Grado en Estadística

TRABAJO FIN DE GRADO

Introducción al Deep Learning. Aplicación en R

Javier Cano Ávalos

Sevilla, Junio de 2022

Índice general

Resumen

Estamos empezando a ver como algo normal utilizar el asistente de nuestro teléfono móvil, usar el traductor, buscar en google simplemente echando una foto, coches que frenan al encontrar un obstáculo delante, etc. Todas estas actividades no eran posibles hace relativamente pocos años, y es que la Inteligencia Artificial, la encargada de estas tareas, nació en el siglo pasado. Este trabajo consiste en introducir unas de las ramas de aprendizaje dentro de esta ciencia, que pese a nacer en los años 50 del siglo pasado, ya está más que presente en nuestro día a día. Se trata del Deep Learning, o Aprendizaje Profundo.

El Deep Learning, englobado dentro del Maching Learning, es una rama de la Inteligencia Artificial. Es un método de aprendizaje automático, formado por diferentes capas de aprendizaje. Su estructura se basa en las Redes Neuronales, que son métodos de aprendizaje por capas. Cada modelo de Redes Neuronales tiene una composición diferente y puede usarse para realizar unas tareas en especial.

El trabajo se inicia con una introducción al Deep Learning, y da una visión de su historia y sus precedentes. Seguidamente se introduce cómo funciona, que es a través de las Redes Neuronales. En este aspecto, se centra en las Redes Neuronales Convolucionales. Por último, se lleva a cabo dos aplicaciones en el lenguaje R donde se intentará clasificar unas imágenes; primero en blanco y negro, y luego a color, utilizando la librería *Keras*.

Abstract

We are getting used to using our smartphone asistance, the automatic translator, searching in Google just by taking a photo, seeing cars which are able to stop if case some obstacle is on the road, etc. All this activities were not possible a few years ago and the reason is that Artificial Intelligence, which is responsible for this tasks, were born in the last century. This project consists in introducing one field from inside this science, which in spite of beeing born in the 50s, it is already present daily in our lifes. We are talking about Deep Learning.

Deep Learning, which is included inside Maching Learning, it is a subcategory of Artificial Intelligence. It is a self learning method, form by diferent learning layers. It is Neural Networks the structures which Deep Learning models are made of. Each Neural Network model has a diferent composition and can be used in some specific tasks.

This project starts with an introduction to Deep Learning, its history and its precedents. Right after it is introduced how Deep Learning works, what is through Neural Networks. About this topic, we will focus on Convolutional Neural Networks (CNN). Finally, two applications are done in R lenguage where we will try to clasify some images, first in black and white, and then in color, all using the keras library.

Índice de figuras

Índice de tablas

Capítulo 1

Introducción al Deep Learning

1.1. Inteligencia Artificial

La Inteligencia Artificial (IA) nació en los años 50 a raíz de que sus pioneros, pertenecientes al campo de las ciencias de la computación, empezaron a preguntarse si era posible hacer que los ordenadores "piensen". Una forma de definir la IA es: el esfuerzo por automatizar tareas inteligentes llevadas a cabo normalmente por humanos. La IA engloba muchos campos, entre ellos el Machine Learning y el Deep Learning.

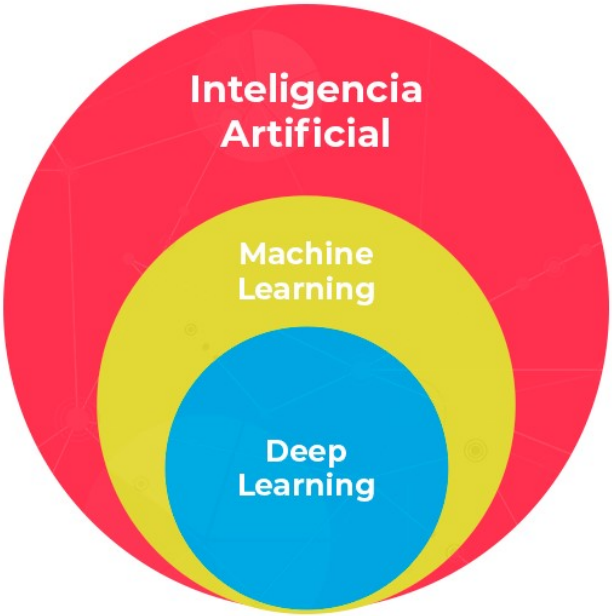


Figura 1.1: Comparación IA-ML-DL

Durante mucho tiempo, los expertos creyeron que la IA a nivel humano se podría alcanzar haciendo que los programadores crearan a mano un conjunto considerable de reglas específicas para manipular el conocimiento. Esta estrategia se conoce como IA simbólica, y dominó en el campo de la IA hasta los años 80, cuando alcanzó su pico de popularidad durante el *boom* de los Sistemas Expertos.

Pese a demostrar ser eficaz ante problemas lógicos y bien definidos, resultó ser inexplicable ante problemas más complejos y confusos como reconocimiento de voz, clasificación de imágenes o traducciones de idiomas.

1.2. Machine Learning

En el siglo XIX Ada Lovelace dijo acerca de su Máquina Analítica que podría hacer todo lo que supiéramos cómo ordenarle que hiciera. Más adelante Alan Turing citaba a Ada mientras reflexionaba sobre si los ordenadores podrían aprender y ser originales, y concluyó afirmando que podían.

¿Puede un ordenador ir más allá de los que "sabemos ordenarle que haga" y aprender por sí solo a realizar una tarea específica? En vez de recibir reglas dadas por programadores, ¿puede un ordenador aprender automáticamente estas reglas simplemente mirando los datos? ¿Puede un ordenador sorprendernos? De estas preguntas es de donde nace el Machine Learning (ML), allá por los años 90 y convirtiéndose rápidamente en el campo de la IA más popular. Esto lleva a un nuevo paradigma de programación. Vimos que en la IA simbólica, a partir de los datos y las reglas dadas, se obtenían las respuestas. Ahora en el ML se busca que dados unos datos y unas respuestas esperadas, se obtengan dichas reglas. Diremos entonces que un sistema de ML no se programa, se entrena, para que viendo muchos ejemplos de una tarea concreta encuentre una estructura estadística que le permita dar con las reglas necesarias.

El problema central del ML es transformar (operar) los datos significativamente, es decir, aprender una representación útil de ellos, de manera que nos acerquen a la salida esperada. Estas operaciones pueden ser cambios de coordenadas, poyecciones lineales, operaciones no lineales, traducciones, etc.

1.3. Deep Learning

El Deep Learning (DL), el tema de este trabajo, es un campo dentro del ML. El concepto "deep" simplemente significa que consta de varias capas sucesivas de aprendizaje. Al número de capas que forman parte del modelo se les denomina "depth" (profundidad). Dicha profundidad puede llegar a ser, sobre todo en modelos de DL modernos, de incluso cientos de capas. Otros métodos de ML se centran solo en el estudio de 1 o 2 capas, y se les llama Shallow Learning (aprendizaje poco profundo).

En DL, estas capas se aprenden a través de modelos llamados Redes Neuronales, estructuradas en capas apiladas unas encima de otras. Se podría decir que consisten en un proceso de depurar información en múltiples etapas, en los que la información pasa por sucesivos filtros para salir cada vez más depurada.

1.4. Diferencias entre el Machine Learning y el Deep Learning

Acaba de ser introducido el ML, o aprendizaje automático, que consiste en que un ordenador pueda resolver un problema sin que lo tenga programado explícitamente. También el DL, que hace que el aprendizaje sea mediante profundas redes neuronales. Pero esto nos lleva a una pregunta, ¿cuáles son realmente las diferencias entre el ML y el DL?

Tenemos 2 diferencias claves entre ambos:

- La primera está relacionada con la forma en la que el ordenador aprende a resolver un problema concreto. El ML requiere de unos datos iniciales (datos de entrenamiento) y de una extracción de características, es decir, necesita que se le especifique algo de información sobre las características de esos datos. Sin embargo en el DL no es necesario facilitarle esto último, con los datos iniciales es suficiente, ya que su procedimiento de aprendizaje consiste en deconstruir los datos en múltiples niveles de detalle, y en caso de encontrar algún patrón que se repita mucho el programa lo etiquetará como una característica importante.
- La segunda se refiere a las limitaciones de pueden tener. El DL requiere un conjunto de datos iniciales mucho más extenso que el ML y, por tanto, también una potencia de cómputo mayor.

Vemos un ejemplo:

> Suponemos que tenemos unas imágenes de animales y queremos un algoritmo que nos clasifique dichas imágenes en 2 grupos, es un gato o no es un gato. Si aplicamos ML, tendremos que darle al programa las imágenes de los animales y que alguna vaya etiquetada como "gato". Si por el contrario se le aplica DL, no será necesario explicarle al programa cómo son los gatos, simplemente darle un número suficiente de imágenes de gatos y será capaz de resolverlo por sí mismo.

1.5. Aplicaciones del Deep Learning

Pese a que el DL es un subcampo del ML bastante antiguo, no ha sido hasta 2010 cuando empezó a ganar protagonismo. Pero no por haberse alzado hace relativamente pocos años significa que no se hayan logrado muchos avances en este campo, en especial en problemas de percepción como ver y oír.

