembre de 2022

Dirigido por
*Jose A. Acosta Pulido*

*Jose A. Acosta Pulido*
*Departamento de Astrofísica*
*Universidad de La Laguna*
*38200 La Laguna, Tenerife*

# Resumen · Abstract


### *Resumen*

*En la presente memoria se pretende investigar técnicas de reconocimiento de imagen mediante el uso de redes neuronales para separar imágenes mal tomadas, así como desenfoques. Inicialmente introducimos las nociones básicas de qué es una imagen astronómica, cómo es la forma de los objetos estelares que la conforman, el perfil de intensidad, describiendo las herramientas a utilizar para visualizar y clasificar los objetos de forma automática mediante el uso del deep learning. A continuación se pasa a estudiar sobre las redes neuronales, qué son y de qué se componen, desarrollando los algoritmos del descenso del gradiente y backpropagation mediante los cuales estas pueden aprender de forma jerarquizada. Posterior a algunos ejemplos ilustrativos, se introducen las redes neuronales convolucionales, las cuales se especializan en el tratamiento de imágenes, viendo cómo la red puede reconocer patrones de forma semejante a como lo realiza el ojo humano, mediante el proceso de convolución y filtros de imágenes. Finalmente, se ilustran los conceptos previos mostrando los resultados de implementar una red neuronal convolucional que clasifique imágenes astronómicas, discutiendo cómo la dificultad del aprendizaje automático recae en la buena elección del modelo, concluyendo con las predicciones del mismo.*



**Palabras clave:** *Imágenes astronómicas – Deep Learning – Redes Convolucionales – Python – Gradient Descent – Backpropagation – Convolución de matrices – Filtros*

## *Abstract*

*In the present manuscript it is intended to investigate image recognition techniques through the use of neural networks to separate badly taken images, as well as blurs. Initially we introduce the basic notions of what an astronomical image is, what the shape of the stellar objects that make it up is like, the intensity profile, describing the tools to be used to visualize and automatically discriminate objects using deep learning. Next, we go on to study about neural networks, what they are and what they are made of, developing gradient descent and backpropagation algorithms through which they can learn in a hierarchical way. After some illustrative examples, convolutional neural networks are introduced, which specialize in image processing, seeing how the network can recognize patterns in a similar way to how the human eye does it, through the convolution process and image filters. Finally, the previous concepts are illustrated, showing the results of implementing a convolutional neural network that discriminates astronomical images, discussing how the difficulty of machine learning lies in the good choice of the model, concluding with its predictions.*

**Keywords:** *Astronomical Images – Deep Learning – Convolutional Neural Networks – Python – Gradient Descent – Backpropagation – Matrix Convolution – Filters*

# Contenido

# 1

# Motivación Astronómica

En la actualidad hay muchos proyectos en Astrofísica que se basan en la adquisición de una gran cantidad de imágenes, en muchos casos obtenidas con telescopios robóticos; el tratamiento de esas imágenes y la extracción de información a partir de las mismas se hace de forma automática (fotometría de aperturas). Por medio de esta obtenemos el flujo observado de los objetos estelares que tendremos a disposición para esta memoria. Algunas imágenes muestran problemas debido a distintas razones; por ejemplo, que el seguimiento del telescopio no ha sido estable, el foco del instrumento no es correcto, la nitidez/contraste[1] ha sido deficiente, etcétera. Por tanto, se hace necesario introducir un sistema de clasificación de imágenes que permita detectar aquellas que potencialmente pueden causar problemas.

*Nota.* La distribución radial de energía para una fuente puntual se puede aproximar por una función gaussiana, cuya anchura a media altura ($FWHM$) se denomina "*seeing*" y depende de la noche y en general del momento de la observación.

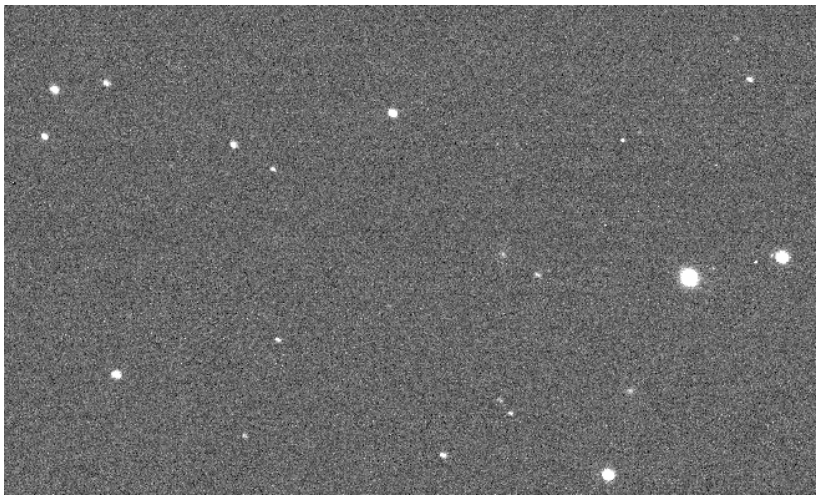


Figura 1.1: Imagen astronómica en el entorno *DS9*, ampliada

[1] *Seeing*

Vamos a investigar entonces técnicas de reconocimiento de imagen mediante el uso de redes neuronales para separar imágenes mal tomadas (como pueden ser por ejemplo los desenfoques). En cuanto a la clasificación, las imágenes se organizarán principalmente por la forma que presentan las estrellas (*fwhm* de las *gaussianas*). Se requerirá del uso del lenguaje de programación *Python* (en especial las librerías *keras* y *tensorflow*: la primera una biblioteca puramente enfocada a redes neuronales y la segunda, a aprendizaje automático en general). Además serán de ayuda: *numpy*, *matplotlib* y *h5py* para la lectura de datos.

Los datos de los que disponemos son los parámetros relacionados con las imágenes astronómicas de las cuales queremos detectar potenciales problemas. Estas imágenes (*.fits*) se encuentran en la escala de grises «repletas» de estrellas, las cuales se presentan, idealmente, de forma circular; con telescopios pequeños, el fondo de cielo es debido principalmente a la contaminación lumínica (iluminación civil y, sobre todo, a la *luz* de la Luna), produciéndose «ruido» sobre el flujo de las estrellas.

El entorno inicial de trabajo será *DS9* (visualizador de imágenes), donde podremos comenzar a analizar las imágenes con las que trabajaremos. Disponemos de una lista de imágenes tomadas con distintos telescopios con cierta frecuencia y se requiere de una herramienta automática para clasificar las imágenes, determinar la utilidad de cada una de estas. El trabajo se realizará con un conjunto de 80 imágenes[2], las cuales analizaremos y podemos clasificar en diferentes formas:

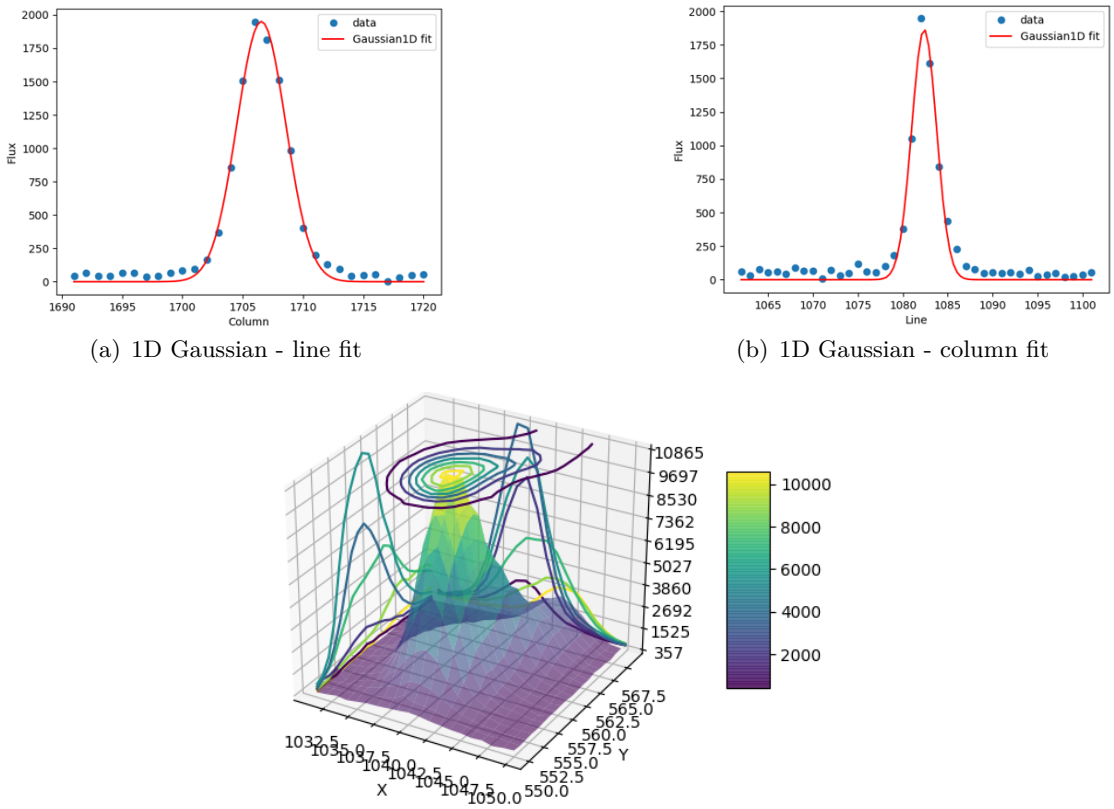
(a) *Circulares* (b) *Cometas* (c) *Donuts*

Figura 1.2: Clasificación de las imágenes astronómicas

siendo las que presentan deformidades 1.2(b), 1.2(c) las que queremos detectar como defectuosas.

Desde la terminal de *Ubuntu* podemos hacer nuestros primeros análisis de las imágenes mediante *DS9*, y un módulo que permite analizar el perfil de intensidad de las estrellas, *imexamine*, en *python*. Para comenzar con el análisis,

[2] Clasificaremos objetos de tipo estelar que se muestran como compactos; no se incluyen objetos extensos como galaxias o nebulosas.




(a) 1D Gaussian - line fit

(b) 1D Gaussian - column fit

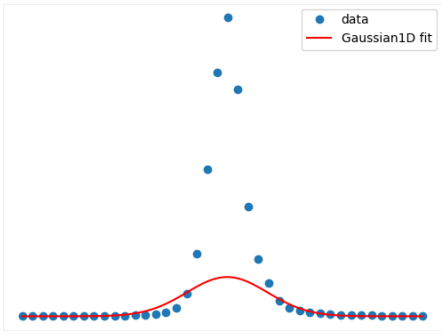
(c) 2D Gaussian - fit

Figura 1.3: Anchura y altura (radial) de una estrella en píxeles

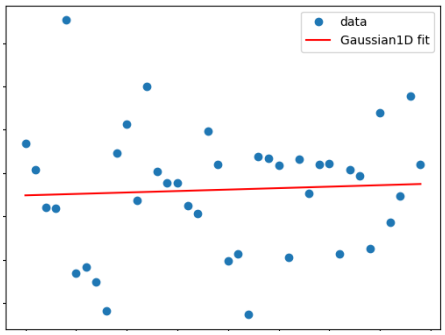
en primer lugar hacemos uso de este módulo *imexamine* para calcular el *fwhm* tanto de la anchura radial como de la altura de las estrellas de las imágenes, apuntando dicho valor en una tabla *Excel* con vistas a tener una clasificación de las imágenes con sus respectivas etiquetas, 1.2. En esta figura vemos como resultado las gaussianas asociadas al tamaño de la mancha representativa de cada estrella, cómo se observa el perfil de intensidad cuando se toman las imágenes desde tierra; cuando aparecen estrellas del tipo 1.2(b)-1.2(c), se salen de esta forma, que es lo que nos interesa detectar. En este caso, por ejemplo, hemos medido una estrella localizada en la Figura 1.1.[3] Podemos apreciar que en esta las estrellas se presentan circulares y es por tanto de esperar que las amplitudes de las gaussianas (altura y anchura a lo largo de las direcciones vertical y horizontal) sean similares (4.33 y 3.15, 2.51 y 1.82, respectivamente).

*Nota.* Al determinar la base de la gaussiana, si se seleccionan estrellas muy brillantes, el perfil se extiende mucho, saturándose la parte central de este; por tanto, no se puede medir.[4]

[3] Como nota, las imágenes *.fits* son de 3054 × 2042 píxeles (ancho × alto).
[4] Al medir las estrellas, se hace en torno al centroide de la mancha representativa.




(a) Saturación en la parte central del perfil de intensidad.




(b) Medición de estrella tipo *Cometa*, fuera del centroide.

Figura 1.4: Errores en el ajuste de los datos.

Midiendo el ancho de la gaussiana a mitad de altura, clasificaremos las estrellas de cada una de la lista de imágenes en 2 grupos principales: buena y mala visibilidad (*good seeing* y *bad seeing*, respectivamente); el límite escogido entre estos grupos viene dado por 2.0 *arcsec*(") como *fwhm*, siendo las imágenes por debajo de este las clasificadas como *good seeing*.[5] En el caso de imágenes en las que se aprecian manchas circulares, lo que haremos es medir $3-4$ estrellas y sacar el ancho vertical y horizontal y apuntar los valores para clasificarlas como buena o mala visibilidad. Nos va a interesar tener una clasificación global para toda la imagen, una etiqueta para cada imagen basada en las propiedades promedio de las estrellas.[6]

Una vez hecho esto, requerimos del módulo *DAOStarFinder* disponible en la librería photutils de astropy; gracias a este podemos (mediante un programa realizado en *python*, [1]) detectar y localizar las estrellas de las imágenes automáticamente, para poder recortarlas y guardar la información que contiene en estas, en archivos *hdf5* (*.h5*).[7] De este programa obtenemos una lista con las posiciones de todas las estrellas y algunos parámetros importantes que caracterizan a estos objetos estelares: forma del perfil de intensidad, el flujo, la magnitud de la estrella, la redondez, etcétera.

```
['id', 'xcentroid', 'ycentroid', 'sharpness', 'roundness1',
    'roundness2', 'npix', 'sky', 'peak', 'flux', 'mag']
```

Los parámetros más relevantes a tener en cuenta en la clasificación son: *FWHM* Full Width Half Maximum (Anchura máxima a mitad de altura) y *sigma* $\sigma$, este último correspondiente a la desviación típica en los valores medidos sobre el cielo.[8] También es relevante el parámetro *threshold*: especifica cuántas veces sobre el va-

[5] *Good seeing* se considera en general para valores de *fwhm* inferiores a $\sim 1.2$ *arcsec*; debido a que todas nuestras imágenes presentaban este valor superior a 1.5, se eligió a priori esta clasificación.

[6] Si medimos $3-4$ estrellas, tendremos el promedio de la imagen en cuanto a forma; es decir, el *fwhm* no variará en promedio.

[7] Usamos *hdf5* pues permite organizar los datos dentro del archivo de muchas formas estructuradas diferentes; además, este formato también permite «incrustar» metadatos.

[8] ...

lor sigma debe estar la medida de un pixel para que sea un candidato a estrella; determinará el valor absoluto de la imagen por encima del cual se seleccionan las fuentes (*DAOStarFinder*).

*Nota.* El valor proporcionado del *fwhm* (*imexamine*) corresponde a píxeles de la imagen; este se convierte a " segundos de arco multiplicando por 0.58.

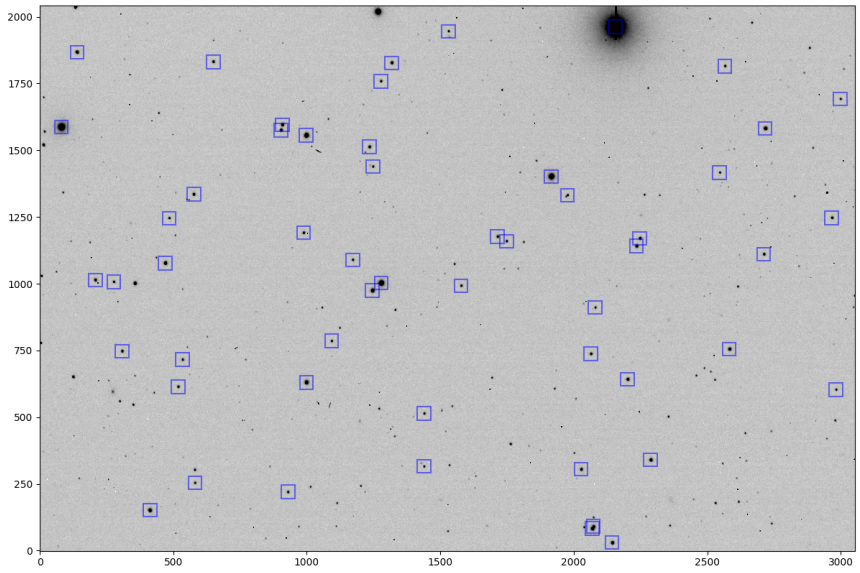
Guardaremos entonces las imágenes («data» y «meta-data») en formato *hdf5*, para luego poder pasárselo a la red. Estas imágenes, como comentamos, han sido clasificadas asignándoles un tipo según las 3 categorías mencionadas, Figura 1.2. Disponemos de un total de 80 imágenes, las cuales se repartirán en 3 listas disjuntas (listas de entrenamiento, validación y evaluación).[9] De cada imagen de nuestras listas se «extraen» automáticamente las estrellas estableciendo un tamaño de caja en la que recortar las estrellas que se detecten (centradas); se extraerá tanto el valor de los píxeles de estas cajas, *data*, junto con la etiqueta asignada, *meta-data*, (*Circular*, *Cometa* a las más alargadas y *Donut* a las que se presentan desenfocadas).

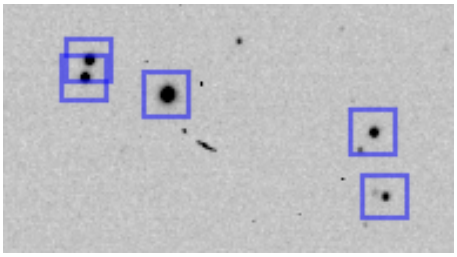
Vemos un ejemplo de la salida que presenta el programa ante una de las imágenes, seleccionando y recortando las estrellas, en la Figura 1.5.