En particular, el DL se ha aplicado, y con éxito, a:

- **Clasificación de imágenes**: a nivel casi humano.
- **Reconocimiento de voz**: a nivel casi humano.
- **Transcripción de textos escritos a mano**: a nivel casi humano.
- **Text to Speech**: traducir un texto escrito a palabras habladas.
- **Búsqueda de respuestas en lenguaje natural**.
- **Asistencia digital**: como Google Now y Alexa
- **Mejorar resultados de búsqueda en la web**.
- **Publicidad dirigida**.
- **Conducción autónoma**: a nivel casi humano.
- **Ajedrez y GO**.

A pesar de todo esto, aún no se sabe qué más puede hacer el DL.

1.6. Promesa del Deep Learning

Las expectativas son muy altas. Normal si observamos, como acabamos de ver, que en cuestión de muy poco tiempo, solo una década, el DL ha alcanzado muchos logros importantes. Algunos de ellos incluso han cambiado la forma de ver el mundo, como que haya coches que se conduzcan solos. Pero hay mucho aún por lograr y no va a ser a corto plazo. Hablamos de inteligencia general a nivel humano: traducciones, diálogos, entender el lenguaje natural... Pensar que podemos lograrlo a corto plazo puede ser muy peligroso, la tecnología falla y los plazos pueden alargarse y la investigación congelarse.

Aún así, que estas metas se van a cumplir es un pensamiento realista. Acabamos de empezar a aplicar DL a muchos problemas importantes como diagnósticos médicos y asistencia digital, aunque estos avances aún no se utilizan. Por supuesto que ya podemos hacerle preguntas a nuestro teléfono móvil y recibir respuestas razonables, obtener recomendaciones de Amazon, etc. Pero hay que tener en cuenta que la IA está destinada a cambiar nuestra forma de trabajar, vivir y pensar.

No es ninguna exageración decir que la IA se va a convertir en tu asistente, o incluso en tu "amigo", gracias a los futuros avances del DL. Responderá a tus preguntas, ayudará a educar a tus hijos, cuidar tu salud, traerte la compra a casa, llevarte a los sitios, y lo más importante, ayudará a los humanos a avanzar en todos los campos científicos.

En el camino puede que haya parones, "inviernos" para la IA, pero al final, más tarde o más temprano, todo lo nombrado antes llegará. No hay que quedarse en las expectativas a corto plazo, pero podemos creer en la visión a largo plazo.

1.7. ¿Cómo funciona el Deep Learning?

Por último, sabiendo ya cómo ha empezado, qué ha logrado y qué se espera en un futuro del DL, solo queda un punto por ver. ¿Cómo se hace DL?

El DL es una nueva forma de aprendizaje automático que se basa en aprender a través de sucesivas capas de representación. El número de niveles de capas que contenga el modelo es lo que denominaremos profundidad. Podemos deducir que otros nombres apropiados para este campo podrían ser por ejemplo "representación por capas" o "aprendizaje jerárquico". Dichas capas se aprenden de manera automática al exponerlas ante el conjunto de datos de entrenamiento.

Estas capas se aprenden a través de modelos llamados Redes Neuronales (Neural Networks) y se estructuran apilándolas unas encima de otras. Estas redes están, en cierto modo, inspiradas en los conocimientos que tenemos del cerebro, aunque no funcionan de igual forma. No hay relación entre el DL y la Biología.

Veamos un ejemplo de cómo se comporta un modelo sencillo de DL y sus capas:

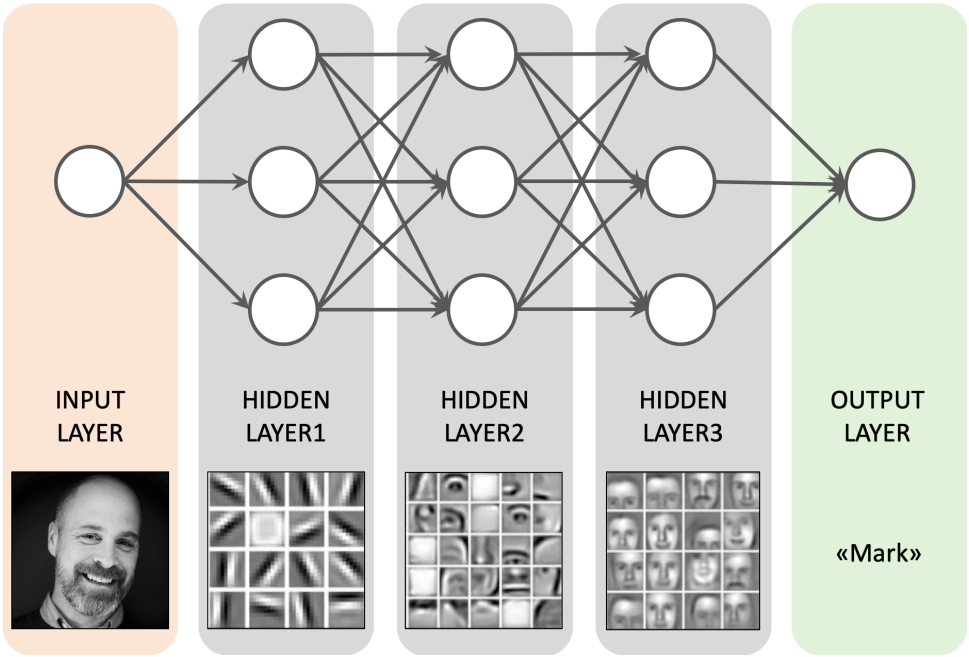


Figura 1.2: Ejemplo capas DL

Como se observa en la Figura 1.2, la red transforma la imagen inicial en representaciones totalmente diferentes a la original de las que obtendrá la información para dar el resultado final. Así, se puede concluir diciendo que una red profunda es una operación de destilación de información en múltiples etapas, donde esta información se "purifica" a través de sucesivos filtros.

Capítulo 2

Redes Neuronales

A finales del siglo XIX, el Premio Nobel de Medicina español Santiago Ramón y Cajal situó por primera vez las neuronas como elementos funcionales del sistema nervioso. Propuso que las neuronas actúan como entidades discretas que mediante sus conexiones e interacciones forman una red y dan lugar a las respuestas complejas del cuerpo humano. Podemos decir que desde un punto de vista biológico, las neuronas son células capaces de generar señales eléctricas que transmiten a otras partes del cuerpo para realizar una determinada función (mover un músculo, reaccionar ante un estímulo externo, etc).

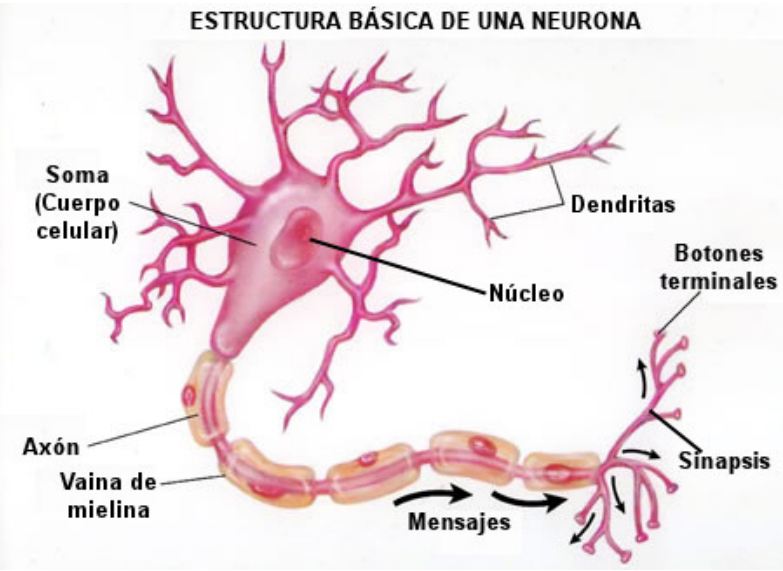


Figura 2.1: Neurona

En el cuerpo de la neurona se encuentra el núcleo, que es el que se encarga, entre otras funciones, de recibir información de otras neuronas a través de conexiones sinápticas. Las dentritas es por donde se recibe la información, conexiones de entrada. Los axones son los encargados de la salida de información de la neurona, son los que envían impulsos a otras neuronas. La capacidad de una neurona de generar señales eléctricas se debe a que posee una membrana semipermeable que es selectiva al paso de iones a través de ella, y separa dos medios acuosos con diferentes concentraciones de iones.

Una red neuronal está compuesta por diferentes tipos de neuronas. Las neuronas sensoriales (de entrada), conectadas a las neuronas de proyección e interneuronas (serían las ocultas) y, éstas a su vez, conectadas a las neuronas motoras (de salida) que son las que controlan por ejemplo los músculos. En el cerebro, las más abundantes son las neuronas de proyección y las interneuronas, que forman una gigantesta red de neuronas "ocultas" y son las que realizan los "cálculos".

2.1. Redes Neuronales Artificiales.

Una Red de Neurona Artificial (RNA) es un modelo matemático inspirado en el funcionamiento y las interconexiones del cerebro humano. Se basa en la red de neuronas biológica y utiliza funciones matemáticas para simular su comportamiento.

Podríamos decir que las neuronas se agrupan dando lugar a las capas que dijimos formaban las redes neuronales. En DL sabemos que existen 3 tipos de capas:

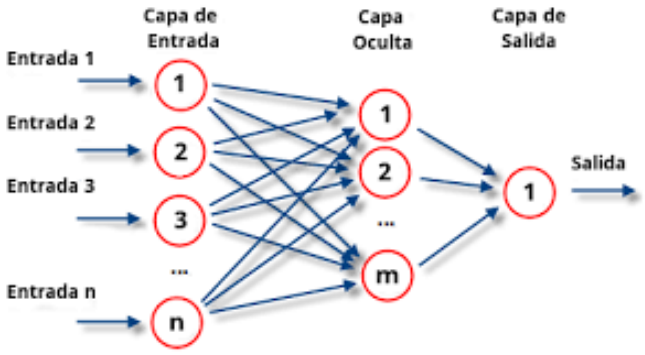


Figura 2.2: Capas de RNA

- La capa de entrada, el input, donde entran los datos.
- La capa de salida, el output, de la que se extrae el resultado final.
- Las capas ocultas, hidden, en las que se desglosa y se procesa la información.

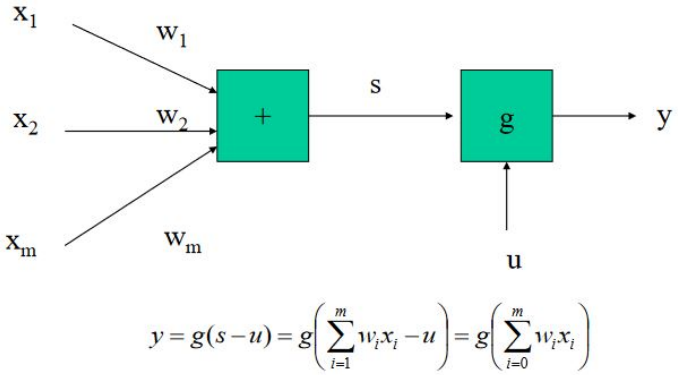


Figura 2.3: RNA

En una RNA encontramos los siguientes elementos:

- Conjunto de datos de entrada $X = (x_1, ..., x_m)^t$, siendo m el número de entradas.
- Enlaces sinápticos, que son las flechas de entrada y salida de cada neurona, y sus respectivos pesos $W = (w_1, ..., w_m)^t$.
- Un sesgo o umbral u.
- La regla de propagación, que determina el potencial resultante de la interacción de cada neurona con las salidas anteriores, ya sea de otras neuronas o los datos de entrada. Resulta de la suma de las entradas ponderadas con sus pesos, siendo $s = \sum_{i=1}^{m} w_i x_i = W^t X$
- Función de activación $g(s-u)$ que es función de la regla de propagación y del sesgo.

Hay diferentes tipos de redes neuronales, y estas se clasifican atendiendo a varias características:

- Según la arquitectura de la red y el número de capas tendremos redes monocapas o multicapas.
- Según las características de los nodos podemos encontrar redes unidireccionales (feedforward) o recurrentes (feedback).
- Según el método de aprendizaje:
  - Aprendizaje supervisado: si se dispone de los datos de entrada y de la salida correcta, lo que hace que no sea de naturaleza flexible; se usa para clasificación, aproximación, identificación, etc.
  - Aprendizaje no supervisado: no se dispone de la salida esperada, solo de los datos de entrada, luego sí es un modelo flexible; se usa para agrupar, extracción de características, análisis de datos. Se trata del pan de cada día en análisis de datos, incluso puede ser necesario aplicarlo antes de intentar resolver un problema por aprendizaje supervisado, simplemente para entender mejor el conjunto de datos.
  - Aprendizaje autosupervisado: es un caso específico del aprendizaje supervisado, pero suficientemente diferente como para ser una categoría aparte. Se trata de un aprendizaje supervisado sin la intervención humana. Hay ciertas indicaciones a seguir aun así, porque de alguna forma tiene que estar supervisado, pero están generadas por los datos de entrada, normalmente usando algoritmos heurísticos.
  - Aprendizaje de refuerzo: es un algoritmo que especifica cómo debe aprender a seleccionar acciones para maximizar el éxito esperado (es un tipo de aprendizaje supervisado) y además, también es de naturaleza flexible y se entiende su entorno como un profesor; suele usarse en inteligencia artificial.

Sabiendo todo esto, vamos a empezar analizando las redes neuronales más simples y comunes que podemos encontrar, cuyas aplicaciones giran en torno a la regresión y la clasificación.

2.2. Perceptrón Simple

Es la Red Neuronal más simple (SLP) y para muchos, la que marcó el comienzo de la IA e inspiró el desarrollo de otras redes neuronales. Posee una arquitectura bastante simple, como se observa en la figura:

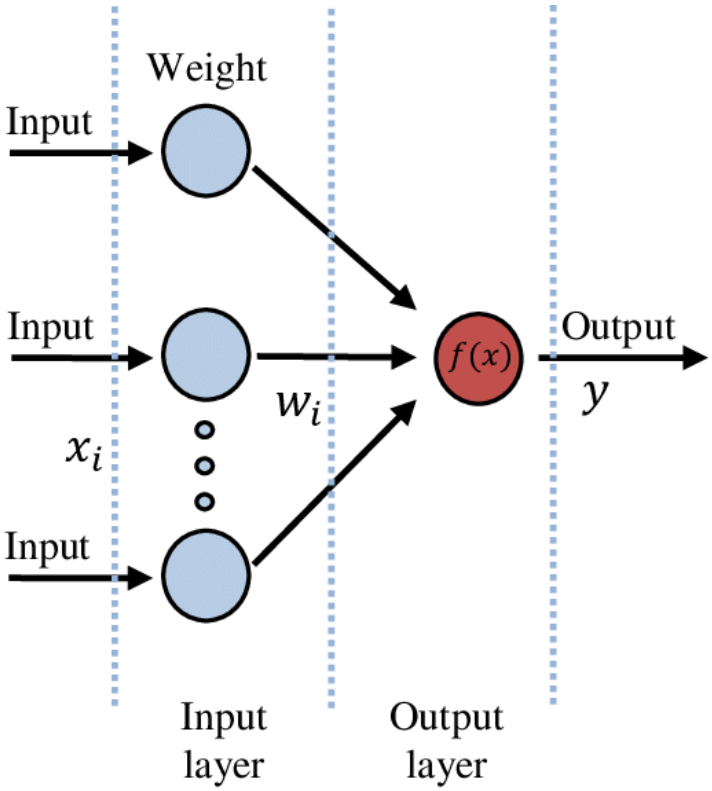


Figura 2.4: SLP

Donde:

$$Output = \begin{cases} 0 & \text{si } \sum \omega_i x_i > \theta \\ 1 & \text{c.c.} \end{cases}$$

- Está formada únicamente por 2 capas.
  - La primera de ellas no realiza ningún cálculo, simplemente proporciona los valores de entrada x_i a la red .
  - La segunda está formada por un solo nodo, o una sola neurona, que recoge los datos de entrada mediante conexiones sinápticas, cada una de ellas con un peso ω_i, y que aplica una función de activación.
- Es una Red Neuronal unidireccional.

El modelo SLP requiere de un entrenamiento para el que existen diferentes métodos. La función de la salida (output) del modelo SLP es $y = f(X, W^t)$ y el objetivo de entrenar el modelo será encontrar los pesos que minimizan la función del error. El algoritmo detrás del modelo SLP consiste en lo siguiente:

- Paso 1: Inicializar los pesos y un umbral mediante valores aleatorios.
- Paso 2: Sumar los pesos y si la suma da por encima del umbral, se activa la red.
- Paso 3: Si el valor calculado coincide con el de la salida esperada, el modelo es bueno.
- Paso 4: Si no se cumple el Paso 3, hay que recalcular los pesos usando la fórmula $\Delta\omega = \eta * d * x$, donde x son los datos de entrada, d la diferencia entre la salida calculada en el Paso 3 y la salida esperada y η el ratio de aprendizaje, normalmente menor que 1; y volver al Paso 2

La salida que se obtiene viene de ponderar los datos de entrada por los pesos. Suele ser 1 y 0, aunque también hay veces que podemos encontrarnos esta salida como -1 y 1, dependiendo de la función de activación usada, aunque siempre booleanos.

La principal limitación de los modelos SLP, que es la que ha dado lugar a posteriores modelos de redes neuronales, es que solo es acertado cuando tenemos datos claramente diferenciables en dos grupos. Esto obviamente se complica cuando tenemos un conjunto de datos más complejo y denso.