Se observa que hay intersección entre las cajas, dando lugar a diferentes fuentes en un mismo conjunto de datos; ya que nos interesa pasarle a la red patrones no contaminados, se implementa un algoritmo para dejar de tener en cuenta las estrellas que se solapen, además de aquellas que se encuentren cercanas a los bordes.[10]

*Nota.* En cuanto a las estrellas tipo *Donut*, a la hora de pasarle este tipo de imágenes al módulo *DAOStarFinder* para detectar las fuentes, este no funciona tal como se espera, pues detecta varias fuentes en torno a la misma estrella. Como se disponen de pocas muestras, se han recortado manualmente proporcionándole al programa los centroides de estas estrellas.[11]

[9] Este proceso se explicará posteriormente en la memoria, pues compete al tema de las redes neuronales.
[10] En el programa hacemos *máscaras* (o filtros) para eliminar estos *problemas*.
[11] El tamaño de las cajas será de $60 \times 60$ píxeles.



(b) Ampliación de una región de la imagen superior.

Figura 1.5: Ejemplo de gráfica donde aparecen enmarcadas las fuentes seleccionadas.

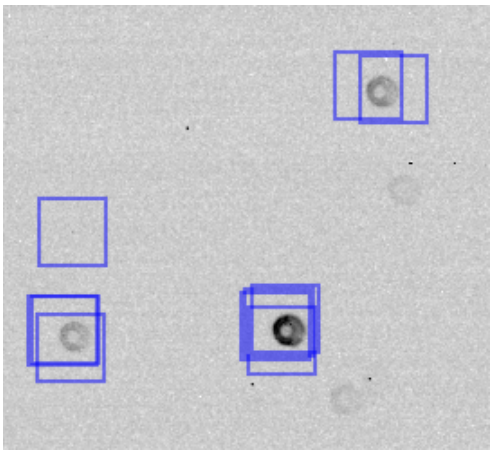


Figura 1.6: *DAOStarFinder* detectando repetidas fuentes, tipo *Donut*.

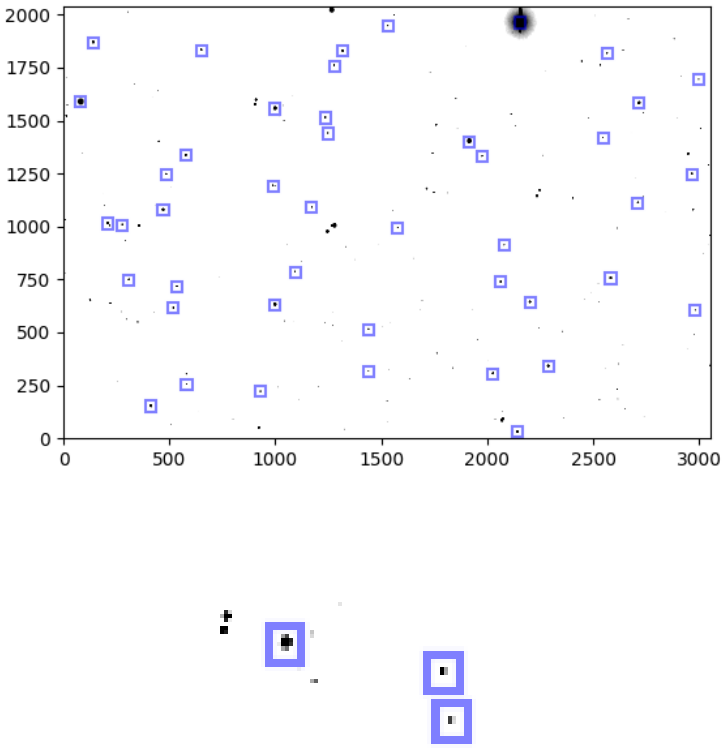



Figura 1.7: Imagen 1.5 sin solapamiento de fuentes, eliminando el fondo de cielo.

A la hora de tratar con las imágenes, le restamos la mediana de la imagen (*img_nobck* (no background)), es decir, el fondo, ya que así se detectan mejor las fuentes. Si tenemos mucha iluminación (como puede ser una noche con luna), tendremos un fondo notable, siendo así más complicado detectar las estrellas.[12] Estas imágenes sin fondo son las que recortaremos y almacenaremos para la posterior clasificación de la red, cuyos valores están comprendidos entre 0 y 1 para que la red pueda aprender eficientemente.[13]

Por tanto, tenemos una lista de entrenamiento, otra de validación y otra de evaluación. Hacemos el proceso de «extracción de fuentes» y guardamos esta información en archivos *hdf5*. En promedio, de cada una de las 80 imágenes se obtendrán entre

[12] Se calcula la mediana porque la mayoría de los valores de la imagen corresponde al fondo; a la hora de trabajar con las imágenes, se restará para trabajar con menos «ruido».

[13] Los valores negativos de las imágenes se han «hecho» 0 y se ha dividido cada recorte de imagen entre el máximo de cada uno.

$50 - 100$ recortes de estrellas. Como comentaremos en los siguientes capítulos, en general se requieren decenas de miles de imágenes para que la red obtenga buenos resultados (como Ejemplo, 2.4.2, con 50.000 imágenes para entrenar a la red). Esto se debe a que el *deep learning* requiere de grandes cantidades de datos para que las redes puedan entregar buenos resultados.

Es por ello que incluso para mejorar la estadística (pues para la red final se usan en torno a 5000 repartidas entre las 3 listas) se han rotado imágenes para añadirlas a la cantidad total; además, hay imágenes con estrellas muy pequeñas, resultando en la casi nula detección de estas. Mediante un condicional en *python*, se disminuye el valor de *threshold* con vistas a detectar estrellas con menor intensidad, además de reducir ligeramente también el *fwhm* máximo.[14]

A lo largo de la memoria veremos que el camino de construir la estructura de las redes neuronales es relativamente «simple»; sin embargo, la complicación resulta en establecer los parámetros de estas así como las técnicas que se empleen para que el resultado final sea óptimo. Vamos a introducir entonces capítulos relacionados sobre las redes, viendo cómo podemos ir poco a poco estructurando la red que clasifique imágenes astronómicas, con diversos ejemplos e ilustraciones, mejorando poco a poco los modelos a implementar.

Todos los programas *python* que se han realizado a lo largo de la memoria se encuentran en un repositorio *GitHub*, [1], donde poder ver los códigos comentados. Las gráficas o figuras de los ejemplos ilustrativos de la memoria se han realizado gracias tanto a la herramienta *Tikz* como a *python*.

[14] Si esta cantidad no supera la detección de al menos 3 estrellas, se aplica el condicional.

# 2

# Introducción a las Redes Neuronales

**Inteligencia artificial**

Subdisciplina del campo de la informática, que busca la creación de máquinas que puedan «imitar» comportamientos inteligentes (*IA*).

**Aprendizaje automático**

(*Machine Learning*, ML) Rama del campo de la inteligencia artificial, que busca cómo dotar a las máquinas de capacidad de aprendizaje, entendido este como la generalización del conocimiento a partir de un conjunto de experiencias. Tenemos varios tipos de aprendizaje: supervisado, no supervisado y reforzado.

Dentro del *ML*, existen diferentes técnicas que sirven para cubrir diferentes tipos de aplicaciones; por ejemplo, técnicas como los árboles de decisión, modelos de regresión, modelos de clasificación, técnicas de clusterización, etc. La técnica más destacable en la época actual: las redes neuronales. Lo interesante de estas es que son capaces de aprender de forma jerarquizada; es decir, la información se aprende por niveles (*layers* o capas), donde en los primeros se aprenden conceptos muy concretos (como por ejemplo qué es una rueda) y en las capas posteriores se usa la información previa para aprender conceptos más abstractos (más complejos; por ejemplo, qué es un coche). La tendencia es que estos algoritmos adhieran más y más capas, convirtiéndose cada vez en algoritmos más complejos. Este incremento en el número de capas y en la complejidad, es lo que hace que estos algoritmos sean conocidos como algoritmos de *Deep learning*, aprendizaje profundo.

Durante la memoria, trabajaremos con **Aprendizaje supervisado**.

**Aprendizaje Supervisado**

Se basa en descubrir la relación entre unas variables de entradas y otras de salida; el aprendizaje surge de enseñarle a estos algoritmos cuál es el resultado que se quiere obtener para un determinado valor. Tras mostrarle muchos ejemplos, el algoritmo será capaz de dar un resultado correcto incluso ante valores que no ha visto antes, si se dan las condiciones (Sección 2.3.2).

## 2.1. Qué son las Redes Neuronales

Como suele ocurrir con la mayoría de comportamientos y estructuras avanzadas, la complejidad de estos sistemas emerge de la interacción de muchas partes más simples trabajando conjuntamente. En el caso de una red neuronal, cada una de estas partes se le denomina: **neurona**, la unidad básica de procesamiento dentro de la red. Similar a una neurona biológica, tiene conexiones de entrada a partir de las cuales reciben estímulos externos, los valores de entrada $x_i$; esta realizará un cálculo interno y generará un valor[1].

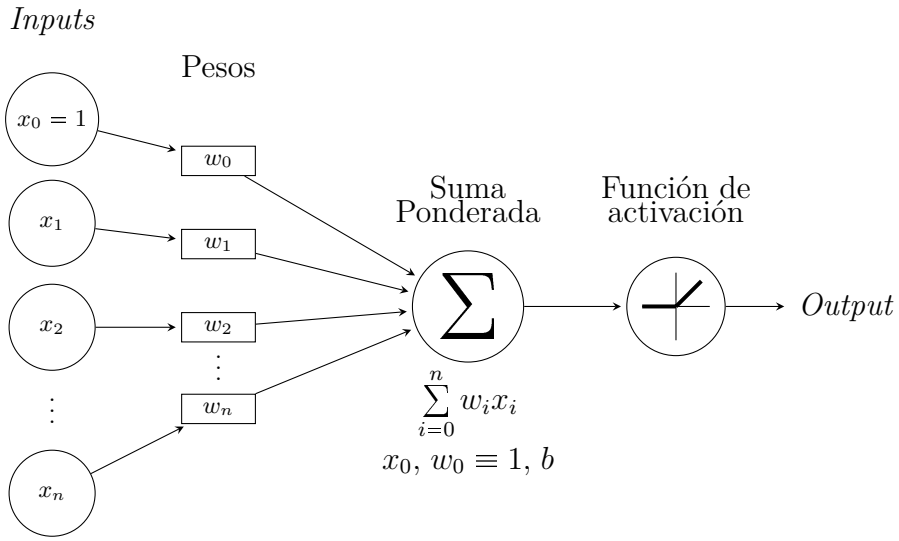



Figura 2.1: Perceptrón, con función de activación (ver Sección 2.1.1)

La neurona utiliza todos sus valores de entrada $(x_i)$ para realizar una suma ponderada, determinada por los pesos asignados a cada una de las conexiones de entrada, $w_j$; es decir, cada conexión que llega a nuestra neurona tendrá asociada un valor que servirá para definir con qué intensidad afecta cada variable a la neurona. Estos

[1] El término neurona hace referencia a lo que conocemos comúnmente como: función matemática.

pesos son los parámetros de nuestro modelo y son los valores que podremos ajustar para que nuestra red neuronal pueda «aprender»[2].

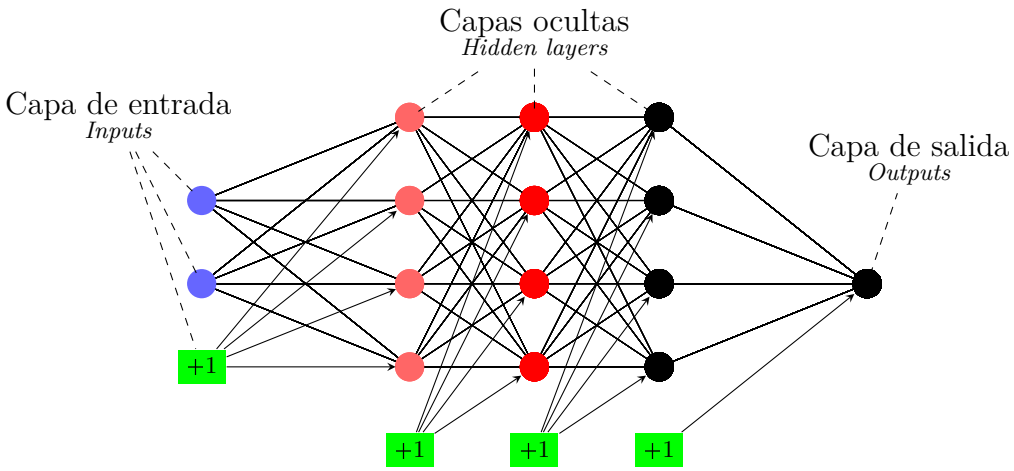



Figura 2.2: Red neuronal multi-capa.

$b$ bias: **sesgo**, se representa como otra conexión a la neurona cuya variable siempre está asignada a 1, y que podemos manipular modificando el valor del peso asociado $b$. Vemos que la forma de operar de la neurona se asemeja al modelo de regresión lineal; es decir, solo podemos modelar problemas lineales[3]. Debido al limitante de trabajar con una única neurona, se necesita combinar varias de estas unidades básicas para conseguir modelos más complejos; la combinación de estas y la cantidad de capas conforman la **red neuronal**, **ANN** Artificial Neural Networks, colocándose cada unidad de forma secuencial y recibiendo cada una de estas la información procesada por las neurona de la capa anterior, consiguiendo así que la red pueda «aprender» conocimiento jerarquizado; entre más capas añadimos, más complejo puede ser el conocimiento que elaboremos deep learning.

Al construir la red, tenemos una cierta cantidad de neuronas que operan linealmente, resultando así la red en un modelo lineal[4], «colapsando» la red a una unidad básica (2.1).La diferencia vendrá con las funciones de activación, que modificarán la linealidad en cada una de las neuronas de la red, dando lugar así a una variación en el resultado de salida. Ahora la suma ponderada pasará por nuestra función de activación para distorsionar el valor de salida añadiéndole deformaciones no lineales, pudiendo así encadenar de forma efectiva la computación de varias neuronas (o perceptrones).

[2] En una pequeña sección (2.3.2) más adelante debatiremos brevemente sobre el hecho de que las redes neuronales *aprenden* o *memorizan*.
[3] Sin tener en cuenta el término función de activación.
[4] La combinación de linealidad produce linealidad: las variables de salida definen una recta (regresión lineal simple), o un hiperplano (regresión lineal múltiple).

### 2.1.1. Funciones de activación

Las redes neuronales deben pensarse como *aproximadores* de funciones. Como hemos mencionado, cada neurona de la red recibe una suma ponderada de las entradas a esta, se le suma el sesgo y se traslada el resultado a las siguientes capas, haciendo esto para cada una de las neuronas; esta operación es lineal. Añadiendo por tanto las **funciones de activación**, esa suma ponderada será «distorsionada» a las capas posteriores. Así, una red con suficientes capas y neuronas (y entrenamiento[5]), podrá ajustarse a funciones mucho más complejas: «entender» imágenes por ejemplo[6].

Veamos a continuación las funciones de activación más conocidas y utilizadas.

- Función **escalón**: esta función nos proporciona la no linealidad que buscamos pero no podemos considerarla para el aprendizaje automático.

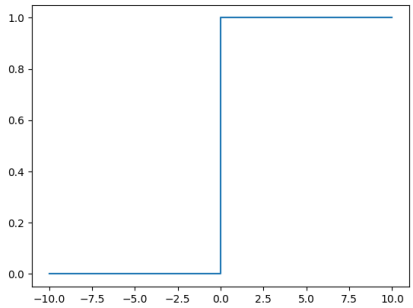



Figura 2.3: Función *escalón*

Como veremos, necesitamos de una función de coste (Sección 2.2.3) para analizar la precisión de las predicciones de la red; querremos ajustar los parámetros de la red para mejorar, los pesos y sesgos. Para ello, necesitaremos calcular en el proceso las derivadas de la función de activación que usemos en nuestra red, por lo que esta función no es opción, ya que tiene derivada nula[7].

- Función **sigmoide**: también llamada función logística, una función diferenciable; la distorsión que produce hace que los valores muy grandes se saturen en 1 y los pequeños en 0. Por tanto, podemos representar probabilidades; es altamente

[5] El aprendizaje y entrenamiento de la red vendrá dado por 2 algoritmos que veremos en las siguientes secciones del capítulo.

[6] Redes convolucionales en el Capítulo 3.

[7] Además, no es diferenciable en 0; sin embargo, esto no es relevante ya que en general obtener este valor durante el proceso de entrenamiento de la red es bastante improbable. También sucede esto con la función *ReLU*.

utilizada, por ejemplo, en clasificaciones binarias, asignando esta función a la capa de salida, asegurando así que la salida corresponderá a un valor en el rango [0, 1].

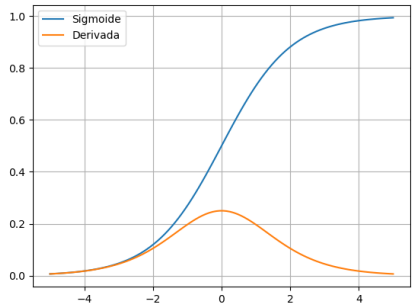



Figura 2.4: Función logística *sigmoide* $\frac{1}{1+e^{-\mathbf{x}}}$

No obstante, esta función tiene un inconveniente, lo que se conoce como *desvanecimiento de gradiente*. Esto lo podemos apreciar gráficamente, como al recibir valores muy grandes en valor absoluto, la derivada es prácticamente nula. Esto da lugar a que los ajustes de pesos y sesgos no varíen significativamente, por lo que la red aprenderá muy lentamente, o dejará de aprender en las primeras capas. Similar a esta tenemos la función *tanh*.

- Función **tanh** (tangente hiperbólica): similar a la sigmoide, salvo el rango de valores el cual varía desde $-1$ hasta 1. La derivada de esta es superior a la de la logística, dando por tanto mejores resultados de aprendizaje, además del beneficio de que esté centrada en el origen[8].

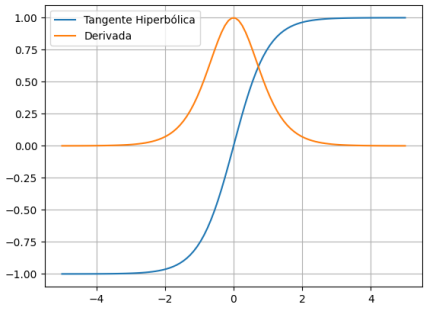



Figura 2.5: Función *Tanh* $\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$

[8] El *output* de la función sigmoide siempre es positiva; esto genera problemas en el algoritmo de *back-propagation* (Sección 2.3.1).

Conforme vamos actualizando los sesgos y pesos capa por capa, usar esta función limita las actualizaciones que se hacen en cada capa que sean positivas o negativas, limitando la velocidad a la que puede aprender la red. No obstante, la *tanh* también presenta el problema del desvanecimiento de gradiente, además del alto coste computacional debido a la exponenciación.