2.3. Perceptrón Multicapa

El modelo de Perceptrón Multicapa (MLP) se diferencia del modelo SLP por el hecho de que encontramos capas ocultas que intervienen en la salida del modelo, lo vemos en la figura siguiente:

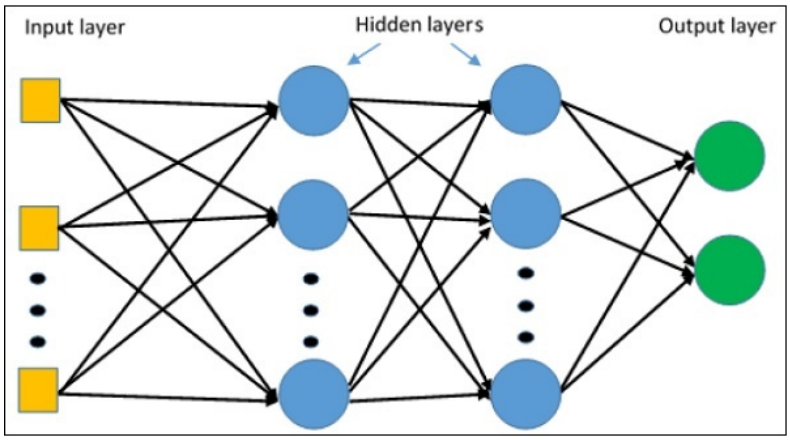


Figura 2.5: MLP

También, al introducir una capa oculta, el modelo podrá representar una función booleana, y al introducir dos o más, será capaz de representar todo un espacio de decision arbitrario. Cada neurona se conecta mediente un enlace sináptico con su correspondiente peso, de forma parecida al modelo SLP.

Ahora una de las preguntas que surgen es, ¿qué arquitectura pueden tener los modelos MLP y cómo afecta a su comportamiento?. Estos modelos son no lineales, luego encontrar una solución óptima no es tarea sencilla. El algoritmo estandar usado para entrenar los modelos MLP es el algoritmo back-propagation. Fue el primer método práctico utilizado para su entrenamiento, que además usa el descenso de gradiente. El algoritmo back-propagation consiste en:

- Paso 1: Inicializar los pesos mediente valores aleatorios obtenidos de una distribución Normal.
- Paso 2: Introducir los datos de entradas y que pasen a través de las capas ocultas hasta la capa de salida. La activación de α_j^l de la j-ésima neurona de la l-ésima capa viene dada por la función

$$\alpha_j^l = \sigma(\sum_k \omega_{jk}^l \alpha_k^{l-1} + b_j^l),$$

  donde la suma se basa en todas las k neuronas de la (l-1)-ésima capa
- Paso 3: Calcular el gradiente y, acorde a ello, actualizar el valor de los pesos.
- Paso 4: Repetir los pasos 2 y 3 hasta que el algoritmo converja en un intervalo de error tolerable o se alcance el máximo de interacciones posibles.

Al aplicar este algoritmo, debido a que el error es función de los pesos, hay situaciones en las que es más difícil lograr la convergencia a un punto óptimo global. Al tratar de optimizar funciones no lineales, hay mínimos locales que pueden tapar al mínimo global. Debido a esto, podemos llegar a pensar que hemos encontrado un modelo que soluciona el problema cuando, en realidad, nuestra solución no es la más efectiva.

Por último solo falta resolver la pregunta de cuántas capas usar y qué número de neuronas forman cada capa. Respecto a las capas, normalmente se recomienda usar 2 capas, dado que la diferencia de efectividad de usar 2 a más de 2 es mínima. Respecto al número de neuronas, no hay un número exacto y se deberá analizar qué número minimiza el error de entrenamiento. Aun así, hay quien recomienda no usar más del doble del tamaño de la entrada, o simplemente la fórmula:

$$hidden\ layers = (input + output) * 2/3$$

2.4. Otros Modelos de Redes Neuronales

2.4.1. Redes Neuronales Convolucionales (CNN)

Las Redes Neuronales Convolucionales están diseñadas de forma que imitan el córtex visual de los animales, que es la parte del cerebro encargada de decodificar la percepción

y convertirla en visión. En los modelos CNNs las neuronas se encuentran ordenadas en tres dimensiones: anchura, altura y profundidad. Dada una capa, sus neuronas solo están conectadas a una pequeña región de la capa anterior. Estos modelos se usan normalmente para el procesamiento de imagen y la visión artificial.

2.4.2. Redes Neuronales Recurrentes (RNN)

Las Redes Neuronales Recurrentes son modelos de RNA donde sus enlaces entre unidades forman círculos dirigidos. Un círculo dirigido es una secuencia donde el paso entre vértices y aristas está determinado por el conjunto de estas últimas y, por tanto, parece seguir un orden específico. Estos modelos suelen utilizarse para reconocimiento de voz y transcripciones a mano.

2.4.3. Máquinas de Boltzmann Restringidas (RBM)

Las Máquinas de Boltzmann Restringidas son un tipo de modelo de Markov binario, con una arquitectura única, de tal modo que hay múltiples capas de variables aleatorias ocultas y una red de unidades binarias estocásticas dispuestas dos a dos simétricamente. Las RBM constan de unidades visibles y una serie de capas ocultas. Sin embargo, no hay enlaces entre unidades de la misma capa. Las RBM pueden aprender representaciones internas complejas y abstractas en tareas referidas al reconocimiento de voz y de objetos.

2.4.4. Redes de Creencias Profundas (DBN)

Las Redes de Creencias Profundas son parecidas a las RBM excepto porque cada capa oculta de cada subred es en realidad la capa visible de la siguiente. En general, son un modelo gráfico generativo compuesto de múltiples capas de variables no observables, con enlaces entre las capas en vez de entre las unidades de cada capa. Estas redes se usan para el reconocimiento y generación de imagen, en secuencias de vídeo y para la captura de movimiento.

Capítulo 3

Redes Neuronales Convolucionales (CNN)

3.1. Introducción

Las CNNs ya fueron introducidas en el capítulo anterior. El desarrollo de estos modelos nos lleva hasta los años 50, donde los científicos Hubel y Wiesel modelaron el córtex visual animal. Identificaron células más simples y complejas. Las células simples maximizaban la salida con respecto a los bordes consecutivos. Por el contrario, el campo receptivo en células complejas era considerablemente mayor, y sus salidas no estaban afectadas por la posición de los bordes dentro de dicho campo.

Recordemos que son modelos formados por neuronas ordenadas en 3 dimensiones, cada una conectada solo a una pequeña región de la capa anterior. Estos modelos se usan en diferentes tareas, aunque las más habituales son la visión artificial y el procesamiento de imagen (será esta última en la que estará enfocada la parte práctica de este trabajo), aunque también se pueden usar para el procesamiento natural del lenguaje y el aprendizaje reforzado.

3.2. Convolución

Las CNNs explotan la correlación espacialmente local forzando la conectividad local entre neuronas de capas adyacentes. Esto implica que el *input* de una unidad oculta en la capa l proviene de un subconjunto de unidades de la capa $l-1$ que tiene un campo receptivo espacialmente contiguo. Cada unidad no responde a las variaciones fuera de su campo receptivo con respecto a su *input*, lo cual hace que los filtros aprendidos resulten en una respuesta más fuerte a un patrón de entrada espacialmente local.

La principal diferencia entre modelos de redes neuronales con capas densamente conectadas y estos con capas convolucionales, si hablamos del procesamiento de imágenes, es cómo se analiza cada imagen. Los modelos de capas densas aprenden observando la imagen de forma global, analizando todos los píxeles a la vez, mientras que los modelos CNN aprenden de observar patrones locales, pequeñas partes de la imagen. Otra diferencia es

la velocidad de entrenamiento, es que esta arquitectura de red es mucho más rápida de entrenar.

La operación de convolución es el bloque fundamental a la hora de construir una CNN. De momento trabajaremos con escala de grises, por eso nuestros datos de entrada estarán expresados solo en 2 dimensiones. Vamos a ver un ejemplo donde un filtro de 3 x 3 es aplicado a una matriz 6 x 6 mediante producto escalar, y se obtiene una salida 4 x 4:

$$\begin{pmatrix} 1 & 3 & 5 & 2 & 6 & 1 \\ 7 & 0 & 2 & 4 & 1 & 3 \\ 5 & 1 & 3 & 6 & 2 & 7 \\ 2 & 4 & 3 & 0 & 0 & 1 \\ 1 & 7 & 3 & 9 & 2 & 1 \\ 3 & 6 & 2 & 4 & 1 & 0 \end{pmatrix} * \begin{pmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{pmatrix} = \begin{pmatrix} 3 & -8 & 1 & 1 \\ 6 & -5 & 5 & -1 \\ -1 & -3 & 5 & 6 \\ -2 & 4 & 4 & 11 \end{pmatrix}$$

Para obtener esta salida tendremos que multiplicar cada término del filtro por cada valor de la matriz de entrada, en cada lugar posible. Obtenemos por ejemplo el -8 del lugar $(1, 2)$:

$$3 * 1 + 0 * 1 + 1 * 1 + 5 * 0 + 2 * 0 + 3 * 0 + 2 * (-1) + 4 * (-1) + 6 * (-1) = -8$$

Si nuestra imagen es n x n y el filtro que aplicamos es f x f, el resultado tendrá dimensión $n - f + 1$ x $n - f + 1$, pero puede variar. Normalmente, f será impar, porque así siempre tendrá una posición central.

3.2.1. Detectar líneas

Existen diferentes filtros que podemos aplicar; dependiendo de cuál usemos, obtendremos una información específica. Si nos centramos en el filtro usado en el ejemplo anterior, este nos ayudará a detectar líneas verticales en nuestros datos de entrada. Otra alternativa para obtener líneas verticales sería por ejemplo el operador Sobel y el Scharr. Veamos un ejemplo práctico:

$$\begin{pmatrix} 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \\ 10 & 10 & 10 & 0 & 0 & 0 \end{pmatrix} * \begin{pmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{pmatrix} = \begin{pmatrix} 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \\ 0 & 30 & 30 & 0 \end{pmatrix}$$

Lo vemos gráficamente:

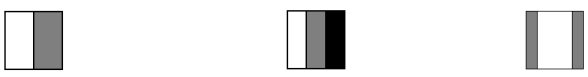


Figura 3.1: Línea vertical

Observamos que en nuestros datos de entrada tenemos media imagen blanca y media gris, y en el centro, la línea vertical que los separa. Esta entrada la convolucionamos con el filtro de la escala grises vertical y obtenemos el resultado. Este es una imagen gris a los lados y con una franja blanca en medio; esta franja blanca indica que ha detectado una línea o marca vertical en los datos de entrada en esa zona, en medio.

Un ejemplo de filtro para obtener líneas horizontales es:

$$Filtro = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{pmatrix}$$

3.2.2. Padding

Al realizar la convolución tal y como hemos visto, existen 2 problemas fundamentales. Primero, obtenemos una salida reducida, y segundo, estamos ignorando la información de los bordes. Esto es posible solucionarlo aplicando *padding* a la imagen de entrada, lo cual consiste en añadir un borde extra de ceros alrededor de la imagen. Si tenemos una imagen de 6 x 6 y le añadimos un *padding* de $p = 1$, obtenemos una nueva imagen de 8 x 8 con todo el borde compuesto por pixeles de valor 0.

Hay 2 tipos de *padding* que suelen usarse:

- Valid: no hacer *padding*.
- Same: añadir tantos bordes de ceros como sea necesario para que la salida tenga las mismas dimensiones que la entrada.

Las dimensiones de la matriz de salida es $n + 2p - f + 1$ x $n + 2p - f + 1$.

3.2.3. Strided

Al aplicar el filtro, podemos movernos en todos los lugares posibles o avanzar dejando uno o dos o los que sea en medio sin utilizar. Este "salto" es a lo que nos referimos con la palabla *strided*. Llamaremos s al salto a realizar. Las dimensiones de la matriz de salida ahora serán: $\left(\frac{n+2p-f}{s} + 1, \frac{n+2p-f}{s} + 1\right)$.

3.2.4. Volumen

Una vez pasamos a 3 dimensiones, estamos trabajando con colores, donde normalmente apilaremos 3 canales: r, g, b (rojo, verde y azul, respectivamente). Para detectar líneas y otras marcas en este tipo de imágenes, necesitaremos un filtro que sea de dimensiones 3 x 3 x 3.

La imagen vendrá definida por su altura, anchura y número de canales. Lo mismo pasa con el filtro. Sin embargo, la salida solo tendrá un canal (sin color). La razón es que el filtro tendrá el mismo número de canales que la imagen de entrada.

Si queremos detectar por ejemplo las líneas verticales del canal rojo, los demás canales, el azul y el verde, del filtro deberán ser matrices de ceros. Si queremos identificar las líneas verticales independientemente del color, los 3 canales del filtro deberán tener los mismos valores (los correspondientes a las líneas verticales, por supuesto).

Sabiendo todo esto, la operación de convolución en 3 dimensiones se hace exactamente igual que en 2D, aplicando el filtro a la imagen de entrada. El canal rojo del filtro se convoluciona con el canal rojo de la imagen de entrada, el verde con el verde y el azul con el azul. Por último, los 3 resultados se suman para obtener uno solo, que será la salida.

Para identificar diferentes marcas que haya en la imagen, se puede considerar utilizar múltiples filtros. Por ejemplo, un filtro para detectar las líneas verticales rojas, donde los otros canales, verde y azul, serán ceros; y otro para detectar las líneas horizontales, con todos los canales iguales, dado que no importa el color.

3.3. Componentes

3.3.1. Capa convolucional

En esta capa es donde se producen la mayoría de los cálculos y además, es la primera por la que pasan los datos de entrada. En ella se encuentran los filtros que examinan las porciones de la imagen. Cada filtro no es particularmente grande en cuanto a su altura y anchura, pero siempre tendrá el mismo número de canales que la imagen.

Tras aplicar la capa convolucional, calculamos el tamaño espacial:

$$Spatial\,Size_{Output} = \frac{W - F + 2P}{S + 1}$$

donde W = tamaño del volumen de los datos de entrada, F = tamaño del filtro de la capa convolucional, P = cantidad de ceros en el *padding* y S = salto.

3.3.2. Capa de Agrupación (*Pooling*)

Esta capa es la encargada de reducir la dimensión de los datos, lo cual reduce la computación y hace los detectores de rasgos más invariantes respecto de su posición en la imagen. A esta operación también se le puede llamar *downsampling* (submuestreo). Puede ir entre capas convolucionales. Si la respuesta no varía debido a la varianza del *input*, se le denomina "traslación invariante". Hay 2 tipos de capas de *pooling*:

- Capa de *pooling* máxima: desliza el filtro sobre los datos de entrada y almacena el máximo número de valores de estos en la salida, solapando la imagen inicial.

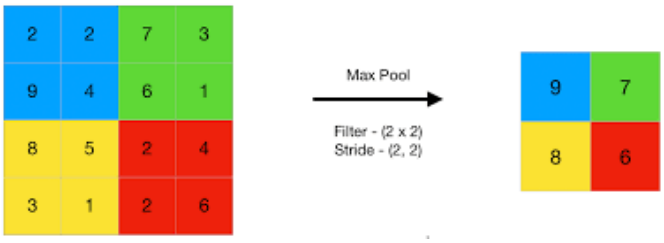


Figura 3.2: Max-Pooling

- Capa de *pooling* media: desliza el filtro sobre los datos de entrada y almacena valor medio de estos en la salida, solapando la imagen inicial.

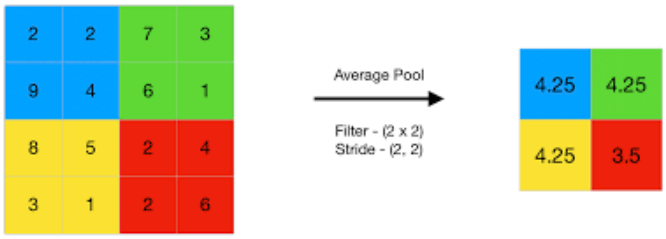


Figura 3.3: Average-Pooling

Normalmente el *max-pooling* se usa más a menudo que el *average-pooling*, aunque si el objetivo es plegar nuestra salida, usaremos el segundo. Además, al usar *max-pooling*, es importante resaltar que no se aplicará *padding*, luego las dimensiones de la salida serán $\frac{n_H - f}{s} + 1$ x $\frac{n_W - f}{s} + 1$ x n_C.

3.3.3. Capa de Unidades Lineales Corregidas (ReLU)

Aplicamos la función $f(x) = max(0, x)$ donde x es la entrada de una neurona. Al aplicarla, los valores que fueran negativos, pasarían a ser cero. Esto ayuda a delinear el mapa característico que más se acerque a la imagen.

3.3.4. Capa Completamente Conectada (FC)

Esta capa suele estar situada después de una cantidad determinada por el usuario de capas convolucionales, de *pooling* y ReLU. La imagen que se introduce es significativamente pequeña debido a las reducciones que se le han efectuado en las capas anteriores.

En esta capa, analizamos las imágenes reducidas y creamos una lista de valores a partir de los de dichas imágenes. Luego esta lista corresponde con una de las k imágenes que se han insertado. La única diferencia entre esta capa y la capa convolucional es que en la segunda cada neurona solo está conectada a una pequeña región de la capa de entrada, mientras que en esta las neuronas se conectan a todos los mapas de activación de la capa anterior.

3.3.5. Capa de pérdida

En esta capa es donde comparamos las marcas predichas respecto de las reales de la imagen. Si intentamos clasificar un objeto de k posibles niveles de marca, usaremos la clasificación de pérdida *softmax*, exponencial normalizada. Para regresión de las marcas de una imagen específica, usaremos la función Euclídea. Sus funciones son las siguientes:

- Función de pérdida *softmax*:

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}}$$

donde:

$z_j = \sum_{k=0}^{d} W_{ki} x_k$ son probabilidades a posteriori.

j representa la i-ésima neurona de la capa de pérdida.

K es el número total de neuronas de la capa de pérdida.

W_{ki} son los pesos.

x_i son los valores de entrada de la capa de pérdida.

$\sigma(z)_j$ es la activación de las K neuronas de la capa de pérdida.

- Función de pérdida Euclídea

$$E = \frac{1}{2N} \sum_{i=1}^{N} \left| |\hat{y}_i - y_i| \right|_2^2$$

donde:

K es el número total de neuronas de la capa de pérdida.