- Función **ReLU** (*Rectified Lineal Unit*): función lineal $max(0,\, x)$ cuando es positiva y constante a cero cuando el valor de entrada es negativo.

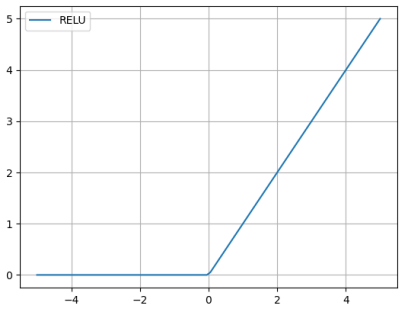



Figura 2.6: Función $ReLU$

Tiene un coste computacional muy bajo y es utilizada principalmente en redes neuronales convolucionales (Capítulo 3), obteniéndose resultados de menor error que las funciones anteriores inspiradas por la probabilidad. Esta no está acotada, generando un gradiente creciente constante, dando un aprendizaje mucho más rápido, pudiendo entrenar de manera eficiente redes cada vez más profundas.

También presenta algunos inconvenientes: principalmente, debido a que $ReLU$ regresa 0 para todos los valores negativos, puede generar neuronas muertas; es decir, durante el entrenamiento comienzan algunas neuronas a dar valores nulos, entorpeciendo el aprendizaje. Otras opciones para mejorar a $ReLU$ son las siguientes:

$$\begin{cases} ParametricReLU \mapsto max(\alpha x,\, x),\ \alpha \in \mathbb{R} \\ Softplus \mapsto ln(1 + e^x) \end{cases}$$

$$\begin{cases} Mish \longmapsto x \tanh(softplus(x)) = x \tanh(ln(1 + e^x)) \\ Softmax \longmapsto \sigma(z) = \dfrac{e^{z_k}}{\sum_{j=1}^{M} e^{z_j}}, \quad j = 1,\ \cdots,\ M. \end{cases}$$

Para nuestra red clasificadora de imágenes astronómicas, probaremos tanto **Mish** (la cual mejora a $ReLU$ en algunas tareas) como $ReLU$ en las capas ocultas y **Softmax** para la capa de salida, pues esta última es ideal para redes de clasificación.

Esta última permite que la capa de salida nos entregue una serie de probabilidades, cuya suma total resulta en 1; se *exponencia* cada valor de salida de la red y se divide entre la suma de todas estos *outputs* ($\sum_{j=1}^{M} e^{z_j}$), resaltando así los valores más grandes, elevando por tanto la probabilidad. Como ejemplo, veamos la salida de una red que quiere clasificar si una imagen es un tipo de animal u otro,

cuyos *outputs* de la red son, respectivamente, 8, 0.4 y 1.5:

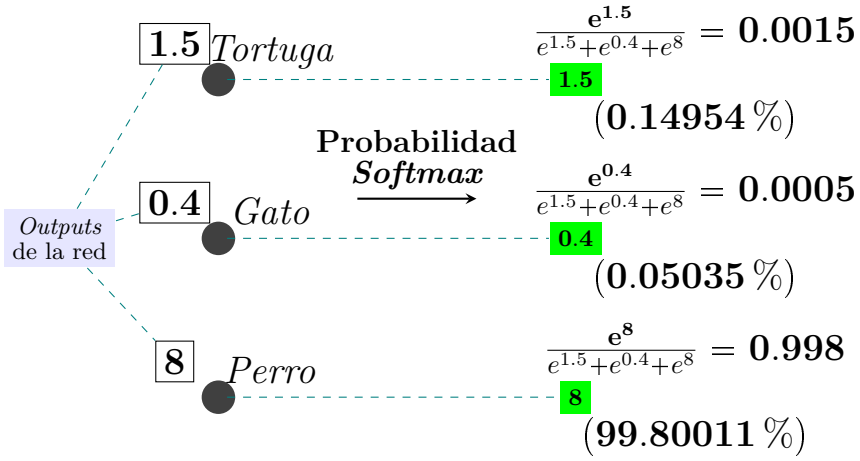



Figura 2.7: Ejemplo de clasificación de una única imagen de animales; la imagen de entrada es *Perro*.

*Nota.* Cuanto mayor sea el *output* de una cierta neurona de la capa de salida, significará que la red determina que dicha neurona es la más «influyente»; es decir, que en las redes de clasificación como en el ejemplo anterior, el hecho de que el *output* de la primera neurona (asociado a *Perro*) sea superior al resto, significa que tras el entrenamiento, la mayor probabilidad recae en la etiqueta de esa primera neurona, en contraposición con los que tienden a cero. *Softmax* potencia esto y hace que no hayan dudas en la clasificación, elevando así la precisión en el entrenamiento.

En cuanto a la Mish, a diferencia de las mencionadas en la sección anterior, esta función es no monótona, cuya derivada evita ligeramente el desvanecimiento de gradiente[9].

[9] Las funciones de activación no monótonas son tema de estudio debido a la gran eficacia al aplicarse a redes neuronales. Se piensa que la clave para asemejarse a las biológicas reside en las funciones de activación oscilatorias.

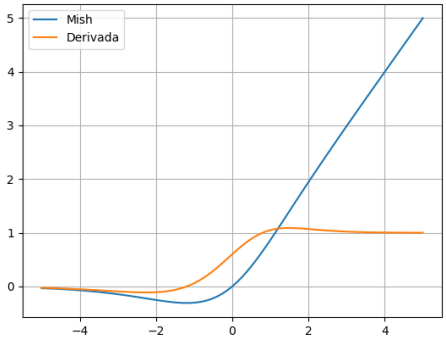



Figura 2.8: Función *Mish*

## 2.2. Algoritmo del Descenso del gradiente

Como vimos en la Sección 2.1 anterior, la suma ponderada de una neurona o perceptrón es $\sum_{i=0}^{n} w_i x_i$; si queremos generalizar esto a una red neuronal, debemos adherir una notación de *sub/súper*-índices para denotar la capa y las conexiones entre neuronas (nos basaremos en la notación de la bibliografía [2]). Los pesos: $w_{ij}^{(l)}$, $i$, $j$ son los subíndices asociados a los nodos en las capas $l+1$ y $l$, respectivamente. En cuanto al parámetro de sesgo $b$, dado que este se encuentra en cada capa conectado a las neuronas de la siguiente capa: $b_i^{(l)}$, análogamente. La suma ponderada vendrá caracterizada por $z_i^{(l)}$ (en este caso, $i$ denota al $i$-ésimo nodo de la capa $l$) y tendrá como resultado para la primera capa oculta de la red[10]:

$$z_i^{(2)} = w_{i1}^{(1)} x_1 + \ldots + w_{in_1}^{(1)} x_{n_1} + b_i^{(1)} = \sum_{j=1}^{n_1} w_{ij}^{(1)} x_j + b_i^{(1)},$$

donde $n_1$ es el número de nodos en la primera capa de la red e $i$ denota las neuronas de la capa 2; en cuanto al *output* de la capa, lo denotaremos por $h_i^{(l)}$, valor que viene caracterizado por aplicar a la suma ponderada del nodo $i$ de la capa $l$, la función de activación: $f(\cdot)$. La ecuación previa lleva por tanto a que:

$$h_i^{(2)} = f(w_{i1}^{(1)} x_1 + \ldots + w_{1,n_1}^{(1)} x_{n_1} + b_i^{(1)}) \longmapsto h_i^{(2)} = f(z_i^{(2)}).$$
$$\text{En general:} \quad h_i^{(l+1)} = f(z_i^{(l+1)}).$$

Desde una perspectiva computacional, resulta más conveniente escribir las ecuaciones de forma más compacta; eliminamos los subíndices $i$, $j$, «vectorizando» y generalizando:

[10] Se define así al igual que para la activación de la neurona $h$ para que haya concordancia con la notación que continua; además de que la primera capa de una red neuronal carece de suma ponderada.

$$\begin{cases} z^{(2)} = W^{(1)} \cdot x + b^{(1)} & \longmapsto h^{(2)} = f(z^{(2)}) \\ z^{(3)} = W^{(2)} \cdot h^{(2)} + b^{(2)} & \longmapsto h^{(3)} = f(z^{(3)}) \\ & \vdots \\ z^{(l+1)} = W^{(l)} \cdot h^{(l)} + b^{(l)} & \longmapsto h^{(l+1)} = f(z^{(l+1)}) \end{cases}$$

y así para todas las capas de la red, $n_l$, $L$[11]. $W$ denota la matriz de pesos; ahora el resto de elementos de la ecuación anterior se presentan en forma vectorial.[12]

$$\begin{aligned} z^{(l+1)} &= \begin{pmatrix} w_{11}^{(l)} & \cdots & w_{1n_l}^{(l)} \\ \vdots & \ddots & \vdots \\ w_{n_{l+1},1}^{(l)} & \cdots & w_{l+1,3}^{(l)} \end{pmatrix} \begin{pmatrix} h_1^{(1)} \\ \vdots \\ h_{n_l}^{(1)} \end{pmatrix} + \begin{pmatrix} b_1^{(l)} \\ \vdots \\ b_{n_{l+1}}^{(l)} \end{pmatrix} \\ &= \begin{pmatrix} w_{11}^{(l)} h_1^{(1)} + \cdots + w_{1n_l}^{(l)} h_{n_l}^{(1)} + b_1^{(1)} \\ \vdots \\ w_{n_{l+1},1}^{(l)} h_1^{(1)} + \cdots + w_{n_{l+1},n_l}^{(l)} h_{n_l}^{(1)} + b_{n_{l+1}}^{(l)} \end{pmatrix}_{(n_{l+1}) \times (n_l)} \end{aligned}$$

*Nota.* Si la función de activación puede aplicarse a cada elemento de dicha matriz (cada fila; es decir, aplicarse a cada neurona de la capa $(l+1)$), entonces podemos hacer todos nuestros cálculos usando matrices y vectores.

### 2.2.1. El modelo de regresión lineal

Previo a introducir el término de función de coste, repasemos el modelo de regresión lineal como punto de partida.

Como mencionamos en la Sección 2.1, cuando sumamos ponderadamente en las neuronas sin aplicar funciones de activación, estas son «equivalentes» al modelo de regresión lineal[13]. Este modelo se plantea matemáticamente buscando, mediante el método de mínimos cuadrados, una fórmula que determine exactamente cuál es el punto mínimo de la función de coste; es decir, el mínimo del *ECM* Error Cuadrático Medio:

$$\frac{1}{N} \|y_i - \hat{y}_i\|_2^2 = \frac{1}{N} \sum_i^N (y_i - \hat{y}_i)^2 = \frac{1}{N} \sum_i^N (\epsilon_i)^2$$

[11] $n_l$ denota los nodos de la capa $l$ y $L$ la cantidad de capas de la red.

[12] Gracias a la vectorización, los modelos de redes neuronales se entrenarán de manera mucho más eficiente, ya que la GPU está especializada en el procesamiento de matrices.

[13] Regresión lineal múltiple: las soluciones serán hiperplanos en espacios multidimensionales, donde cada una de estas dimensiones son características de la realidad que los datos $x_i$ representan.

$$\begin{cases} y_1 = w_0 + w_1 x_{11} + \ldots + w_n x_{1n} + \epsilon_1 \\ \vdots \\ y_N = w_0 + w_1 x_{N1} + \ldots + w_n x_{Nn} + \epsilon_N \end{cases}$$

$$X = \begin{pmatrix} 1 & x_{11} & \cdots & x_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ 1 & x_{N1} & \cdots & x_{Nn} \end{pmatrix} \quad Y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{pmatrix} \quad \epsilon = \begin{pmatrix} \epsilon_0 \\ \vdots \\ \epsilon_N \end{pmatrix} \quad W^T = \begin{pmatrix} w_0 \\ \vdots \\ w_n \end{pmatrix}$$

$$ECM: \ C(W) = \epsilon^T \epsilon = (Y - XW^T)^T (Y - XW^T)$$

$$\xRightarrow[C]{Derivando} \hat{W} = [(X^T X)^{-1} X^T Y]^T = Y^T X (X^T X)^{-1} \longmapsto \hat{Y} = X\hat{W}^T.$$

Sin embargo, el método de cuadrados ordinarios está limitado a este modelo y función de coste. Al trabajar con otros modelos u otras funciones de coste, en general, no podremos encontrar el mínimo de forma analítica; además, la solución $\hat{W}$ obtenida para la regresión es bastante ineficiente computacionalmente. Por tanto, requeriremos de otro método (algoritmo) que resuelva este problema: el descenso del gradiente, que irá ajustando los pesos hasta acercarse al mínimo de la función de coste.

### 2.2.2. Descenso del gradiente

Como sabemos, la configuración de los pesos que vinculan las capas en la red es lo que constituye el entrenamiento del sistema. En el aprendizaje supervisado, la idea es reducir el error entre la entrada y la salida deseada por medio de la función de coste; para el caso de la regresión lineal, pudimos encontrar un mínimo para esta analíticamente, ya que impusimos que fuera convexa[14]. Para generalizar a modelos más complejos, nos interesará estudiar por tanto las funciones no-convexas, en las que al buscar mínimos tengamos que resolver una gran cantidad de ecuaciones: $f'(\mathbf{x}) = 0$), sistema ineficiente de resolver.[15]

Vamos a ayudarnos de la derivada para, de forma iterativa, ir encontrando los mínimos locales de este tipo de funciones.

Vemos a continuación un diagrama bidimensional en el que los pesos se van actualizando de modo que se van acercando al mínimo de la función de error (o coste, representada mediante curvas de nivel)[16]. Y un ejemplo de implementación en *python* (*Gradient_Descent+.py* en [1]) de este algoritmo:

[14] La función de coste, *ECM*, es una función convexa con un mínimo (global).
[15] La diversidad de modelos y funciones de coste en el mundo del *ML* nos obliga a encontrar una solución para las funciones no-convexas.
[16] Cada anillo representa una curva de nivel con valores similares entre sí, como la diferencia de colores en la Figura 2.10(b).

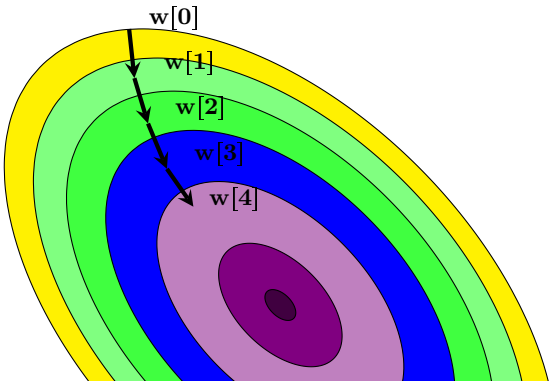
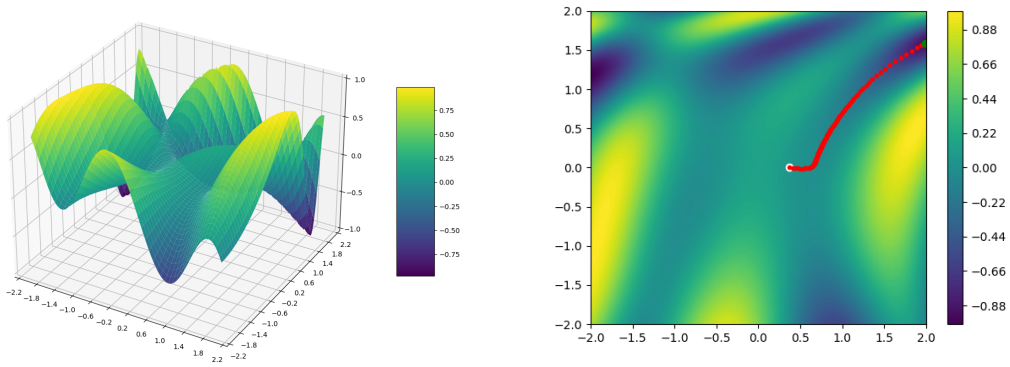



Figura 2.9: Descenso de gradiente bidimensional, herramienta *Tikz*.



(a) $z = sin\left(\frac{1}{2}x^2 - \frac{1}{4}y^2 + 3\right) \cdot cos(2x + 1 - e^y)$

(b) Ejemplo: iteraciones en busca del mínimo de la función de coste.

Figura 2.10: Ejemplo: función de coste no-convexa arbitraria y su proyección sobre el plano.

viendo como empezando en un punto aleatorio, el algoritmo optimiza hacia un mínimo de la función de coste (zona más azulada del gráfico); queremos conseguir llegar al valor mínimo de esta analizando el gradiente, paso a paso.

*Nota.* En nuestro caso, nuestros valores de entrada $\mathbf{x}$ tendrán una dimesionalidad $N \times N$ (imágenes $N \times N$). $\mathbf{y}$ en este caso podría ser un valor escalar único, ya sea un 1 o un 0 para designar si la imagen es de un tipo u otro, por ejemplo (clasificación binaria); en nuestro caso (Capítulo 4), será un vector de dimensión 3, pues clasificaremos las imágenes astronómicas en 3 tipos.

Para proceder con el algoritmo, primero se calcula el gradiente del error con respecto a $w$ en un primer punto "aleatorio"; es decir, la pendiente de la curva de error en ese punto. El gradiente ($\nabla$) proporciona información direccional: si es positivo con respecto a un aumento en $w$, un paso en esa dirección conducirá a un aumento en el error; si es negativo con respecto a un aumento en $w$, conducirá a una disminución en el error, buscando esta segunda opción. La magnitud del gradiente da una indicación de qué tan rápido está cambiando la función o curva

de error en ese punto; cuanto mayor sea esta, más rápido cambiará el error en ese punto con respecto a $\mathbf{w}$. Este método iterativo actualiza el valor de $\mathbf{w}$ tal que:

$$\mathbf{w} \longleftarrow \mathbf{w} - \alpha * \nabla_{\mathbf{w}} error.$$

$\alpha$ es el tamaño de paso learning rate; este determinará qué tan rápido converge la solución al mínimo error. Sin embargo, si es demasiado grande, el proceso no convergerá, pues las actualizaciones darán pasos grandes. Si es demasiado pequeño, la optimización será lenta; a medida que la solución se acerque al error mínimo, dará como resultado solo pequeñas mejoras en el error.

### 2.2.3. La función de coste en redes neuronales

Matemáticamente, la forma con la que observar la reducción del error se centra en minimizar la **función de coste**; variarán los pesos a lo largo del entrenamiento y la red *visualizará* la deficiente o correcta modificación de estos parámetros.