$\hat{y}_i$ son las predicciones de las imágenes.

y_i son los valores reales de las imágenes.

3.4. Hiperparámetros

En los apartados anteriores se han introducido conceptos básicos relacionados con las CNNs, y se ha visto que dependen de unos valores o grados de libertad, que denominamos hiperparámetros. No hay bases matemáticas sólidas para la elección de estos, aún sigue siendo una elección basada en la intuición. En la operación convolución, vista en la sección 3.2, se vieron varios hiperparámetros:

- F_d: Número de filtros.

- $2k$ x $2k$: Área del filtro, de convolución o *pooling*.
- S: Salto en el filtro.
- P: Número de filas/columnas de ceros que se añaden al hacer *padding*.

Además de estos, cada arquitectura de redes convolucionales tiene otros hiperparámetros a estudiar, ya que la elección de estos sigue siendo un problema aún sin resolver. Estos son: dimensión de los datos de entrada, número de capas convolucionales, posición del *pooling*, elección de la función *pooling* a utilizar, tamaño del filtro, número de capas FC y número de neuronas que la forman, elección de la función de pérdida, función de activación no lineal, y muchos otros.

3.5. Arquitecturas CNN destacadas

Una vez estudiadas las diferentes capas que tienen las CNNs (convolucionales, *pooling*, ReLU, etc.) y sus parámetros, vemos que existen diferentes posibilidades a la hora de construir la red de neuronas. Algunas de las arquitecturas que más destacan son:

- *LeNet*: desarrollada en los 90 por Yann LeCun, tiene una arquitectura bastante simple donde todo se considera. Se considera el "Hola Mundo" de las CNNs por ser la primera aplicación del campo con éxito. Está formada por: entrada, capa convolucional, ReLU, *pooling*, capa convolucional, ReLU, *pooling*, FC, ReLU, FC y clasificador *softmax*. Vemos un ejemplo de la arquitectura LeNet5, la quinta versión de LeNet:

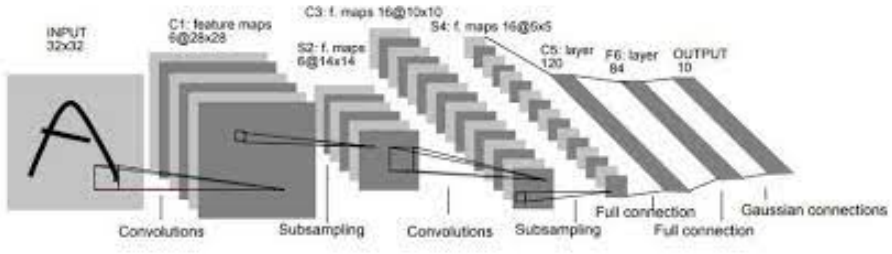


Figura 3.4: LeNet5

- *GoogLeNet*: desarrollada por Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dimitru Ehran, Cinven Vanhoucke y Andrew Rabinovich. Su nombre se debe a que muchos de sus desarrolladores trabajan en Google Inc. Dicen que su arquitectura permite incrementar la profundidad y la anchura de la red manteniendo el límite presupuestario computacional. Esta es tal que así: entrada, capa convolucional, *max-pooling*, capa convolucional, *max-pooling*, comienzo (2 capas), *max-pooling*, comienzo (5 capas), *max-pooling*, comienzo (2 capas), *average-pooling*, DropOut, ReLU y clasificador *softmax*. El modelo de red de neurona convolucional permite aumentar el número de unidades en cada etapa, sin llegar al punto de que se vuelva demasiado complejo. Busca procesar información visual a varias escalas y luego agregar cálculos a la siguiente etapa, así los niveles de abstracción más altos son analizados de manera simultánea. Lo vemos gráficamente:

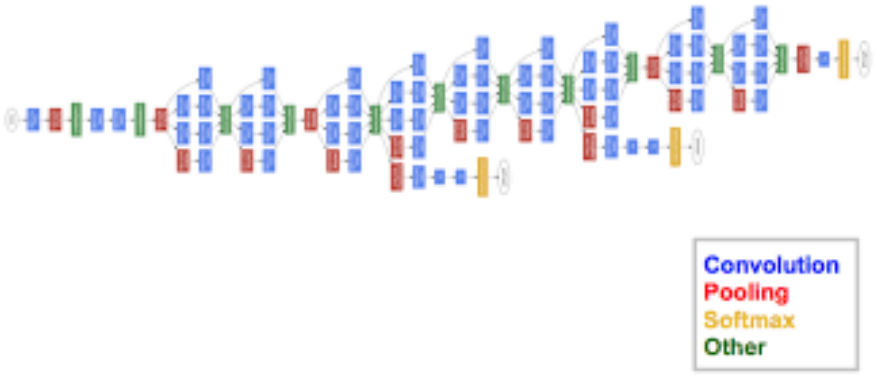


Figura 3.5: GoogLeNet

- *AlexNet*: desarrollada por Alex Krizhevsky, Ilya Sutskever y Geoffrey Hinton. Utiliza "neuronas no saturadas". Su arquitectura es: entrada, 5 capas convolucionales, 3 capas FC y clasificador *softmax*.
- VGGNet: desarrollada por Karen Simonyan y Andrew Zisserman de la universidad de Oxford. El campo receptivo es 3 x 3, con un filtro 1 x 1, salto tamaño 2 y *max-pooling* tamaño 2 x 2. Su arquitectura es: entrada, varias capas convolucionales, 3 capas FC y clasificador *softmax*.
- ResNet: desarrollada por Kaimin He, Xiangyu Zhang, Shaoqing Ren y Jian Sun, todos investigadores en Microsoft. Cuenta con 152 capas. El objetivo de su arquitectura es formar una red que aprenda de las funciones residuales con referencias en las entradas de la capa, en vez de aprender sin referencias. Resulta ser una red bastante fácil de enseñar, más fácil de optimizar y más precisa a partir de cierta profundidad.

3.6. Regularización

Cuando un perceptrón multicapa tiene más de una capa, se sabe que pueden aproximarse a un objetivo concreto, lo cual lleva a que se obtenga sobreajuste (*overfitting*). Para evitarlo, se recomienda regularizar los datos de entrada, sin embargo en el caso de las CNNs el proceso es diferente. Se puede usar:

- DropOut, inspirado en un fenómeno observado en el cerebro humano. Una determinada capa oculta tiene la probabilidad de no activarse como el valor que fijamos como hiperparámetro. El efecto que produce es que la red se vuelve menos sensible a los pesos específicos de las neuronas, dando como resultado una red que es capaz de dar una mejor generalización y un menor sobreajuste.
- Agrupación (*pooling*) estocástica, donde la activación se escoge aleatoriamente. No requiere hiperparámetros y puede usarse como heurística, es decir, con otras técnicas de regularización.

- DropConnect, una generalización del DropOut, donde cada conexión puede disminuir con probabilidad $1 - p$. Cada unidad de esta capa introduce datos de unidades aleatorias de la capa anterior, cambiando en cada iteración. Esto ayuda a garantizar que los pesos no tengan sobreajuste.
- Decadencia del peso, funciona de forma similar a las regularizaciones L1 (coste agregado igual al valor absoluto de los coeficientes de los pesos) y L2 (coste agregado igual al cuadrado de los coeficientes de los pesos) donde penalizamos duramente los grandes vectores de peso.

El método de DropOut es el más usado en las CNNs, debido a que se ha probado que es la técnica más fuerte y efectiva. Más allá de prevenir el sobreajuste, se ha observado que mejora la eficiencia computacional de las redes con una gran cantidad de parámetros, debido a que hace que una red se vuelva más pequeña dada una iteración. El comportamiento de una red más pequeña puede ser promediado para predecir el de una red completa. Además, este método introduce un rendimiento aleatorio que permite promediar el ruido dentro de los datos, de tal manera que se disminuyen las señales ocultas de estos.

Capítulo 4

Librería Keras

La librería *Keras* es un paquete de R que proporciona herramientas para definir y entrenar casi cualquier tipo de modelo de DL. Según la página web de la librería, *Keras* es una API (conjunto de definiciones y protocolos que se usa para diseñar e integrar el software de las aplicaciones) de redes neuronales artificiales de alto nivel desarrollada con el objetivo de permitir la experimentación rápida. El paquete *Keras* tiene las siguientes carásterísticas:

- Permite que se ejecute el mismo código en CPU o en GPU.
- Es una API sencilla de usar que facilita y agiliza el trabajo con modelos de DL.
- Soporte integrado para redes convolucionales, recurrentes y cualquier combinación de ambos.
- Soporta arquitecturas de redes arbitrarias, lo que significa que *Keras* es apropiado para construir cualquier modelo de DL.

Vamos a introducir ahora las diferentes funciones que usaremos junto a sus parámetros más importantes.

4.1. to_categorical()

Transforma un vector o una matriz de 1 columna a una matriz de p columnas. Lo usaremos para aplicar el método de *one hot encoding*, que crea una columna para cada variable o, en nuestro caso, para cada imagen. Parámetros:

- **y**: vector de clases a convertir en matriz.
- **num_classes**: número total de clases.

4.2. keras_model_sequential()

Con esta función se crea un modelo con capas apiladas linealmente. No llevará parámetros y aún no tendrá capas. Simplemente queda el modelo vacío y a continuación se empezarán a añadir las sucesivas capas del modelo.

4.3. layer_conv_...()

Esta capa crea un núcleo convolucional que es convolucionado con los datos de entrada a la capa para producir un tensor de salida. Hay varias funciones dado que los datos pueden venir expresados en 1, 2 o 3 dimensiones, e incluso traspuestos. Nosotros nos centramos en la función *layer_conv_2d()* porque estamos trabajando con imágenes. Parámetros:

- **object**: modelo.
- **kernel_size**: lista de números enteros que especifican las dimensiones de la ventana de convolución. Se puede usar un único entero para especificar el mismo valor para todas las dimensiones.
- **filters**: número de filtros de salida en la convolución.
- **activation**: función de activación a usar. Usaremos de normal "relu".
- **input_shape**: vector que contiene las dimensiones de los datos de entrada y el número de canales (colores). Solo se usa si es la primera capa de la red.

4.4. layer_max_pooling_...()

Esta capa es la que realiza la operación de *pooling* máxima. Al igual que en la anterior, existen 3 funciones, habrá que usar una u otra dependiendo de si tenemos datos temporales (1 dimensión), espaciales (2D) o espacio-temporales (3D). Parámetros:

- **object**: modelo.
- **pool_size**: tamaño del *pooling*.
- Se le pueden dar más parámetros como *padding*, salto, pesos...

4.5. layer_dropout()

Es una de las capas usada para evitar el sobreajuste. Parámetros:

- **object**: modelo.
- **rate**: probabilidad fijada que requiere este método.

4.6. layer_flatten()

Esta capa se usará siempre una vez terminado con las capas de convolución y antes de la capa FC. Con ella se intenta suavizar los datos. Solo es necesario como parámetro el modelo, al igual que todas las capas.

4.7. layer_dense()

Es la capa FC, que puede usarse de 2 formas diferentes, como capa ReLU y como la última de salida. Estos son sus parámetros:

- **object**: modelo.
- **units**: número de neuronas de la capa.
- **activation**: si le damos como parámetro de activación "*relu*", aplicará la capa ReLU; y si ese parámetro se marca como "*softmax*", esta será la capa de salida, la última de nuestro modelo.

4.8. compile()

Una vez aplicadas todas las capas de nuestra red neuronal, hay que compilar. Parámetros:

- **object**: modelo.
- **loss**: hay diferentes categorías de funciones de pérdida, nosotros usaremos "categorical_crossentropy", aunque existen muchas como cuadrado medio del error, *poisson*, etc.
- **optimizer**: optimizer_adadelta y optimizer_sgd() serán los que utilicemos, y ambos son métodos de gradiente descendente estocástico.
- **metrics**: hace referencia a la medida a adoptar para evaluar el modelo durante el entrenamiento, que normalmente usaremos "accuracy".

4.9. fit()

Entrena el modelo para un número determinado de iteraciones. Parámetros:

- **object**: modelo.
- **x**: datos de entrenamiento.
- **y**: resultado de los datos de entrenamiento.

- **epochs**: número de iteraciones.
- **batch_size**: número de muestras por cada actualización.
- **validation_split**: porcentaje de datos de entrenamiento que se usará como datos de validación. Dado entre 0 y 1.

4.10. Librería tensorflow

Será necesario además utilizar la librería *tensorflow*. *tensorflow* es una librería de software de código abierto para el cálculo numérico mediante gráficos de flujo de datos. Este paquete es necesario para el funcionamiento de *keras* y además, usaremos una de sus funciones.

La función que usaremos será **evaluate()**. Esta evalúa el modelo y muestra la tasa de bondad de ajuste y la función de pérdida, los cuales deberán acercarse lo máximo posible a 1 y a 0, respectivamente, que de alcalzarlo, resultaría ser la perfección. Parámetros necesarios:

- **object**: modelo.
- **x**: datos test.
- **y**: resultado de estos datos test.

Capítulo 5

Práctica

5.1. Práctica 1: Números

5.1.1. Librerías

Cargamos las librerías necesarias para nuestro estudio. Entre ellas destacan los paquetes de:

- **Keras**.
- **Tensorflow**.
- **Tidyverse**: que nos simplifica muchas de las funciones.
- **png**: para leer las imágenes de la prueba que estarán en formato .png.
- **EBImage**: usado para leer y tratar las imágenes.
- **kableExtra**: para realizar tablas.

5.1.2. Datos

En esta práctica vamos a utilizar unos datos que nos proporciona el paquete *keras*, el conjunto *mnist* en concreto. Este contiene 60.000 imágenes de números escritos a mano para entrenamiento y 10.000 de test. Los datos vienen proporcionados por el color de cada pixel en una escala de grises, con tamaño 28 x 28, y por supuesto, 1 solo canal de color al ser en blanco y negro.

Dimensión de los datos de entrenamiento:

```
dim(x_train)
```

```
## [1] 60000    28    28     1
```

Dimensión de los datos test:

```
dim(x_test)
```

```
## [1] 10000    28    28     1
```

Las respuestas, tanto de entrenamiento como test, hay que convertirlas en matriz. Para ellos usaremos el método *One Hot Encoding*, utilizando la función **to_categorical()**.

```
y_train <- to_categorical(y_train, 10)
y_test <- to_categorical(y_test, 10)
```

5.1.3. Modelo

Una vez estén leídos los datos, hay que construir el modelo de redes neurolanes. El modelo se crea con la función keras_model_sequential().

La primera capa será una capa convolucional con 32 filtros, cada uno de tamaño 3 x 3. Usará la función de activación "ReLU". Por ser la primera es necesario proporcionarle la dimensiones de los datos de entrada en el parámetro **input_size**, que recordamos que eran 28 x 28 x 1. Junto a esta, tenemos otra vez una capa convolucional. Esta vez usaremos 64 filtros de tamaño 3 x 3 al igual que antes, y la misma función de activación. A continuación de estas, escontramos una capa de agrupación máxima, aplicada por cuadrículas de 2 x 2. Y después utilizaremos una capa *dropout* que dejará al 25 % de las neuronas sin activarse.

Una vez se han terminado las capas convolucionales, encontramos la capa *flatten* que nos suavizará el modelo, dejando todas las neuronas al mismo nivel, antes de introducir la capa FC. Introducimos ahora una capa totalmente conectada con 128 neuronas, utilizando el método ReLU. Después de esta, volvemos a tener una capa *dropout* de 0.5 que nos ayudará a evitar el sobreajuste. Y por último, obtenemos la salida con una capa totalmente conectada, haciendo uso de la activación *softmax* para que nuestra salida esté normalizada, y con un tamaño de 10 clases (0, 1, 2, ..., 9).

Para terminar nuestra red, antes de entrenarlo, será necesario compilar todas las capas. Hacemos uso de la función **compile()**. En ella necesitaremos una función de pérdida, la cual usaremos "categorical_crossentropy", usada para clasificación multiclase donde cada imagen pertenece a una única clase de todas las posibles. Durante el entrenamiento, evaluaremos el modelo con el método *accuracy*. Por último, optimizaremos el modelo con la función **optimizer_adadelta()**, que es un método de gradiente descendente estocástico.

```
modelo <- keras_model_sequential()

modelo %>%
  layer_conv_2d(kernel_size = c(3,3), filters = 32,
                activation = 'relu',
                input_shape = c(28, 28, 1)) %>%
  layer_conv_2d(kernel_size = c(3,3), filters = 64,
                activation = 'relu') %>%
```