Como mencionamos en el ejemplo anterior, tendremos unos *inputs* a la red y una etiqueta *labels* asociada a dicha entrada; $\mathbf{x} \leftarrow$ vector $\mathbf{N}-$*dimensional* de una imagen, y como en Sección 2.4.2, $\mathbf{y} \leftarrow (1,\ 0,\ \ldots,\ 0)$ si es la letra $\mathbf{a}$, ..., $(0,\ 0,\ \ldots,\ 1)$ si es $\mathbf{z}$. Luego, si tenemos $m$ imágenes para entrenar a la red, tendremos $m$ pares de entrenamiento $(x^{(k)},\ y^{(k)})$, $k = 1,\ \ldots,\ m$ ($k$-ésima muestra de entrenamiento).[17] Asociada a un único par de entrenamiento, la función de coste será:

$$C = \frac{1}{2}\left\|y^{(k)} - \hat{y}(w,\ x^{(k)},\ b)\right\|_2^2 = \frac{1}{2}\left\|y^{(k)} - h^{(L)}(w,\ x,\ b)\right\|_2^2 = C(W,\ b,\ x^{(k)},\ y^{(k)})$$
$$\text{(Norma } \mathcal{L}^2\text{)}$$

$h^{(L)}$ denota la salida (*output*) de la red neuronal.[18] Sobre todos los pares de entrenamiento:

$$C(W,\ b) = \frac{1}{m}\sum_{k=0}^{m}\frac{1}{2}\left\|y^{(k)} - h^{(n_l)}(x^{(k)})\right\|_2^2 = \frac{1}{m}\sum_{k=0}^{m} C(W,\ b,\ x^{(k)},\ y^{(k)})$$

Vemos por tanto que la forma de actuar de $C$ es tomar la suma de los errores de todos los *inputs* entregados a la red. Una vez que se obtenga el valor total del coste, la red entrará en otra etapa de entrenamiento realizando el mismo proceso anterior, salvo por la diferencia de que los parámetros han sido ajustados para minimizar la función de coste. Este bucle se repetirá hasta que los parámetros ($W$, $b$) se hayan ajustado lo suficiente como para obtener buenos resultados.

[17] El orden de imágenes a entrenar suele ser $\sim 10^4$.
[18] El factor $\frac{1}{2}$ es una constante agregada para facilitar cuando se diferencia la función de coste.

La función de coste que utilizaremos para la red que clasificará nuestras imágenes astronómicas es la llamada *Cross-Entropy* *Categorical Cross-Entropy* definida por la ecuación

$$\left[\mathcal{C} = -\sum_{i=1}^{M} y_i \cdot log(\hat{y}_i)\right] = -\sum_{i=1}^{M} y_i \cdot log\left(\frac{e^{z_i}}{\sum_{j=1}^{M} e^{z_j}}\right) \xRightarrow[One-hot]{Realizando} -log\left(\frac{e^{z_k}}{\sum_{j=1}^{M} e^{z_j}}\right),$$

donde $M$ es el número de clases de clasificación, $y_i$ es el valor real; es decir, la etiqueta asignada a la clase $i$. $\hat{y}_i$ denota el valor del *output* predicho por la red, de la clase $i$-ésima. Se ha sustituido el valor predicho por el resultado de la función de activación *softmax* y realizado *one-hot* para las etiquetas de salida de la red: $y_k = 1 \mapsto y_i = 0$, $\forall i \neq k$ (ver el ejemplo ilustrativo en 2.4.2); por tanto, $z_k$ denota la salida de la red asociada a la etiqueta $y_k$.

Esta función está pensada para valores de $\hat{y}_i \in (0, 1]$; por ello usamos *softmax*, pues es la que mejor rendimiento proporciona junto a *Cross-Entropy*.[19] Siguiendo el Ejemplo 2.7, el resultado de la función de coste para una sola imagen será:

$$\mathcal{C} = -log(0.998) \sim 0.002 = C; \text{ la imagen corresponde a } Perro\ (y_k = 1).$$

La función de coste para un lote de $N$ imágenes será:

$$C = -\frac{1}{N}\sum_{k=1}^{N} log\left(\frac{e^{z_k}}{\sum_{j=1}^{M} e^{z_j}}\right).$$

**Descenso de gradiente en redes neuronales**

El descenso de gradiente para cada peso $w_{ij}^{l}$ y cada sesgo $b_i^{l}$ en la red neuronal, por lo visto anteriormente:

$$w_{ij}^{(l)} \longleftarrow w_{ij}^{(l)} - \alpha\frac{\partial}{\partial w_{ij}^{(l)}}C(w,\, b)$$
$$b_{i}^{(l)} \longleftarrow b_{i}^{(l)} - \alpha\frac{\partial}{\partial b_{i}^{(l)}}C(w,\, b)$$

*Nota.* Si $\mathbf{x}$ es un vector, entonces dicha derivada también será un vector, mostrando el gradiente en todas las dimensiones de $\mathbf{x}$. Con la notación prevista:

[19] Ambas funciones están pensadas para la clasificación multiclase. También podemos usar la función logística.

$$\begin{cases} W^{(l)} \longleftarrow W^{(l)} - \alpha \dfrac{\partial}{\partial W^{(l)}} C(W,\, b) \\ b^{(l)} \longleftarrow b^{(l)} - \alpha \dfrac{\partial}{\partial b^{(l)}} C(W,\, b) \end{cases} \tag{2.1}$$

**Optimizadores en redes neuronales**

La correcta configuración del *learning rate* es fundamental para poder hacer que el algoritmo del descenso del gradiente optimice eficientemente. Hay diferentes técnicas que sirven para ajustar este parámetro $\alpha$ de forma dinámica, al igual que mejoras en las fórmulas que hemos visto para que el movimiento iterativo del punto hacia el mínimo de la función de coste sea más eficiente:

Adagram, Adadelta, AdaGrad, SGD Stocastic Gradient Descent,
**Adam**, Newton, RMSprop, Ftrl, etcétera.

Utilizaremos **Adam** Adaptative moment estimation como optimizador al clasificar nuestras imágenes astronómicas debido a que, en general, es la que mejor resultados proporciona ante diferentes problemas de clasificación, además de ser computacionalmente eficiente, entre otras ventajas, [4].

## 2.3. *Backpropagation*

Vamos a ver a continuación cómo una red neuronal realiza su aprendizaje automático, mediante la unión del algoritmo del descenso del gradiente y de *backpropagation*. Para este último la idea es, a partir del valor de salida de la red, ir analizando el error que resulta de cada una de las neuronas de la capa anterior. Si una neurona comete un error *considerable*, es de esperar que el valor de salida de la red sea significativamente erróneo. Esta es una de las componentes claves que necesitamos tener en cuenta a la hora de optimizar a la red neuronal.

### 2.3.1. Las matemáticas detrás del *Backpropagation*

Cada neurona de la red se especializa en una tarea determinada; el resultado de salida (*output*) final de la red tendrá un cierto error comparándose con el valor estimado, controlando dicho error por medio de la función de coste que denotaremos $C$ como en secciones anteriores.

La idea de **backpropagation** es, debido al error que se produce en la salida de la red, analizar toda la cadena de responsabilidades de cada una de las neuronas de la red, para que, al entrenarla, para cada muestra de entrenamiento los parámetros de la red se vayan minimizando; es decir, que la función de coste $C$ se minimice. En el proceso, el entrenamiento de la red irá determinando dicha responsabilidad de error a cada neurona, lo que se usará para que esta modifique los parámetros

asociados a cada neurona.[20] Para cada neurona de una cierta capa, podemos asumir que esta es nuestra *nueva* última capa de salida con cierto error e ir capa por capa hacia atrás analizando y computando los errores (recursivamente).

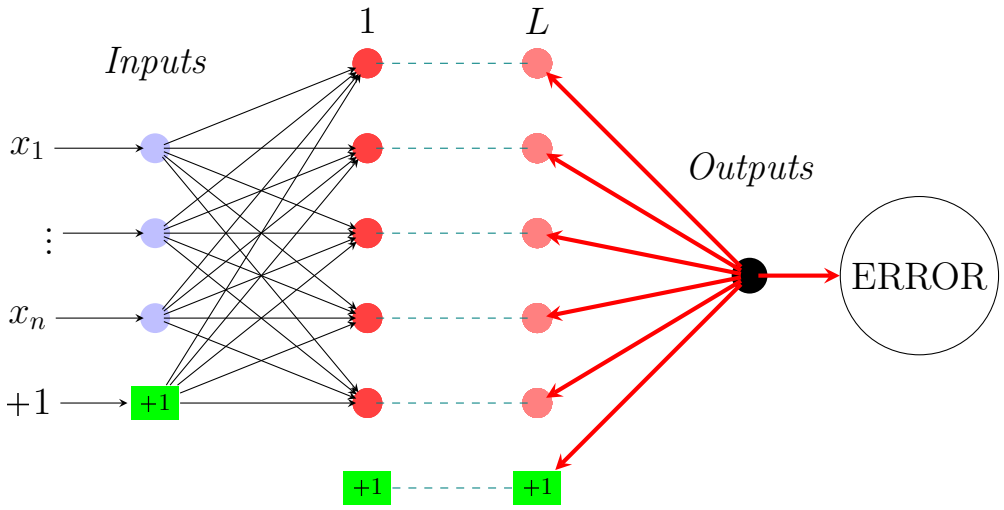



Figura 2.11: Algoritmo de *Backpropagation*.

Inicialmente, la red tiene configurados sus parámetros de forma aleatoria, lo que resulta en un valor de salida (*output*) aleatorio; la función de coste le asignará a la predicción un valor bastante elevado. Usaremos este error para entrenar a la red; estudiaremos cómo varía el coste (error) ante un cambio de los parámetros: $w,\ b \mapsto \frac{\partial C}{\partial w,b}$.

Suponiendo que nuestra red tiene $L$ capas, siguiendo la notación (Sección 2.2); $z^L = W^{(L-1)}X^L + b^{(L-1)}$, establecemos $z^{(L)}$ como el vector suma ponderada de las neuronas de la última capa, donde $W^{(L-1)}$, $b^{(L-1)}$ son las matrices de pesos y sesgos de la capa anterior asignadas a la *actual*. $X^{(L)}$ denota los *inputs*, que equivalen al resultado de aplicar las funciones de activación a las sumas ponderadas de la capa previa, $h^{(L-1)} = f(z^{(L-1)})$.

Mediante la regla de la cadena tenemos las siguientes expresiones:

$$\begin{cases} \dfrac{\partial C}{\partial W^{(L-1)}} = \dfrac{\partial C}{\partial h^{(L)}} \cdot \dfrac{\partial h^{(L)}}{\partial z^{(L)}} \cdot \dfrac{\partial z^{(L)}}{\partial W^{(L-1)}} \\ \dfrac{\partial C}{\partial b^{(L-1)}} = \dfrac{\partial C}{\partial h^{(L)}} \cdot \dfrac{\partial h^{(L)}}{\partial z^{(L)}} \cdot \dfrac{\partial z^{(L)}}{\partial b^{(L-1)}} \end{cases}$$

$$z^{(L)} = W^{(L-1)}h^{(L-1)} + b^{(L-1)} \mapsto h^{(L)} = f(z^{(L)}) \longmapsto C(h^{(L)}(z^{(L)}))$$

[20] Previo a inventarse el método, el cálculo de la responsabilidad de cada neurona en el resultado se hacía con «fuerza bruta», perturbando aleatoriamente cada una de las neuronas de la red para comprobar si había variación en el error final.

- $\frac{\partial C}{\partial h^{(L)}}$: cómo varía el coste de la red cuando variamos el *output* en la última capa; esto es, la salida de la red. Como ejemplo, si la función de coste es el *ECM* Error Cuadrático Medio:

$$C = \frac{1}{2}\sum_j (y_j - h_j^{(L)})^2 \longmapsto \frac{\partial C}{\partial h_j^{(L)}} = (h_j^{(L)} - y_j)$$

- $\frac{\partial h^{(L)}}{\partial z^{(L)}}$: cómo varía el *output* de la última capa al variar la suma ponderada de la neurona; si nuestra función de activación es la sigmoide:

$$h^{(L)}(z^{(L)}) = \frac{1}{1+e^{-z^{(L)}}} \longmapsto \frac{\partial h^{(L)}}{\partial z^{(L)}} = h^{(L)}(z^{(L)}) \cdot (1 - h^{(L)}(z^{(L)}))$$

- $\frac{\partial z^{(L)}}{\partial W^{(L-1)},\, b^{(L-1)}}$: variación de la suma ponderada respecto a los parámetros:

$$z^{(L)} \left( = \sum_i h_i^{(L-1)} w_i^{(L-1)} + b^{(L-1)} \right) = W^{(L-1)} \cdot h^{(L-1)} + b^{(L-1)}$$

$$\begin{cases} \dfrac{\partial z^{(L)}}{\partial W^{(L-1)}} = h^{(L-1)} \\ \dfrac{\partial z^{(L)}}{\partial b^{(L-1)}} = 1 \end{cases}$$

El producto $\frac{\partial C}{\partial h^{(L)}} \cdot \frac{\partial h^{(L)}}{\partial z^{(L)}} = \frac{\partial C}{\partial z^{(L)}}$ nos «cuenta» en qué grado se modifica el error cuando se produce un pequeño cambio en la suma ponderada de la neurona. Si esta derivada es grande, significa que ante un pequeño cambio en el valor de la neurona, este se verá reflejado en el resultado final y si por el contrario la derivada es pequeña, relativamente no importará cómo variemos el valor de $z^{(L)}$, ya que este no afectará al valor de salida de la red. Por lo tanto, el valor $\frac{\partial C}{\partial z^{(L)}}$ determina la responsabilidad que tienen las neuronas de la última capa en el resultado final y, por tanto, en el error.[21] Este *error imputado* a la neurona se representa como

$$\left[ \frac{\partial \mathbf{C}}{\partial \mathbf{z^{(L)}}} \equiv \delta^{(\mathbf{L})} \right] = \frac{\partial C}{\partial h^{(L)}} \cdot \frac{\partial h^{(L)}}{\partial z^{(L)}} \longmapsto \begin{cases} \dfrac{\partial C}{\partial W^{(L-1)}} = \delta^{(L)} \cdot \dfrac{\partial z^{(L)}}{\partial W^{(L-1)}} = \delta^{(L)} \cdot h^{(L-1)} \\ \dfrac{\partial C}{\partial b^{(L-1)}} = \delta^{(L)} \cdot \dfrac{\partial z^{(L)}}{\partial b^{(L-1)}} = \delta^{(L)} \end{cases}$$

Haciendo el mismo razonamiento para la penúltima capa: si ahora queremos calcular las variaciones del coste frente a los parámetros de la capa $(L-1)$, tenemos:

[21] Esta derivada parcial se denomina: error imputado a la neurona.

$$\text{Función de coste} \longmapsto \quad C\left(h^{(L)}(\underbrace{w^{(L-1)}h^{(L-1)}(\overbrace{w^{(L-2)}h^{(L-2)}+b^{(L-2)}}^{z^{(L-1)}})+b^{(L-1)}}_{z^{(L)}})\right)$$

$$\begin{cases} \dfrac{\partial C}{\partial W^{(L-1)}} = \dfrac{\partial C}{\partial h^{(L)}}\cdot\dfrac{\partial h^{(L)}}{\partial z^{(L)}}\cdot\dfrac{\partial z^{(L)}}{\partial h^{(L-1)}}\cdot\dfrac{\partial h^{(L-1)}}{\partial z^{(L-1)}}\cdot h^{(L-2)} \\ \qquad\qquad = \delta^{(L)}\cdot\dfrac{\partial z^{(L)}}{\partial h^{(L-1)}}\cdot\dfrac{\partial h^{(L-1)}}{\partial z^{(L-1)}}\cdot\dfrac{\partial z^{(L-1)}}{\partial W^{(L-1)}} \end{cases}$$

$$\begin{cases} \dfrac{\partial C}{\partial b^{(L-1)}} = \dfrac{\partial C}{\partial h^{(L)}}\cdot\dfrac{\partial h^{(L)}}{\partial z^{(L)}}\cdot\dfrac{\partial z^{(L)}}{\partial h^{(L-1)}}\cdot\dfrac{\partial h^{(L-1)}}{\partial z^{(L-1)}}\cdot\dfrac{\partial z^{(L-1)}}{\partial b^{(L-1)}} \\ \qquad\qquad = \delta^{(L)}\cdot\dfrac{\partial z^{(L)}}{\partial h^{(L-1)}}\cdot\dfrac{\partial h^{(L-1)}}{\partial z^{(L-1)}}\cdot 1 \end{cases}$$

donde $\frac{\partial h^{(L-1)}}{\partial z^{(L-1)}}$ es la derivada de la función de activación que ya conocemos, quedando solamente por tanto que calcular una derivada más: $\frac{\partial z^{(L)}}{\partial h^{(L-1)}}$, la variación de la suma ponderada de cada neurona de la capa $L$ respecto al *output* de la capa previa $(L-1)$, resultando en $\frac{\partial z^{(L)}}{\partial h^{(L-1)}} = W^{(L)}$.[22] Análogamente, el error imputado a las neuronas de la capa $(L-1)$ es

$$\frac{\partial C}{\partial z^{(L-1)}} = \delta^{(L-1)};$$

lo que hemos realizado en esta capa ya es extensible al resto de capas de la red:

$$\left.\begin{aligned} \delta^{(L)} &= \frac{\partial C}{\partial h^{(L)}}\cdot\frac{\partial h^{(L)}}{\partial z^{(L)}} \\ \delta^{(l-1)} &= \delta^{(l)}\cdot W^{(l)}\frac{\partial h^{(l-1)}}{\partial z^{(l-1)}};\ \ l=0,\ \ldots,\ L \\ \frac{\partial C}{\partial b^{(l-1)}} &= \delta^{(l-1)};\quad \frac{\partial C}{\partial w^{(l-1)}} = \delta^{(l-1)}h^{(l-2)} \end{aligned}\right\} \quad \text{Ecuaciones del } \textit{Backpropagation} \tag{2.2}$$

En síntesis, estas expresiones nos están «contando» cómo tenemos que utilizar el error de la capa anterior para calcular el error en la capa "actual"; hay 2 casos diferentes:

[22] La matriz de parámetros conecta ambas capas, moviendo así el error a la capa anterior en función de las ponderaciones de las conexiones ($W$).

$$\delta^{(L)} = \frac{\partial C}{\partial h^{(L)}} \cdot \frac{\partial h^{(L)}}{\partial z^{(L)}}$$ última capa, el error pertenece a la función de coste.

$$\delta^{(l-1)} = W^{(l)}\delta^{l} \cdot \frac{\partial h^{(l-1)}}{\partial z^{(l-1)}}$$ resto de capas de nuestra red, el error viene dado por la capa $l$.

Una vez tenemos estas expresiones para calcular el error respecto a la capa anterior, necesitamos otras 2 expresiones para, a partir del error en nuestra capa, calcular las derivadas parciales respecto a los parámetros $w$, $b$:

$$\begin{cases} \dfrac{\partial C}{\partial w^{l-1}} = \delta^{l-1} \cdot h^{l-2} \\ \dfrac{\partial C}{\partial b^{l-1}} = \delta^{l-1}, \end{cases}$$

obteniendo así las ecuaciones del *backpropagation*

### 2.3.2. Overfitting y underfitting

En esta sección hablaremos brevemente sobre algo que ocurre frecuentemente en cualquier modelo relacionado con *deep learning*.

**Subajuste**

(*Underfitting*) Ocurre cuando el modelo no es lo suficientemente complejo como para detectar con precisión las relaciones y características de los datos, ya sea por falta de los mismos o por un aprendizaje erróneo o insuficiente.

**Sobreajuste**

(*Overfitting*) Ocurre cuando el modelo aprende o se ajusta demasiado a los datos de entrenamiento y comienza a aprender sus particularidades y no es capaz de obtener un buen rendimiento con datos de entrada nuevos. Como resultado, un modelo que sufra de *overfitting* será incapaz de obtener un buen rendimiento con nuevos datos de entrada ya que su aprendizaje ha pasado a ser específico y no generalizado a los datos.

En el último Capítulo (4) veremos que se deben ir variando los parámetros de nuestra red para evitar el overfitting; es el caso más probable debido a que: disponemos de 3 tipos de imágenes que no se encuentran en la misma proporción (2040 Circulares - 303 Cometas - 16 Donut), optando por aumentar los datos (rotando las imágenes por ejemplo).[23] También, el optar por redes muy complejas deriva en

[23] En la Sección 3.2.1 del siguiente capítulo se hablará sobre algunas regularizaciones para evitar en lo posible el *overfitting*.

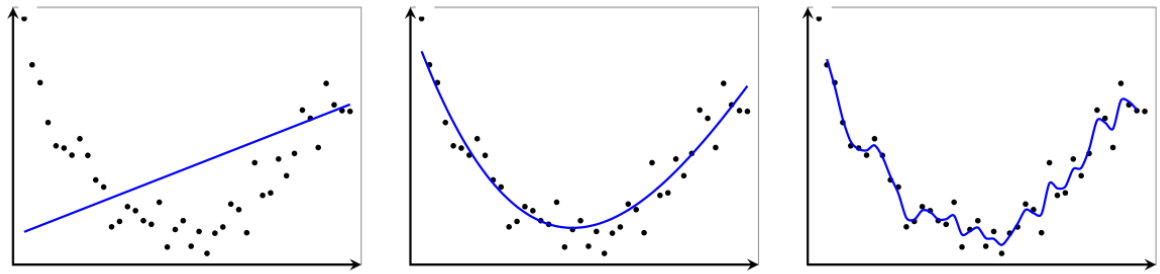


Figura 2.12: *Underfitting*, *Fitting*, *Overfitting*, respectivamente, herramienta *Tikz*.

*overfitting*, al igual que podemos ver en Figura 2.12, donde ajustar con demasiada precisión tiende al error en las predicciones.

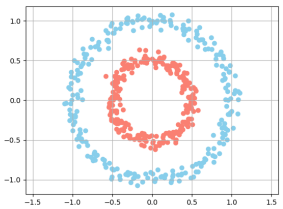
Es por ello que, para evitar esto durante la implementación en código, se separan las imágenes en listas de entrenamiento y validación (listas disjuntas) para ir testando el entrenamiento. También se requiere de imágenes extra, las de *evaluación* o test, las cuales no se hayan incluido durante el entrenamiento para ver qué tan buenos resultados predice el modelo. Por tanto, buscaremos que la precisión tanto de las listas de *entrenamiento* y *validación* sean lo máximas posibles, además de similares; por ejemplo, si la lista de *validación* proporciona malos resultados en contraposición de la de *entrenamiento*, esto significaría que la red estará sobre-entrenándose (*overfitting*), pues se estaría ajustando a los datos en «exceso» y las comprobaciones con las imágenes de *validación* no serían representativas.

## 2.4. Ejemplos de Implementación de Redes Neuronales

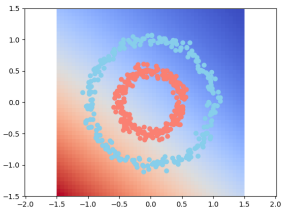
Vamos a ver dos ejemplos de los resultados de implementar redes neuronales para clasificación: clasificación binaria y clasificación multiclase de imágenes, respectivamente.

### 2.4.1. *Make Circles* ANN

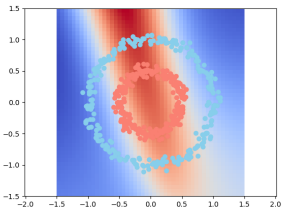
Este primer ejemplo consiste en, dadas dos *nubes* de puntos disjuntas, clasificarlas; es decir, separar ambos grupos como podemos ver a continuación a través de las épocas del entrenamiento:



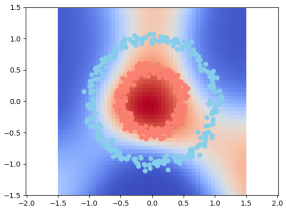
(a) Problema original



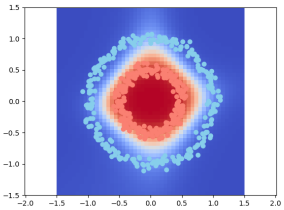
(b) Época 50 de entrenamiento



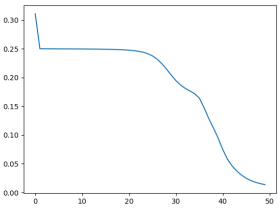
(c) Época 1500 de entrenamiento



(d) Época 2000 de entrenamiento



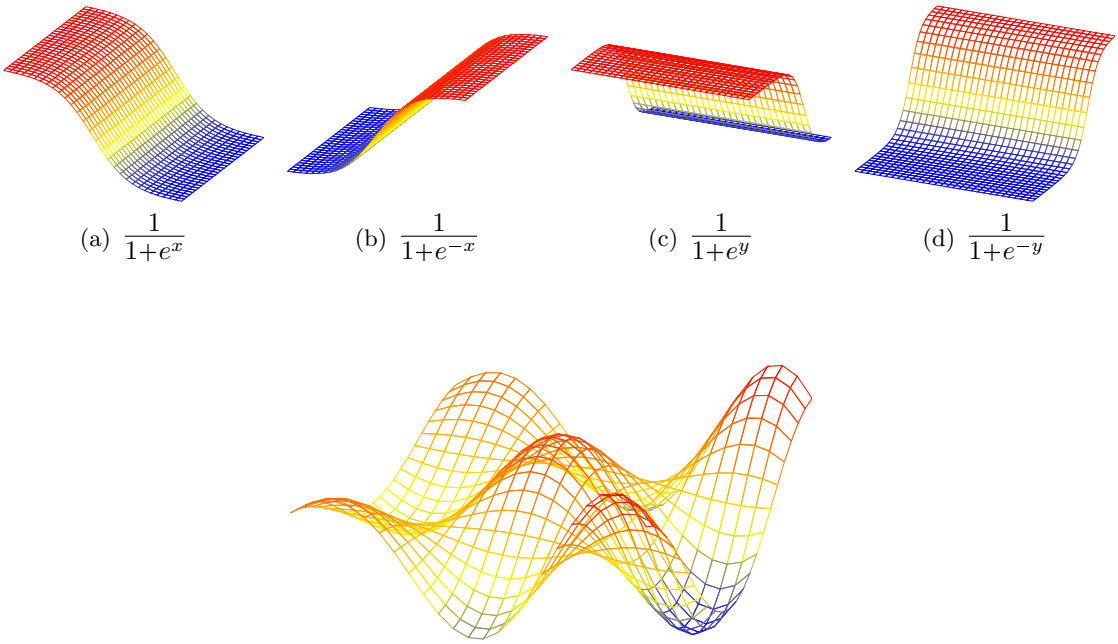
(e) Época 2500 de entrenamiento



(f) La función de coste se minimiza hasta tender a 0.

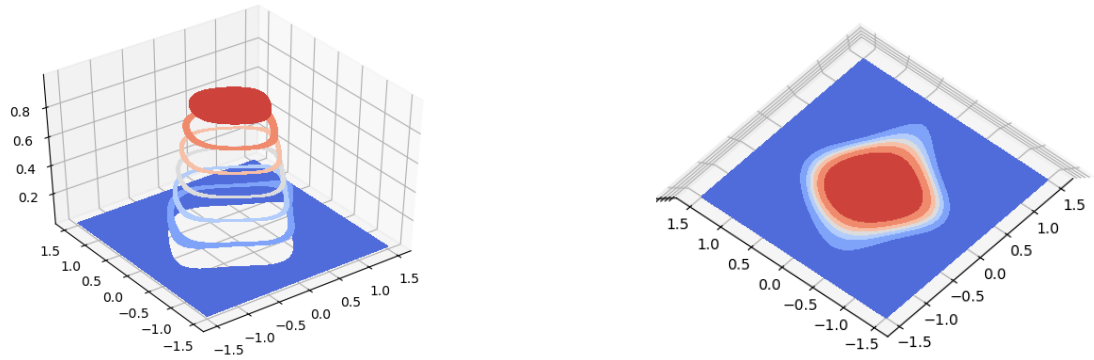
Figura 2.13: Ejemplo: *Make Circles*

Vemos en estas figuras la mejora en la clasificación a medida que se avanza en el entrenamiento ([1]). Algo que ocurre aquí, es que la función estimadora de la red neuronal se intersecta con el plano $z = 0$; es decir, se proyecta. Para la estructura de la red, se parte inicialmente de 2 nubes de puntos que serán los *inputs*; luego, 2 capas ocultas de 4 neuronas con la función sigmoide como función de activación y una capa de salida que en la que 0 determina que el grupo es azul y 1 es la etiqueta asignada al grupo rojo. Como observación, de la combinación de 4 sigmoides con diferente disposición en diferentes capas, con los parámetros de la red, da lugar a una función *estimadora* de la red 2.14(e):



(a) $\frac{1}{1+e^{x}}$ (b) $\frac{1}{1+e^{-x}}$ (c) $\frac{1}{1+e^{y}}$ (d) $\frac{1}{1+e^{-y}}$

(e) Ejemplo de función estimadora de una neuronal con funciones de activación sigmoides.



(f) (g)

Figura 2.14: Ejemplo: *Make Circles*

La combinación de las funciones sigmoides resulta en una función similar a 2.14(e) (ha sido ilustrada mediante la herramienta *Tikz* con vistas a ver cómo quedaría finalmente la función estimadora de la red)[24]; las figuras 2.14(f)-2.14(g) representan, respectivamente, las predicciones del entrenamiento de la red, siendo esta segunda

[24] Para mayor precisión, podemos saber cuáles son los parámetros de la red para cada capa, con *get_weights* en *python*.

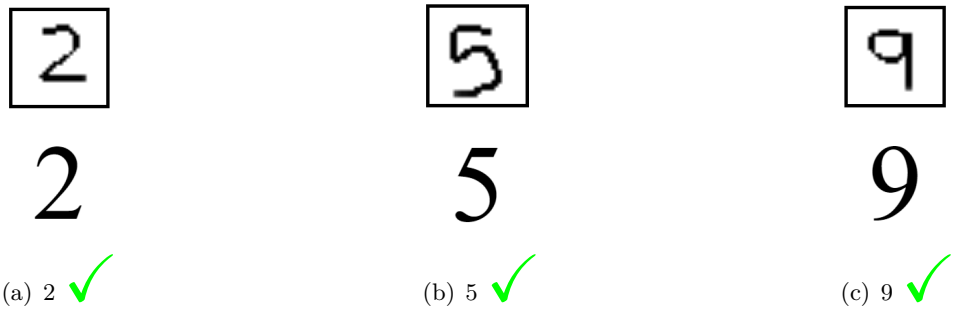
figura la proyección sobre el plano, resultando en la imagen que clasifica las *nubes* de puntos.

### 2.4.2. *MNIST* *ANN*

*MNIST* Modified National Institute of Standards and Technology, [3], es un dataset el cual consta de $50.000 - 10.000$ imágenes ($28 \times 28$ píxeles) con dígitos del 0 al 9 escritos a mano para entrenamiento-test, respectivamente. Podemos hacer uso de este gracias a que se encuentra por defecto en el módulo de *keras.datasets* de *python*.

Como ejemplo ilustrativo previo, [2], supongamos que tenemos una entrada $\mathbf{x}$ de las lecturas de píxeles en escala de grises de una imagen. $\mathbf{y}$ es la salida, un vector 27-dimensional que designa, con un 1 o un 0, qué letra del alfabeto se muestra en la imagen; es decir $(1, 0, \ldots, 0)$ para la letra **a**, $(0, 1, 0, \ldots, 0)$ para **b** y así sucesivamente. Este vector de salida de 27 dimensiones podría usarse para clasificar letras en imágenes.[25] Esta manera de proceder se denomina *one-hot encoding*[26].

Para analizar la calidad de la red neuronal hecha para este problema de clasificación, se ha hecho un archivo *HTML* (*tensorflow*, *javascript*, código en el enlace de *github*) para poder entregarle a la red nuestra escritura «en tiempo real». Vemos cómo este predice



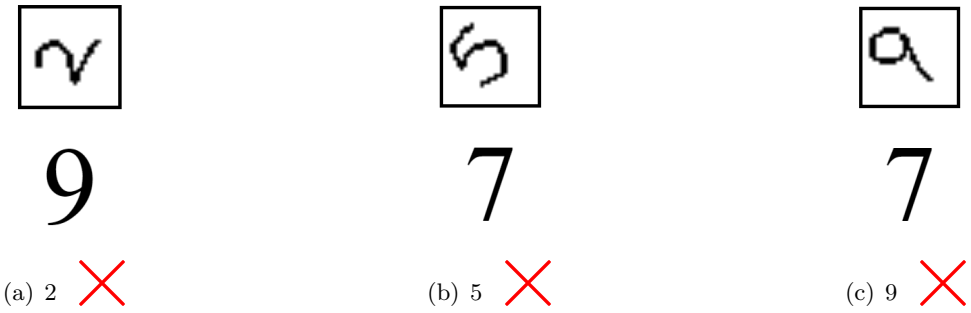

(a) 2 ✓ (b) 5 ✓ (c) 9 ✓

Figura 2.15: MNIST

de forma correcta en todos los ejemplos.

Le hemos pasado a la red dígitos centrados y bastante *parecidos* a los que se encuentran en el propio *dataset* de *MNIST*. Las redes neuronales "convencionales" pueden llegar a ser muy potentes; sin embargo, no lo son lo suficiente en cuanto a reconocimiento de imágenes se refiere, como podemos ver a continuación:

[25] El último ejemplo del presente capítulo realiza esto pero con dígitos escritos a mano del 0 al 9.
[26] *.to_categorical*() en *python*.




(a) 2 (b) 5 (c) 9

Figura 2.16: Limitaciones de las **ANNs**

Cuando le entregamos dígitos diferentes a los «aprendidos», la predicción colapsa; incluso el hecho de enviarle cualquier tipo de escritura obliga a la red a devolver un valor, resultando la predicción en la aleatoriedad.[27]

Esto se debe a que la red ha sido entrenada de manera eficiente para reconocer imágenes similares a las que han sido entrenadas, pues la red recibe como valores de entrada cada uno de los píxeles de la imagen (si se trata de una imagen $28 \times 28$, recibirá 784 pixeles) como inputs independientes; es decir, que la red no atiende a los pixeles de alrededor de cada uno de estos, siendo por tanto la red neuronal multicapa incapaz de detectar formas, patrones, etcétera.

Necesitamos de otra herramienta para poder clasificar imágenes con una mayor precisión, que aprendan *correctamente*; por ello damos paso a las CNN, las cuales se han diseñado para la tarea del procesamiento de imágenes.

[27] Ver Figura 3.10.

# 3

# Redes Neuronales Convolucionales

En el capítulo anterior mostramos un ejemplo en el que entrenamos una red neuronal para leer números escrito a mano y pasarle a esta nuestra propia escritura [ejemplo 2.4.2]. Sin embargo, estábamos empujando los límites de la red para realizar dicha tarea de *leer* ya que en realidad, como hemos visto, este tipo de redes no han sido diseñadas para ello. Existe un tipo de red que nos va a permitir entender imágenes con un gran nivel de precisión; hablamos de las Redes Neuronales Convolucionales, **CNN** Convolutional Neural Network.

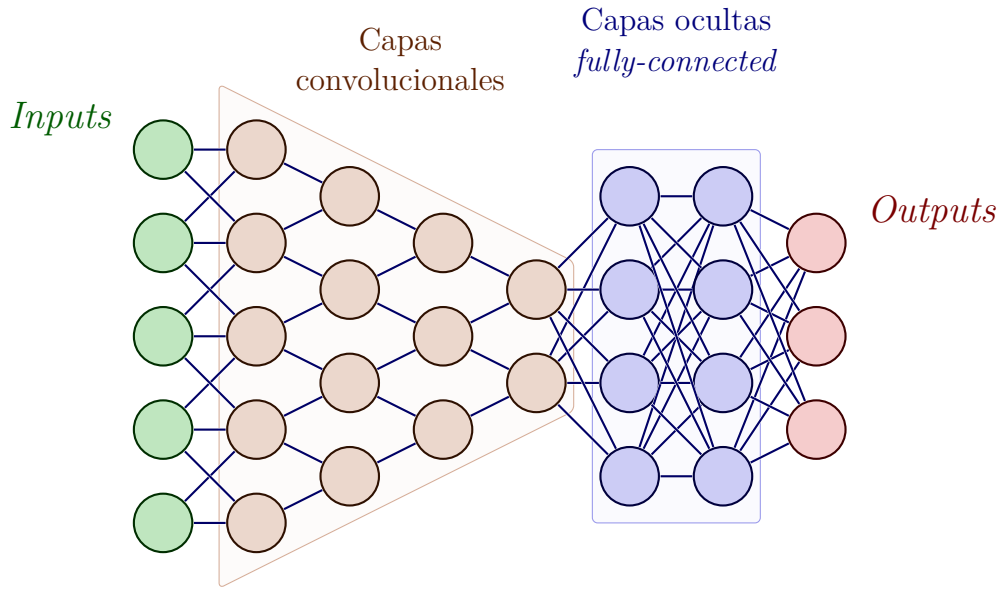



Figura 3.1: Estructura de una CNN, 3.2.1, herramienta *Tikz*, [6].

Para poder llegar a usarla y entenderla, necesitamos enfocarnos en una de sus herramientas fundamentales, el proceso de convolución.

## 3.1. Proceso de Convolución

Muchos objetos que queremos detectar en una imagen dependen de que podamos encontrar y entender los ejes (bordes) de la imagen: dónde están, hacia donde apuntan, etcétera.

Para el ojo humano es simple; si en la imagen hay píxeles contiguos que cambian drásticamente de color, podemos decir «con confianza» que se trata de un borde (los denominaremos ejes). Los ordenadores no ven las imágenes igual que nosotros, sino una serie de píxeles donde cada uno de ellos se conforma de valores numéricos indicando sus niveles de color.

*Nota.* Cada píxel se codifica mediante un conjunto de bits de longitud determinada (la profundidad de color); por ejemplo, en la escala de grises (8 bits), admitiendo así cada píxel hasta $2^8 = 256$ ($[0, 255]$) variaciones de color). Trabajaremos principalmente con la escala de grises debido a que nuestras imágenes astronómicas se presentan en blanco y negro (píxel negro: valor 0 mínimo y píxel blanco: valor 255 máximo).

La única manera de que el ordenador pueda detectar estos ejes, es haciendo una comparativa de cada píxel con los de alrededor y ver si hay cambios fuertes en los valores de los estos;

| | | |
|---|---|---|
| 170 | 171 | 76 |
| 179 | 163 | 54 |
| 176 | 145 | 85 |

Tabla 3.1: Ejemplo de píxeles en una *imagen*, con vistas a ilustrar el efecto de los bordes, 3.5.

es decir, que para poder detectarlo en una imagen no podemos hacerlo viendo los píxeles de manera individual, como ocurre en las redes que vimos en el capítulo anterior, donde cada imagen es un vector de píxeles independientes.[1] Para ello usaremos una pequeña herramienta que nos ayudará con este problema: núcleos (filtros o *kernels*).

### 3.1.1. *Kernels*

Definamos un cuadro de tres píxeles de alto por tres de ancho,

[1] El píxel central de cada bloque $3 \times 3$ lo denominamos: píxel principal; ejemplo Tabla 3.1.

| | | |
|---|---|---|
| $b_{11}$ | $b_{12}$ | $b_{13}$ |
| $b_{21}$ | $b_{22}$ | $b_{23}$ |
| $b_{31}$ | $b_{23}$ | $b_{33}$ |

Tabla 3.2: Kernel

Este cuadro es llamado: **núcleo** o **kernel**, que se aplicará sobre los píxeles de una imagen[2] para realizar transformaciones; luego se extenderá el ejemplo a una imagen al completo. Cogiendo como ejemplo dicha «matriz» de píxeles (Tabla 3.1) y asignando a los valores del kernel: $b_{22} = 1$, $b_{ij} = 0$; $\forall\ i, j \neq 2$, la manera de actuar del filtro a una imagen cualquiera es la siguiente:

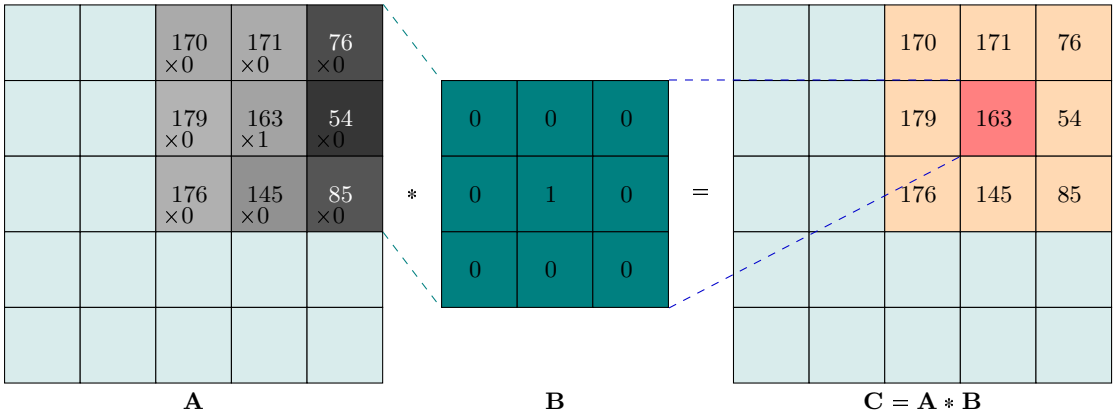



Figura 3.2: Convolución de matrices: Tabla 3.1 y el *kernel identidad*

La forma de operar: se multiplica cada valor numérico del núcleo $B$, por la misma posición de la matriz de píxeles $A$, colocando el *kernel* sobre cada pixel principal de la imagen (en este caso de ejemplo, una "imagen" $5 \times 5$ tomando como ilustración el píxel principal de valor 163); luego se suman los 9 valores obtenidos. Posteriormente, se asigna ese valor resultante a la misma posición del píxel principal (cada *pixel* de la matriz original), en una nueva imagen. Por tanto, el filtro lo que hace es cuantificar la importancia de cada uno de los píxeles a la hora de hacer una transformación a una imagen dada. Como el núcleo tiene un valor de 0 en todas las casillas de alrededor, los píxeles asignados a esas casillas no están siendo considerados para el cálculo; solo lo está siendo el pixel principal ($\times 1$), resultando por tanto la convolución 3.2 una transformación *identidad*.

Las operaciones que hemos realizado se extrapolan a la transformación de un píxel de una imagen cualquiera (cuyos 9 píxeles tomados de una imagen como ejemplo son Tabla 3.1). Esta operación se aplicará a todos y cada uno de los píxeles de la imagen cada vez poniendo el píxel que estamos "revisando" (píxel principal) en

[2] Usualmente se usan filtros de tamaño $3 \times 3$, $5 \times 5$, $7 \times 7$, etcétera.

el centro del núcleo, realizando la operación y colocando el resultado en la nueva imagen uno por uno y así sucesivamente hasta haberlo hecho con la totalidad de la imagen.

En cuanto a los pixeles (de las imágenes a transformar) que se encuentren en los bordes de la imagen, dependiendo del filtro, estos variarán o no, pues solo se considerarían los pertenecientes a la imagen en sí (por ejemplo, en la Figura 3.2 el kernel no se aplicaría a los píxeles principales con valores 170, 76, etcétera.). No obstante, en general, se suele «ampliar» la imagen con vistas a poder aplicar el filtro sobre los que considerar estos como nulos[3]:

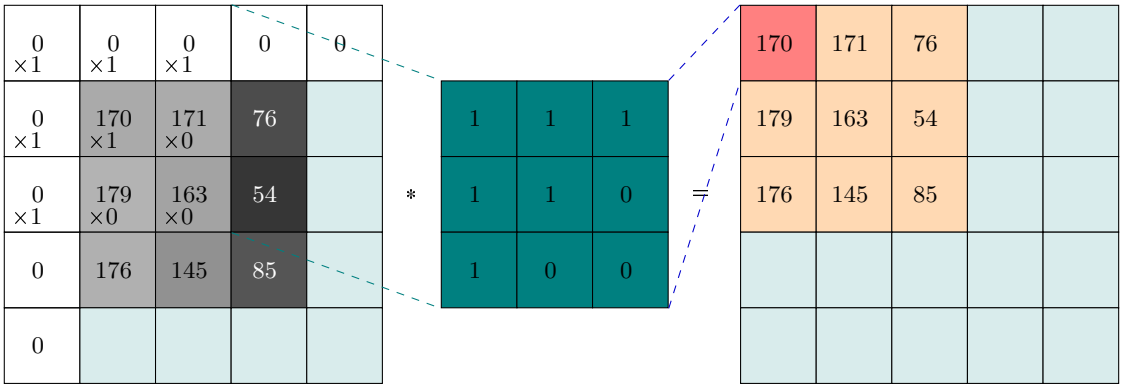



Figura 3.3: Convolución de matrices en el borde izquierdo superior de una imagen aleatoria, usando *zero-padding*.

Matemáticamente, tenemos la **convolución** de matrices de forma generalizada:

$$\underbrace{\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} & A_{15} \\ A_{21} & A_{22} & A_{23} & A_{24} & A_{25} \\ A_{31} & A_{32} & A_{33} & A_{34} & A_{35} \\ A_{41} & A_{42} & A_{43} & A_{44} & A_{45} \\ A_{51} & A_{52} & A_{53} & A_{54} & A_{55} \end{bmatrix}}_{A} \circledast \underbrace{\begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}}_{B} = \underbrace{\begin{bmatrix} C_{11} & C_{12} & C_{13} & C_{14} & C_{15} \\ C_{21} & C_{22} & C_{23} & C_{24} & C_{25} \\ C_{31} & C_{32} & C_{33} & C_{34} & C_{35} \\ C_{41} & C_{42} & C_{43} & C_{44} & C_{45} \\ C_{51} & C_{52} & C_{53} & C_{54} & C_{55} \end{bmatrix}}_{C}, \text{ donde}$$

$$C_{i,j} = \sum_{m=-1}^{1} \sum_{n=-1}^{1} b_{m+2,n+2} \cdot A_{i+m,j+n} \qquad dim(B) = 3,$$

$$C_{i,j} = \sum_{m=-2}^{2} \sum_{n=-2}^{2} b_{m+3,n+3} \cdot A_{i+m,j+n} \qquad dim(B) = 5,$$

[3] Esto es, por ejemplo, opcional en *python*, pudiendo dar como salida de la transformación una imagen del mismo tamaño de la original o eliminar los píxeles del borde (*padding*). Si no se aplica, los píxeles de los bordes no serán considerados para la convolución y la transformación dará una imagen de menor tamaño.

⋮

Cada elemento de la matriz inicial (haciendo mención a la notación en Figura 3.2) $A$ sería equivalente a la imagen de la que disponemos, conformada con los valores de sus píxeles; queremos aplicar el filtro $B$ a esta, generando una matriz $C$ de igual tamaño que la imagen original.[4]

Como vemos, aplicar el filtro *identidad* (Figura 3.2) proporciona la imagen original. Si hacemos el mismo proceso, por ejemplo, con un núcleo de matriz con valores de $\frac{1}{9}$, ahora todas las casillas tienen valor e importancia en la transformación; nos da como salida un desenfoque (comúnmente llamado *blur*). Cada uno de los píxeles se transforma en un promedio de todos los que tiene alrededor, por lo que los bordes se difuminan, la claridad se pierde y el resultado es una imagen un poco más desenfocada.

Este proceso de pasar un núcleo a través de toda una imagen, multiplicar los valores, sumar el resultado y pasarlo a una nueva imagen es llamado **convolución**.[5]

### 3.1.2. Ejemplos de filtros de convolución

Estas «celdas» de los *kernels* las podemos modificar para hacer diversas transformaciones sobre las imágenes. Veamos unos pocos núcleos (con vistas a ilustrar lo mencionado en la sección anterior) bastante utilizados en el procesamiento de imágenes, cogiendo como ejemplo una imagen de una estrella tipo "*Donut*" de nuestra lista de imágenes astronómicas.

*Nota.* Al tratarse de imágenes con poca resolución ($60 \times 60$ píxeles), los efectos de las transformaciones no son tan notorios, mostrando por tanto solo una breve variedad de núcleos:[6]

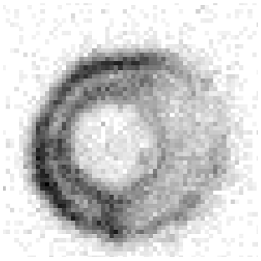


Figura 3.4: Imagen de ejemplo tipo *Donut*, estrellas astronómicas

[4] A mayor dimensión del núcleo, $3 \times 3$ en nuestro ejemplo, más píxeles afectarán a la transformación de cada píxel principal.

[5] Recibe el nombre derivado de la convolución de matrices debido a que las imágenes computacionalmente hablando son matrices.

[6] Convoluciones: librerías de *scipy*, PIL (*Pillow*) u *Open-CV* en *python*.

$$\text{Identidad} \longmapsto \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$\text{Difuminado } (\textit{blur}) \longmapsto \frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

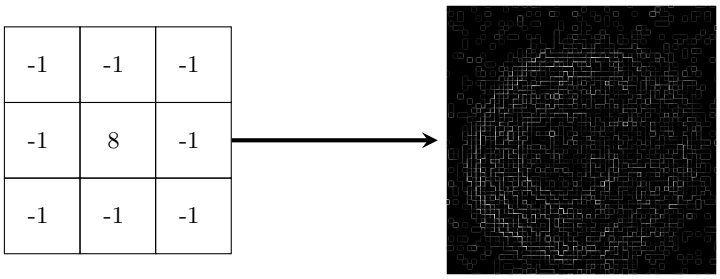



Figura 3.5: Detección de ejes

Regresando al problema de la detección de ejes, si queremos detectarlos, la idea es cuando exista uno de estos, «marcarlo» a píxeles blancos y donde no los haya, a pixeles negros (como en Figura 3.5). El objetivo del núcleo de detección de bordes es que el resultado sea muy alto (blanco) cuando hay mucha diferencia entre el píxel central y los de alrededor y que el resultado sea muy pequeño (negro) cuando los píxeles sean muy similares.

La importancia de los ejes cuando hablamos de las **CNN** reside en que la red aprenderá a reconocer estos cambios drásticos variando los parámetros de todos los núcleos que se utilicen para el entrenamiento de la red (inicializados estos aleatoriamente, actualizándose de forma análoga a como veíamos que los parámetros $W$, $b$ lo hacen), pudiendo así reconocer la mayor cantidad de patrones posibles (ver Sección 3.2.1).

## 3.2. Estructura CNN

Para dar a la estructura de este tipo de redes, vamos en primer lugar a hablar brevemente sobre los principios básicos que hay detrás del proceso de visión humana: el ser humano, entre otras muchas cosas, es capaz de reconocer caras; esto se debe a que ha detectado la presencia de algunos elementos que conforman a un rostro por nuestro aprendizaje: ojos, boca, nariz, etcétera. Luego, sabemos qué es un ojo

porque detectamos: una pupila negra, líneas que son pestañas, superficies blancas y demás, pudiendo reconocerlos al ser capaces de detectar patrones circulares, cambios de contrastes, texturas, entre otras cualidades.

Si reproducimos los pasos que realiza nuestro córtex visual, nos encontramos con un «procesamiento en cascada» donde primero se identifican aquellos elementos básicos y generales, combinándose posteriormente para generar patrones cada vez más complejos.

Una red neuronal convolucional CNN es aquella cuyo diseño ha sido pensado para sacar partido a la estructura espacial de una imagen; es decir, en una red neuronal clásica, se introduce el valor de cada pixel como si de una variable independiente se tratara, como si fuera un vector plano; significaría esto que daría lo mismo introducir imágenes a la red con una cierta ordenación. La red convolucional tendrá en cuenta la dependencia de la posición de los píxeles (pues el valor de un píxel va a estar muy ligado al de sus vecinos, alto y ancho); analizadas correctamente las estructuras, formas y patrones , pueden beneficiar a la red a la hora de entender qué se está observando: con esta idea surgen las **CNN**.

Una *CNN* es un tipo de red neuronal que se caracteriza por aplicar un tipo de capa donde se realiza una operación matemática conocida como **convolución**, como ya hemos comentado previamente. Se configuran filtros que determinan diferencias de contraste, que sirve a la red para detectar bordes y patrones en general; esta operación de convolución sobre una imagen detecta propiedades diferentes según cuáles sean los valores del filtro que apliquemos o definamos.

Al igual que hablábamos en el capítulo anterior de cómo la red *aprendía* ajustando los parámetros (pesos) de esta, en este tipo de redes los parámetros a ajustar serán los valores de estos filtros, inicializados aleatoriamente para luego ir ajustándose por medio de los algoritmos del descenso del gradiente y backpropagation. Gracias a estos filtros, se detectarán patrones y se generarán nuevas imágenes; a cada una de estas imágenes generadas se las conoce como: mapa de características, ya que actúa como un mapa donde se indica en qué parte de la imagen se ha detectado la característica buscada por dicho filtro.[7]

Por tanto, una imagen «entra» a la red, se aplica una serie de convoluciones y genera un conjunto de mapas de características. Principalmente, las *CNN* constan básicamente del proceso de convolución; sin embargo, el potencial de este tipo de redes se encuentra en que esta operación se va a realizar secuencialmente (al igual que el «procesamiento en cascada» de la visión humana), donde el output de una de las capas se va a convertir en el input de la siguiente.

[7] Un mapa de características de una imagen puede ser un ojo; otro mapa en el siguiente proceso de convolución puede ser una pupila.

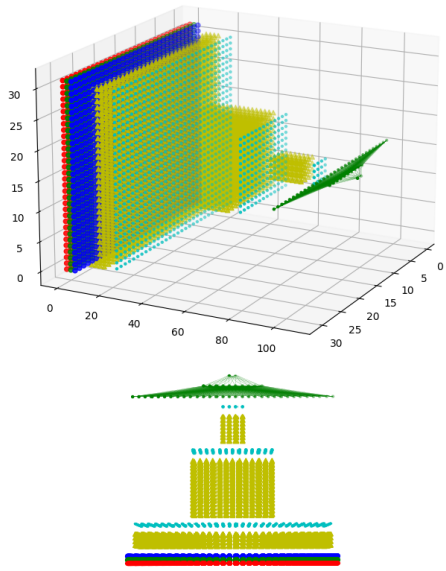



Figura 3.6: Estructura de una CNN, 3.2.1, generada en *python*.

El reconocer patrones cada vez más complejos lo encontramos en la sucesión de capas (*deep learning*, en la profundidad de la arquitectura de nuestra red). La operación de convolución se irá potenciando paso a paso; donde antes teníamos una región de 9 píxeles (Tabla 3.1 por ejemplo), nuestro filtro lo transforma en un único píxel de información y, por tanto, si volvemos a aplicar sucesivamente convoluciones sobre los mapas de características de una imagen, estaremos accediendo cada vez a más información espacial de la imagen original. Esto es algo que además podemos incentivar reduciendo la resolución de nuestros mapas de características (lo que se conoce como *Max Pooling* (ver Sección 3.2.1)).

En conclusión, la operación de convolución por sí sola no puede detectar formas muy complejas más que bordes y patrones muy simples pero, volver a hacer estas detecciones sobre las anteriores nos permite componer cada vez patrones más complejos. Es por ello que el diseño de una arquitectura convolucional normalmente se representa en artículos y papers como una especie de *embudo* donde la imagen inicial se va comprimiendo espacialmente; es decir, su resolución va disminuyendo al mismo tiempo que su grosor va aumentando. Es decir, el número de mapas de características que vamos detectando va en aumento, acabando todo en un punto donde habremos detectado todos los patrones necesarios, contando entonces con muchos mapas de características que ahora sí podremos dar como *inputs* independientes dentro de una red neuronal multicapa clásica, que acabará por tomar la decisión de qué es lo que se presenta en nuestra imagen original.

Esta forma de *embudo* la podemos apreciar en la Figura 3.6: las 3 primeras capas representan los canales *RGB*, las capas amarillas la cantidad de *kernels* con 4, 16 y 8 respectivamente, cuyas láminas son de $32 \times 32$ por el tamaño de la imagen de entrada. Las láminas o capas posterior a estas son las *MaxPooling* y luego en verde se representa la clasificación mediante una red neuronal clásica.

Ahora explicaremos las componentes principales de esa estructura.

### 3.2.1. Procesos durante una *CNN*

Explicado como funcionan las redes convolucionales, enumeremos las capas (o procesos) que van sucediendo a medida que avanzamos en la red, comentando las tareas fundamentales que estas realizan:

- Fase convolucional CNN:

  1. **Capa de convolución**: (*conv2D()* en *python*) Esta capa utilizará una cierta cantidad de *kernels* para aplicar a la imagen que se entregue a la red como *input*; cada *kernel* genera un conjunto de características de la imagen (líneas verticales, bordes, etcétera). A través de las épocas de entrenamiento, las componentes de los *kernels* se irán actualizando, para obtener mediante el aprendizaje aquellos patrones que hagan que la red reconozca cada vez con mayor precisión las imágenes. Al ir aplicando sucesivamente capas de convolución, se irán detectando patrones más complejos.[8]
  2. **Reducción de dimensionalidad**: (*MaxPooling2D()* en *python*) Esta capa reduce la dimensión de la imagen, consiguiendo principalmente aumentar la abstracción sobre los datos de entrada.[9] Se van cogiendo grupos, usualmente, de $2 \times 2$ y nos quedamos con el máximo valor de cada grupo de $2 \times 2 = 4$ píxeles. De esta forma reducimos la imagen a la mitad de tamaño, consiguiendo concentrar más la información; este procedimiento se plasma en la siguiente figura:

*Nota.* Cada bloque $2 \times 2$ «coloreado» es lo que se denomina *pool size*, lo que se puede modificar en *python*. En cuanto a la capa de convolución, el ir pixel por pixel (principal) realizando la convolución asociando un filtro, se denomina *stride*, también modificable.

[8] Usualmente en cada capa de convolución se incrementan los *kernels* en una potencia de 2 (32, 64, 128, ...).

[9] El resultado de ir incrementando el número de filtros y disminuyendo la dimensionalidad de la imagen da lugar a 3.6.

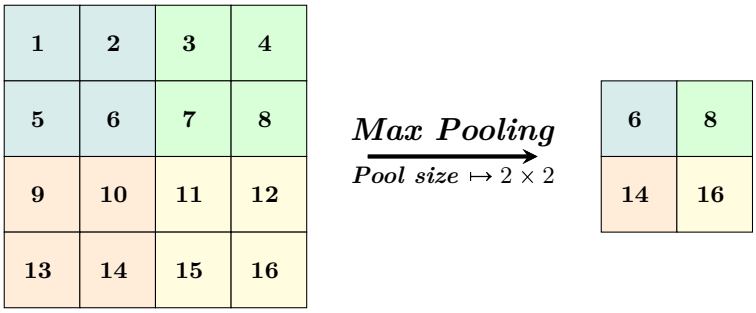



Figura 3.7: *MaxPooling*

- **Aplanamiento de las características**: (*Flatten()* en *python*) Una vez terminada la etapa de convolución, se aplanan los datos; es decir, se convierte la imagen $2-dimensional$ en un vector plano $(1D)$.
- Etapa de clasificación ANN: (*Dense()* en *python*) Una vez aplanados los datos, todas las propiedades (o características) asociadas a cada una de las imágenes del entrenamiento, se «clasifican»[10] dentro de la red neuronal multicapa. La multiplicación de la cantidad de píxeles de todas las imágenes generadas (*mapas*) por el total de estas imágenes (tantas como filtros aplicados en la última capa de convolución) determinará el tamaño del vector *flatten*, cuya dimensión será la cantidad de inputs que recibirá cada neurona de la red ($(15 \times 15) \cdot 64$ filtros $= 14400$ *inputs* en el ejemplo a continuación):

```
 Layer (type)                 Output Shape              Parameters
=================================================================

 (Conv2D)                  (None, 60, 60, 32)            320

 (MaxPooling2D)            (None, 30, 30, 32)            0

 (Conv2D)                  (None, 30, 30, 64)          18496

 (MaxPooling2D)            (None, 15, 15, 64)            0

 (Flatten)                   (None, 14400)               0

 (Dense)                      (None, 50)               720050

 (Dense)                      (None, 3)                  153
```

- ***Dropout***, regularización: (*Dropout*$(p)$ en *python*, con $p \in [0, 1]$) con vistas a reducir el *overfitting*, se hace uso de esta técnica para que, en cada época o ciclo de entrenamiento, una cierta cantidad de neuronas se «apaguen» de forma probabilista[11]. En general, hay pesos que tienden a ser muy grandes en comparación con otros en el ajuste de estos, por lo que estos últimos no tienen

[10] Una vez «todas» las propiedades de las imágenes han sido detectadas, la red neuronal clasificará asociando estas propiedades con el valor de salida, las etiquetas de las imágenes.
[11] La probabilidad viene determinada por el parámetro $p$.

mucha influencia en las decisiones de la red; es por ello que al "apagar" neuronas aleatoriamente con una cierta probabilidad, las neuronas con menos influencia trabajarán más, haciendo así a la red más robusta en el entrenamiento.

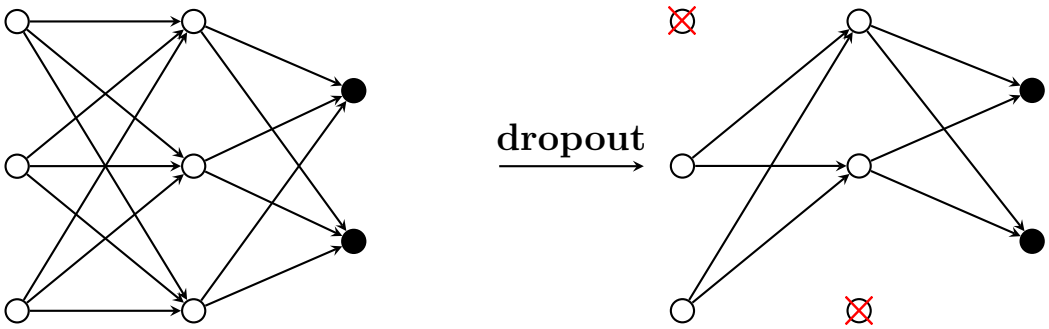



Figura 3.8: Dropout, con probabilidad $P(0.3)$

También hay otras técnicas que usaremos en la red (en busca del mejor modelo de predicción) como pueden ser las regularizaciones $L1$ y $L2$ (*Lasso* y *Ridge* respectivamente)[12], el *Data Augmentation* para reducir el *overfitting* añadiendo variaciones de las imágenes de entrada para que la red no reciba siempre el mismo patrón de imágenes, optimando el entrenamiento; *Early Stopping* para «detener» el entrenamiento cuando las precisiones del entrenamiento y validación sean "máximas", entre otras técnicas.

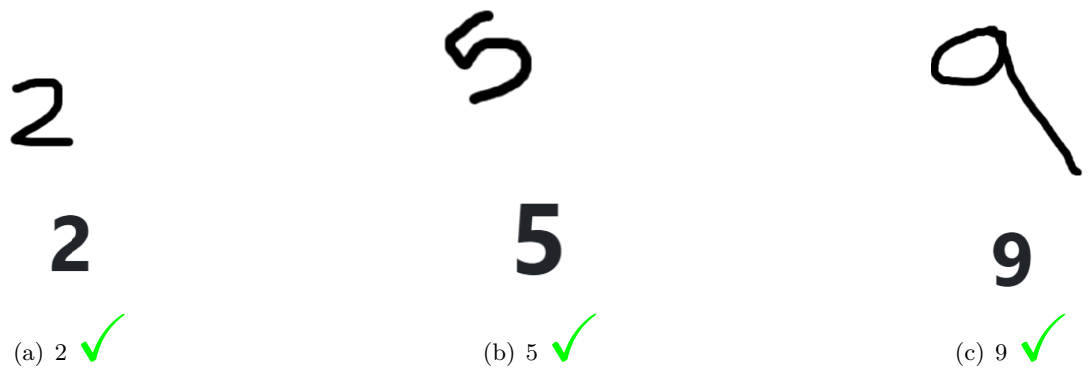
*Nota.* Existen diversas formas de tratar el *overfitting* y, en general, mejorar los resultados de las redes neuronales; no se hace especial énfasis debido a que no es el objetivo principal del trabajo.

Para concluir, veamos el ejemplo de *MNIST* del capítulo anterior, aplicado ahora sobre redes neuronales convolucionales, CNN.

[12] Estas hacen referencia a las normas $\mathcal{L}^1$ y $\mathcal{L}^2$ aplicadas a la función de coste.

## 3.3. Ejemplo MNIST *CNN*

Como ilustramos en el capítulo anterior, las redes neuronales en cuanto a reconocimiento de imágenes se refieren, no son lo suficientemente potentes; vimos como, si variábamos el ángulo de escritura así como el tamaño o la posición en cualquiera de los ejes (ver Ejemplo 3.9), la red colapsaba en una predicción errónea. Por tanto, apliquemos lo visto hasta ahora para crear una red neuronal convolucional CNN con el mismo propósito, el de clasificar dígitos escritos a mano del 0 al 9:




(a) 2 ✓ (b) 5 ✓ (c) 9 ✓

Figura 3.9: MNIST CNN

Vemos como ahora la red predice muchísimo mejor nuestra escritura, pudiendo escribir dígitos orientados de diversas formas y tamaños (se puede encontrar el archivo *.html* en el repositorio de *GitHub*, además de una página web hecha para poder ir directamente a usar la red entrenada).

*Nota.* No obstante, siempre se pueden encontrar errores y mejorar la estructura en general; por ejemplo, si entregamos a la red una figura que no sea un número, esta intentará predecir un valor numérico entre $0-9$, errando en el intento.

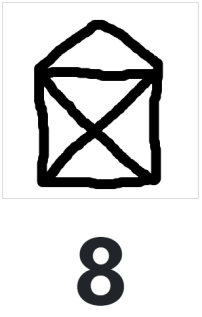


Figura 3.10: Fallos a *pulir* en la implementación en una *CNN*

# 4

# Clasificación de Imágenes Astronómicas

## 4.1. Resultados

Vamos a aplicar todo lo visto para crear la red neuronal convolucional que clasificará nuestras imágenes astronómicas, el objetivo principal del trabajo. Luego de crear la red, la entrenaremos y evaluaremos el modelo «pasándole» a un programa *python* una lista de imágenes reservadas para esta tarea, la lista de evaluación.

*Nota.* La parte más importante del proceso es la de entrenamiento; para las listas de validación y evaluación, podemos repetir algunas imágenes con las que se ha entrenado a la red, aunque no es lo más "adecuado", pues la red podría *memorizar* los resultados entrenados. En nuestro caso, primeramente se repitieron algunas imágenes en las diferentes listas; se comprobó que el gran porcentaje de acierto que se había obtenido en el entrenamiento en los primeros intentos de modelos de clasificación era «falso» (Sección 2.3.2).

La clasificación que usaremos será la siguiente:

```
class_names=['Circular', 'Cometa', 'Donut']
```

con 3 tipos de imágenes; como mencionamos en el primer capítulo, podemos clasificar las estrellas que vemos circulares tanto como *good seeing* o *bad seeing* (buena o mala visibilidad fijándonos en el *seeing* $\mapsto$ *fwhm*). Sin embargo, vamos a predecir resultados con esta clasificación teniendo en cuenta solamente la forma de las estrellas.

Entonces, siguiendo con la explicación del procedimiento en el Capítulo 1, comenzamos leyendo los datos proporcionados por los archivos *.h5* referentes a las listas de entrenamiento y validación (esta segunda para validar el correcto entrenamien-

to del modelo). Se «reestructuran»[1] los datos y lo convertimos a tipo flotante para que la red aprenda más eficientemente, así como pasarle también los valores entre 0 y 1 con el mismo propósito.[2] Como comentamos en el Capítulo 2, realizamos *one-hot encoding*, lo que resulta en un vector de tamaño $n\_classes \mapsto 3$ en nuestro caso, donde solo el resultado correcto (la etiqueta correcta) será 1 y el resto 0; es decir, un vector binario, siendo compatible con el lenguaje de las redes neuronales, pues estas no trabajan con *strings*.

También definimos una variable *BATCHSIZE* que determinará el tamaño del lote a la hora de aprender; entrenaremos a la red por lotes de 32 recortes de imágenes. Hemos pasado las listas de entrenamiento y validación al programa *Clasification.py*, [1]; para la lista de entrenamiento, hemos separado un total de 53 imágenes, consiguiendo un total de 2359 recortes de imágenes.

```
Extracted #2359 stamps
Extracted #2040 stamps of 'Circulares', #303 stamps of '
    Cometas', #16 stamps of 'Donuts'
```

Podemos ver algunas de estas a continuación, en una mezcla (*shuffle*) de recortes de entre las 2359 totales:

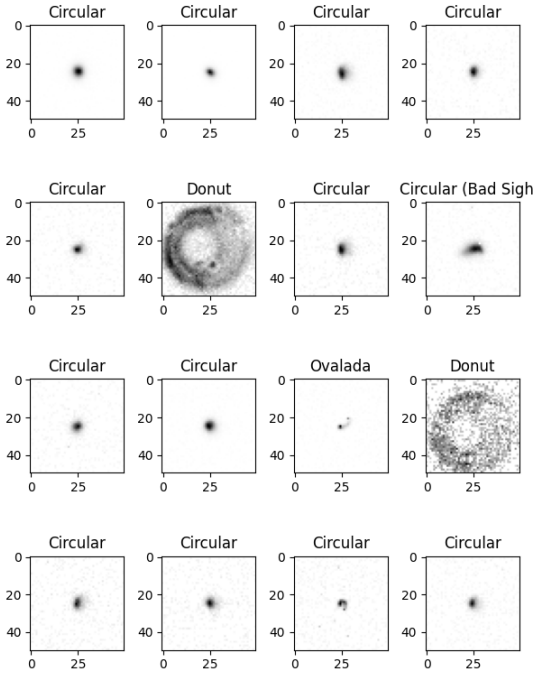



Figura 4.1: Muestra de ejemplo de recortes de imágenes astronómicas.

[1] En la forma $(N, 60, 60, 1)$: $N$ imágenes $60 \times 60$ y 1 canal (grises).

[2] Estos valores ya han sido normalizados al guardarlos en los archivos *.h5* previa lectura de la red, dividiendo entre el máximo de cada una de las cajas recortadas para cada estrella.

Respectivamente, para las listas de validación y evaluación obtuvimos:

```
Extracted #747 stamps
Extracted #548 stamps of 'Circulares', #191 stamps of '
    Cometas', #8 stamps of 'Donuts'
```

```
Extracted #552 stamps
Extracted #448 stamps of 'Circulares', #96 stamps of '
    Cometas', #8 stamps of 'Donuts'
```

haciendo una proporción entre listas de alrededor del: $\sim 80-20\,\%$.[3] Como ya hemos explicado previamente, es una buena práctica empezar con una estructura «sencilla» para no caer en *overfitting* e ir jugando con los hiper-parámetros con vistas a llegar a un resultado considerablemente bueno. No obstante, se empezó con una red tal que

```
 Layer (type)                 Output Shape              Param #
=================================================================

(Conv2D)_1                (None, 60, 60, 32)          832
(MaxPooling2D)_1          (None, 30, 30, 32)          0
(Conv2D)_2                (None, 30, 30, 64)          51264
(MaxPooling2D)_2          (None, 15, 15, 64)          0
(Conv2D)_3                (None, 15, 15, 64)          102464
(MaxPooling2D)_3           (None, 5, 5, 64)           0
(Flatten)                    (None, 1600)             0
(Dense)                       (None, 32)              51232
(Dense)                       (None, 3)               99
=================================================================

Total params: 205,891
Trainable params: 205,891
Non-trainable params: 0
```

con 205.891 parámetros a entrenar[4] con una estructura de gran cantidad de parámetros (determinada por las conexiones) obtenemos *overfitting*, con una función de coste que no termina de optimizarse, con un valor pequeño en las precisiones en el entrenamiento de las listas de validación y test, 75 % y 71.15 %, respectivamente. Ampliando los imágenes de entrenamiento (rotaciones), probando a dotar a la red de más parámetros a entrenar, obtenemos a priori «mejores resultados»:

[3] Se suele disponer de una relación $80-20\,\%$ entrenamiento-validación. En nuestro caso, sin contar con la lista de evaluación, $\sim 75-25\,\%$.

[4] Estas tablas se realizan mediante *model.summary()* en *python*.

```
  Training loss: 0.1008 -   Training accuracy: 0.9932
Validation loss: 0.9912 - Validation accuracy: 0.8016
==============================================================

Test accuracy:
 0.9690265655517578
Test loss:
 0.13090650737285614
```

*Nota.* Se cuadriplicaron los recortes de imagen tipo *Cometa* y *Donut*, rotando sobre los ejes de la imagen 90° ($[:,\ ::-1,\ ::-1]$ para rotar 90° en torno a los dos ejes de la imagen, $[:,\ :,\ ::-1]$, $[:,\ ::-1,\ :]$, para los dos ejes por separado, en *python* $\mapsto$ *numpy*):

```
Extracted #3316 stamps
Extracted #2040 stamps of 'Circulares', #1212 stamps of '
    Cometas', #64 stamps of 'Donuts'
```

Vemos que en este último ejemplo obtenemos nuevamente *overfitting*, manifestado por la diferencia entre los valores de validación y test. Como comentamos en la Sección 2.3.2, sobre la complejidad de las redes y las formas de regularizar a estas, entre estos detalles se encontraba el de aumentar los datos (para los datos de entrenamiento, específicamente, no para validación ni test). Además, como acabamos de ver, debemos reducir la complejidad de conexiones para obtener mejores resultados; en este último caso, se llevaron los parámetros a entrenar a 739.019. Otro motivo por el cual puede haber *overfitting* es el de la "mala" proporción de las categorías a entrenar; es por ello que en el aumento de datos hemos cuadriplicado tanto los recortes de imagen tipo *Cometa* como *Donut*, mejorando ínfimamente la estadística[5]; todo este proceso se realiza con el objetivo de mejorar la estadística y los resultados finales, no para el objetivo del trabajo en sí.

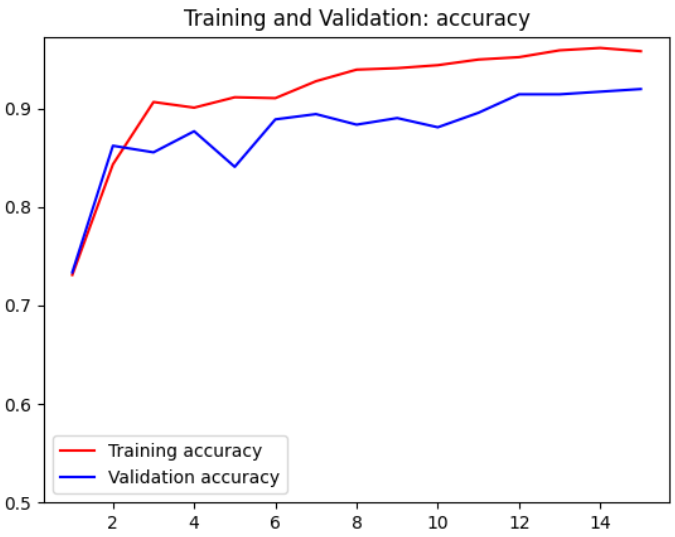
*Nota.* Vemos que, en contraposición con el dataset de *MNIST*, tenemos bastantes menos datos, por lo que se podría pensar que puede haber *underfitting* en nuestro modelo. No obstante, como hemos visto previamente (Sección 2.3.2), es por ello que acudimos a las muestras de entrenamiento y validación (53 y 15 imágenes respectivamente), para intentar evitarlo, además del sobre-ajuste overfitting.

Pasando con la estructura de la red, con vistas a mejorar los resultados anteriores, hacemos un modelo de tipo secuencial añadiendo a este 3 capas de convolución con 16, 32 y 64 filtros de tamaño $3 \times 3$ (los cuales se inicializarán de forma aleatoria para luego ir ajustándose durante el aprendizaje-entrenamiento), con sus respectivas capas *MaxPooling* para ir progresivamente reduciendo la dimensionalidad. Además, terminamos con una capa oculta densa de 8 neuronas y otra de salida

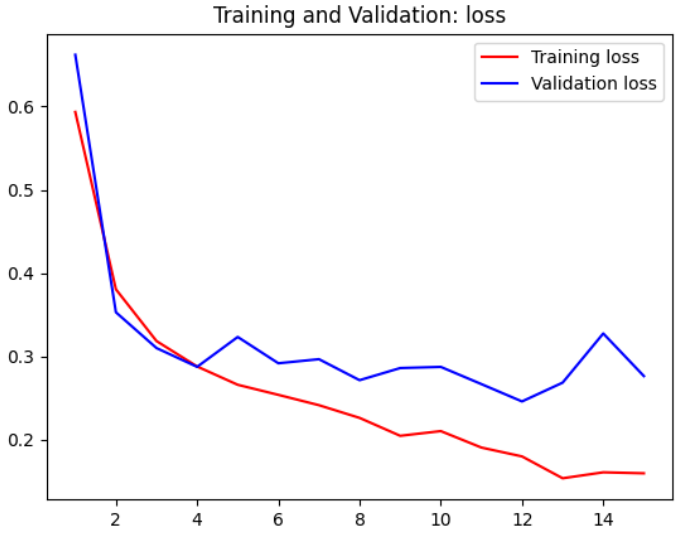
[5] Las imágenes tipo *Donut* las predice la red prácticamente con un 100 % de precisión, pues solo disponemos de 3 imágenes y en la lista de test se repitió una de estas.

con n_classes $\mapsto 3$ como *outputs*, los cuales darán como salida los valores asociados a las etiquetas de las imágenes, con *Softmax* como función de activación para esta última y $ReLU$ para las capas ocultas. En los anteriores modelos buscando el que mejor resultados proporcionara, se usó la función *Mish* (Sección 2.1.1); no obstante, $ReLU$ proporciona mejores resultados para este modelo. Además, después de reducir las dimensiones de la imagen, añadimos capas *Dropout* con probabilidades, respectivamente, de 0.25, $p = 0.30$ y $p = 0.30$, finalizando con otra capa con $p = 0.35$ en las conexiones de las capas densas. Compilamos el modelo y comenzamos el entrenamiento,

```
 Layer (type)        Output Shape              Param #
=================================================================

(Conv2D)           (None, 60, 60, 16)           160
(MaxPooling2D)     (None, 30, 30, 16)             0
(Dropout)          (None, 30, 30, 16)             0
(Conv2D)           (None, 30, 30, 32)          4640
(MaxPooling2D)     (None, 15, 15, 32)             0
(Dropout)          (None, 15, 15, 32)             0
(Conv2D)           (None, 15, 15, 64)         18496
(MaxPooling2D)      (None, 7, 7, 64)              0
(Dropout)           (None, 7, 7, 64)              0
(Flatten)             (None, 3136)                0
(Dense)                (None, 8)              25096
(Dropout)              (None, 8)                  0
(Dense)                (None, 3)                 27
=================================================================

Total params: 48,419
Trainable params: 48,419
Non-trainable params: 0
=================================================================

loss: 0.1597 - accuracy: 0.9581 - val_loss: 0.2764 -
val_accuracy: 0.9197
```




(a) Precisión en el entrenamiento y validación.




(b) Optimización de la función de coste en el entrenamiento y validación.

obteniendo un resultado del $\sim$ 96 % de precisión en el entrenamiento, así como un $\sim$ 92 % en la validación, a través de 15 épocas de entrenamiento. Guardamos el modelo para evaluarlo en otro programa *python* (*model.evaluate()*), con las 13 imágenes reservadas para esta tarea:

```
Test accuracy:
 0.89673912525177
Test loss:
 0.5639076828956604
```

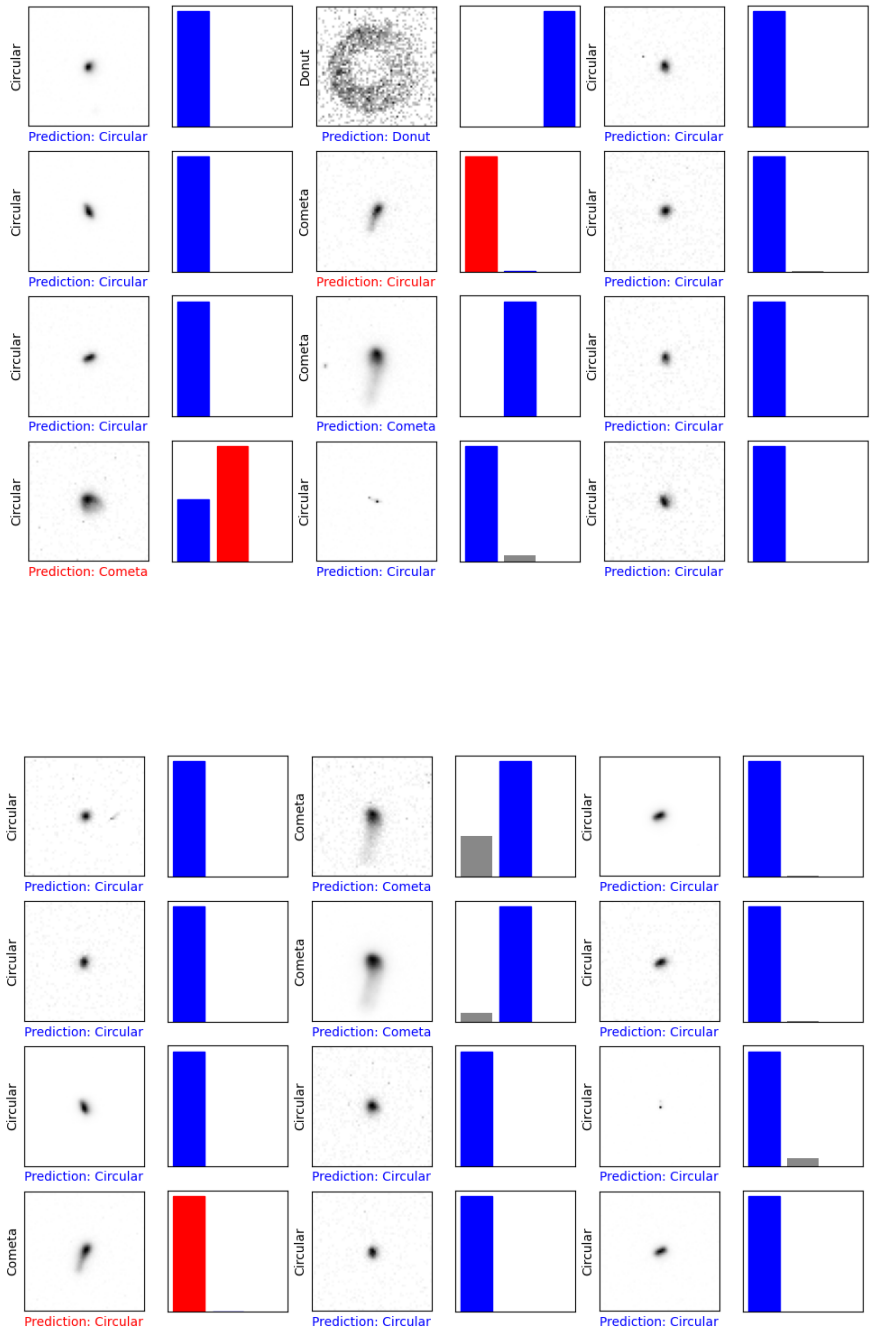



Figura 4.2: Diferentes predicciones del modelo, de entre los 552 recortes de imagen de la lista de evaluación.

Obtenemos una precisión del $\sim 89-90\,\%$, lo que resulta en que el modelo predice con gran acierto, pudiendo ser usado por tanto para clasificar. En las Figuras 4.2

observamos barras representadas en diferentes colores (3 barras para cada una de las clases $\mapsto$ *Circular*, *Cometa* y *Donut*); plasman los porcentajes de acierto o error en las predicciones: la barra lateral izquierda se asocia a la etiqueta *Circular*, la central a las imágenes tipo *Cometa* y tipo *Donut* para la última. El color azul representa el porcentaje de la etiqueta *real*, $\mathbf{y}$; el color rojo representa la predicción errónea y el gris aquella predicción que no supera a la asociada a la etiqueta real.[6]

## 4.2. Conclusiones

Partimos de un conjunto de imágenes astronómicas las cuales queríamos clasificar con vistas a determinar aquellas que resultan defectuosas; expusimos las herramientas con las que analizar estas imágenes y las clasificamos según el tipo de forma de la mancha representativa de las estrellas. Estudiamos las redes neuronales multicapa y posteriormente las convolucionales con el objetivo de clasificar automáticamente nuestras imágenes astronómicas. Por último, creamos una red con dicho propósito, obteniendo buenas predicciones del modelo propuesto. Como mejoras posibles a los resultados, podríamos usar el *Data Augmentation*[7] para entrenar a la red con mayor eficiencia. Además, estos también pueden verse afectados por el «ruido» no eliminado de las imágenes, como por ejemplo en las muestras de entrenamiento, validación y test hay recortes de imagen en los que aparecen pequeñas fuentes que no detecta el algoritmo, además de modificar los parámetros de la red en busca de mejores resultados, obtener mayor cantidad de imágenes, etcétera.

Además, pueden ocurrir a la hora de clasificar previo al entrenamiento de la red errores de observacionales, errores humanos, como puede ser estipular una imagen tipo *Cometa* en lugar de *Circular* o viceversa, entre otros. Estas pautas a mejorar se podrían implementar en un trabajo con mayor extensión y otros objetivos, además de explicar de manera más detallada cada sección de la memoria, pudiendo aportar una mayor cantidad de ejemplos.

[6] Todos estos porcentajes vienen determinados por la función de activación *softmax*.
[7] Método *ImageDataGenerator* del módulo *keras.preprocessing.image* en *python* para rotar imágenes.

# Bibliografía


[1] *GITHUB. Programas Python - TFG* [en línea]. Disponible en: https://github.com/DanielRodriguezEspinosa/PythonTFG-Redes-Neuronales

[2] Andy Thomas. *An introduction to neural networks for beginners* [en línea]. Disponible en: https://adventuresinmachinelearning.com/wp-content/uploads/2017/07/An-introduction-to-neural-networks-for-beginners.pdf

[3] Sanad, 2020. *Learn Image Classification on 3 Datasets using Convolutional Neural Networks (CNN)* [en línea]. Disponible en: https://www.analyticsvidhya.com/blog/2020/02/learn-image-classification-cnn-convolutional-neural-networks-3-datasets/

[4] Luis Velasco, 2020. *Optimizadores en redes neuronales profundas: un enfoque práctico* [en línea]. Disponible en: https://velascoluis.medium.com/optimizadores-en-redes-neuronales-profundas-un-enfoque-practico-819b39a3eb5

[5] *Fotometría con imagen directa* [en línea]. Disponible en: http://research.iac.es/galeria/jap/IRAF_notes/notas_fotometria.html

[6] Tikz. *Graphics with TikZ in LaTeX* [en línea]. Disponible en: https://tikz.net/

[7] *Qué es overfitting y underfitting y cómo solucionarlo* [en línea]. Disponible en: https://www.aprendemachinelearning.com/que-es-overfitting-y-underfitting-y-como-solucionarlo/

[8] Kiprono Elijah Koech. *Cross-Entropy Loss Function* [en línea]. Disponible en: https://towardsdatascience.com/cross-entropy-loss-function-f38c4ec8643e

[9] Jose Martínez Heras. *Regularización Lasso L1, Ridge L2 y ElasticNet* [en línea]. Disponible en: https://www.iartificial.net/regularizacion-lasso-l1-ridge-l2-y-elasticnet/

[10] Tensorflow. *Tinker With a Neural Network* [en línea]. Disponible en: https://playground.tensorflow.org/ Tensorflow.

# On the use of Machine Learning techniques to discriminate astronomical images

Sección de Matemáticas
Universidad de La Laguna

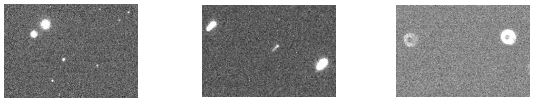
**Daniel Rodríguez Espinosa**
Facultad de Ciencias · Sección de Matemáticas
Universidad de La Laguna
alu0101051550@ull.edu.es

## Abstract

*In the present manuscript it is intended to investigate image recognition techniques through the use of neural networks to separate badly taken images, as well as blurs. Initially we introduce the basic notions of what an astronomical image is, what the shape of the stellar objects that make it up is like, the intensity profile, describing the tools to be used to visualize and automatically discriminate objects using deep learning. Next, we go on to study about neural networks, what they are and what they are made of, developing gradient descent and backpropagation algorithms through which they can learn in a hierarchical way. After some illustrative examples, convolutional neural networks are introduced, which specialize in image processing, seeing how the network can recognize patterns in a similar way to how the human eye does it, through the convolution process and image filters. Finally, the previous concepts are illustrated, showing the results of implementing a convolutional neural network that discriminates astronomical images, discussing how the difficulty of machine learning lies in the good choice of the model, concluding with its predictions.*

## 1. Introduction

Currently there are many projects in astrophysics that are based on the acquisition of many images; the treatment of these images and the extraction of information from them is done automatically. Some images show problems due to various reasons: for example, telescope tracking was not stable, sharpness/contrast (*seeing*) was poor, etc. Therefore, we introduce an image classification system that allows us to detect those that can potentially cause problems, through the use of neural networks.



( a ) *Circles* stars ( b ) *Comet* stars ( c ) *Donut* stars

**Figure 1:** *Clasification*

## 2. Artificial Neural Networks

The perceptron is the basic processing unit within neural networks; similar to a biological neuron, it has input connections from which it receives external stimuli, the input values, performs an internal calculation and generates an output value. Each of these neurons make up the neural network:

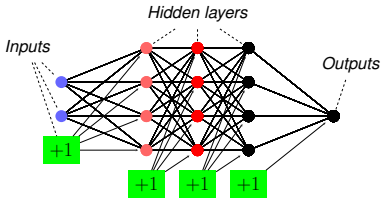



**Figure 2:** *Artificial Neural Network*

Neural networks learn automatically through the gradient descent and backpropagation algorithms:

$$Gradient\ Descent \longmapsto W^{(l)}, b^{(l)} \longleftarrow W^{(l)}, b^{(l)} - \alpha \frac{\partial}{\partial W^{(l)},\, b^{(l)}} C(W, b) \tag{1}$$

$$\left.\begin{aligned} \delta^{(L)} &= \frac{\partial C}{\partial h^{(L)}} \cdot \frac{\partial h^{(L)}}{\partial z^{(L)}} \\ \delta^{(l-1)} &= \delta^{(l)} \cdot W^{(l)} \cdot \frac{\partial h^{(l-1)}}{\partial z^{(l-1)}} : l = 0, \ldots, L \\ \frac{\partial C}{\partial b^{(l-1)}} &= \delta^{(l-1)}; \quad \frac{\partial C}{\partial w^{(l-1)}} = \delta^{(l-1)} h^{(l-2)} \end{aligned}\right\} \quad Backpropagation \tag{2}$$

## 3. Convolutional Neural Networks

The type of structure that allows us to understand images with a high level of precision are: Convolutional Neural Networks, **CNN**.

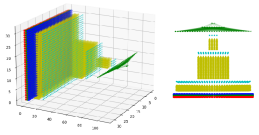


**Figure 3:** *Structure of **CNN**.*

*CNN* apply the idea of matrix convolution to images to, by means of filters or kernels, make transformations to the images, such as detecting the lines of the objects that make up the images.

## 4. Astronomical Image Discrimination

We implement a CNN to discriminate our astronomical images with an accuracy of around $\sim 90\%$.

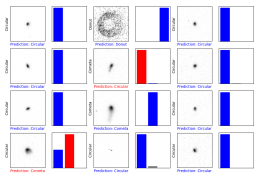


**Figure 4:** *Different predictions of the model, among the* $552$ *test image cutouts.*

## References

[1] Andy Thomas. *An introduction to neural networks for beginners.* https://adventuresinmachinelearning.com/wp-content/uploads/2017/07/An-introduction-to-neural-networks-for-beginners.pdf

[2] Sanad, 2020. *Learn Image Classification on 3 Datasets using Convolutional Neural Networks (CNN).* https://www.analyticsvidhya.com/blog/2020/02/learn-image-classification-cnn-convolutional-neural-networks-3-datasets/