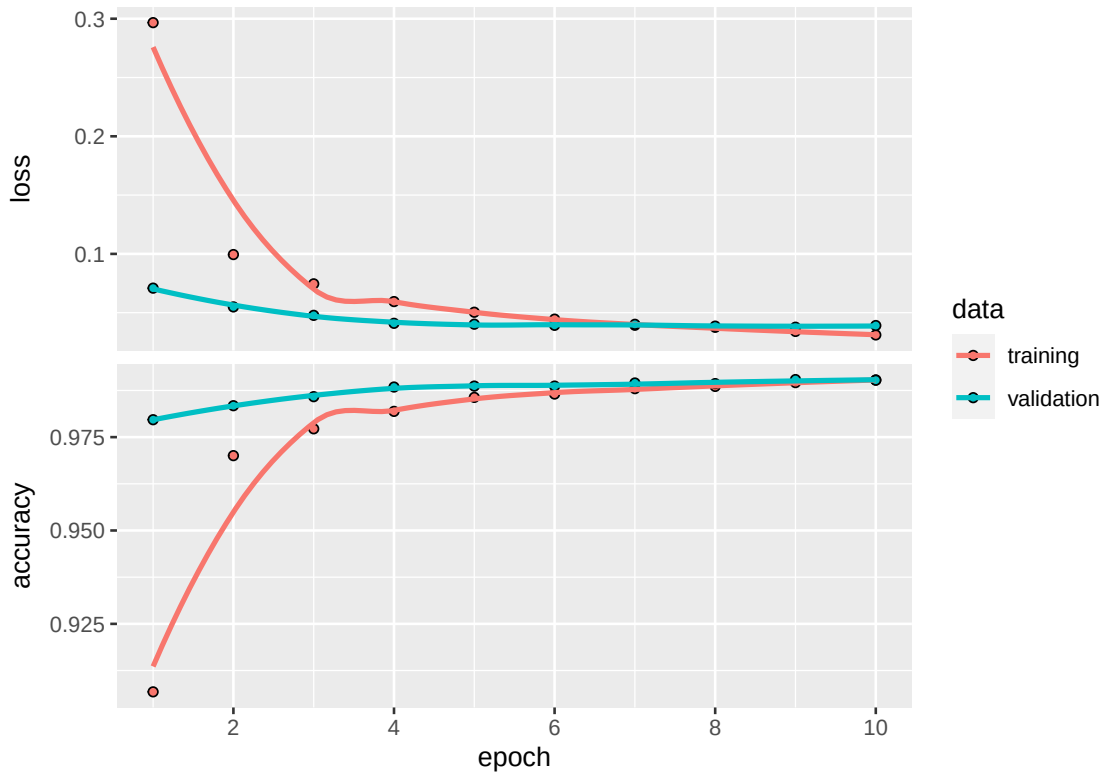
```
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_dropout(rate = 0.25) %>%
  layer_flatten() %>%
  layer_dense(units = 128, activation = 'relu') %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 10, activation = 'softmax')

modelo %>%
  compile(loss = 'categorical_crossentropy',
          optimizer = optimizer_adadelta(),
          metrics = c('accuracy'))
```

Por último, una vez construida la red y compiladas las capas, hay que entrenar el modelo. Tiene que aprender a clasificar los diferentes números partiendo de las 60 mil imágenes que vienen dadas como datos de entrenamiento. Para ello usaremos la función **fit()**. Vamos a realizar 10 iteraciones, con 128 muestras en cada actualización, utilizando el 20 % de los datos para validación. Vemos también el gráfico que muestra el progreso de la función de pérdida y la bondad de ajuste mientras se entrenaba la red.

```
history <- modelo %>%
  fit(x_train, y_train, epochs = 10,
      batch_size = 128, validation_split = 0.2)

plot(history)
```



5.1.4. Evaluación

Realizamos la evaluación de nuestro modelo para los datos test. El objetivo es que *loss* se aproxime lo máximo posible a 0 y *accuracy* a 1. Cuanto más cercanos sean a esos valores, más fiable será la red.

```
##       loss   accuracy
## 0.02843236 0.99119997
```

5.1.5. Prueba

Ahora vamos, en vez de utilizar los datos test proporcionados por R, a realizar una prueba con nuevos números. Con la aplicación *Paint* he creado los 10 números que van del 0 al 9, con un fondo negro y el número escrito en blanco, intentando que sea todo lo parecido posible a mi letra cuando escribo a mano.

Antes de utilizarlos en el modelo, hay que reducirlos a escala de grises, recalcular el tamaño a 28 x 28, combinar la lista y reordenar las dimensiones para que primero distinga entre los 10 números. Veamos estos números primero:

Comprobamos que tenemos 10 imágenes de tamaño 28 x 28 con un solo canal.

```
## [1] 10 28 28  1
```

Tabla 5.1: Probabilidad para la predicción

	0	1	2	3	4	5	6	7	8	9
num_0	99.69	0.00	0.07	0.00	0.00	0.00	0.01	0.00	0.00	0.23
num_1	1.80	74.10	0.03	0.00	16.99	0.87	5.78	0.04	0.37	0.01
num_2	0.00	0.00	100.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
num_3	0.00	0.00	0.00	100.00	0.00	0.00	0.00	0.00	0.00	0.00
num_4	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00	0.00	0.00
num_5	0.00	0.00	0.00	0.00	0.00	100.00	0.00	0.00	0.00	0.00
num_6	0.00	0.00	0.00	0.00	0.00	0.33	99.67	0.00	0.00	0.00
num_7	0.00	0.52	74.66	4.58	0.02	0.00	0.00	20.08	0.11	0.02
num_8	0.00	0.00	0.00	0.00	0.00	0.04	0.00	0.00	99.95	0.00
num_9	0.00	0.02	0.11	14.85	0.48	0.00	0.00	84.21	0.08	0.26

Por último creamos nuestro vector de respuesta, que indicará qué número es cada imagen. Lo convertimos en matriz con el método *One Hot Encoding*.

Realizamos la predicción y vemos las probabilidades de ser clasificado correctamente:

Con estas probabilidades obtenemos las predicciones, que las comparamos con los valores reales:

Tabla 5.2: Predicción VS Real

	0	1	2	3	4	5	6	7	8	9
0	1	0	0	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0	0	0	0
2	0	0	1	0	0	0	0	1	0	0
3	0	0	0	1	0	0	0	0	0	0
4	0	0	0	0	1	0	0	0	0	0
5	0	0	0	0	0	1	0	0	0	0
6	0	0	0	0	0	0	1	0	0	0
7	0	0	0	0	0	0	0	0	0	1
8	0	0	0	0	0	0	0	0	1	0

Se puede observar que la predicción en general es buena, lo cual ya habíamos observado en la evaluación. Aun así, el número 9 sigue siendo complicado clasificarlo, y a veces incluso el 7 también.

5.2. Práctica 2: Vehículos

En este ejercicio vamos a crear un programa para diferenciar imágenes de coche, avión y bicicleta. El programa va a recibir 30 imágenes de cada vehículo y con un modelo de CNN debe aprender cómo es la forma (o silueta) de cada uno. Luego, con una foto de cada vehículo le pediremos que prediga cuál es.

5.2.1. Librerías

Cargamos las librerías necesarias para nuestro estudio. Entre ellas destacan los paquetes de:

- **Keras**.
- **Tensorflow**.
- **Tidyverse**: que nos simplifica muchas de las funciones.
- **EBImage**: usado para leer y tratar las imágenes.
- **kableExtra**: para realizar tablas.

5.2.2. Datos

Leemos los datos y representamos 10 imágenes de cada vehículo para comprobar que se leen bien:

Figura 5.1: Fotos Vehículos Entrenamiento

Tenemos que recalcular el tamaño para que todas las imagenes sean iguales (100 x 100). Luego habrá que combinar la lista con la función **combine()**. Además cambiamos el orden de las dimensiones para que primero distinga entre las 90 imágenes de entrenamiento:

```
for (i in 1:90) {
  train[[i]] <- resize(train[[i]], 100, 100)
}
test <- list()
for (i in 1:3) {
  test[[i]] <- resize(test_original[[i]], 100, 100)
}
```

```
train <- EBImage::combine(train)

test <- EBImage::combine(test)
```

```
train <- aperm(train, c(4, 1, 2, 3))
test <- aperm(test, c(4, 1, 2, 3))
```

```
dim(train)
```

```
## [1]  90 100 100   3
```

Como podemos comprobar tenemos 90 imágenes de tamaño 100 x 100 con 3 canales.

Por último creamos nuestros vectores de respuesta, que indicarán qué vehículo es cada imagen. Los convertimos en matrices con el método *One Hot Encoding*, utilizando la función **to_categorical()**.

```
trainy <- c(rep(0, 30), rep(1, 30), rep(2, 30))
testy <- c(0, 1, 2)
```

```
trainlabels <- to_categorical(trainy)
testlabels <- to_categorical(testy)
```

5.2.3. Modelo

Una vez recogidos todos los datos, hay que construir el modelo de redes neuronales. El modelo se crea con la función keras_model_sequential(). Este modelo se parecerá en cierto modo a la arquitectura LeNet.

Las primeras dos capas serán, cada una, una capa convolucional con 32 filtros, cada uno de tamaño 3 x 3. A la primera de ellas es necesario proporcionarle las dimensiones de los datos de entrada en el parámetro **input_size**, que recordamos que eran 100 x 100 x 3. A continuación de estas, escontramos una capa de agrupación máxima, aplicada por cuadrículas de 2 x 2. Y después utilizaremos una capa *dropout* que dejará al 25 % de las neuronas sin activarse.

Volvemos a encontrarnos con 2 capas convolucionales, esta vez de 64 filtros cada una y una vez más con un tamaño de 3 x 3. También al igual que antes, tenemos una capa

de agrupación máxima de tamaño de cuadrícula 2 x 2, y otra capa *dropout* que dejará un 25 % de las neuronas sin utilizar.

Una vez se han terminado las capas convolucionales, encontramos la capa *flatten* que nos suavizará el modelo, dejando todas las neuronas al mismo nivel, antes de introducir la capa FC. Introducimos ahora una capa totalmente conectada con 256 neuronas, utilizando el método ReLU. Después de esta, volvemos a tener una capa *dropout* del mismo nivel que las anteriores que nos ayudará a evitar el sobreajuste. Y por último, obtenemos la salida con una capa totalmente conectada, haciendo uso de la activación *softmax* para que nuestra salida esté normalizada, y con un tamaño de 3 clases (avión, coche y bicicleta).

Para terminar nuestra red, antes de entrenarlo, será necesario compilar todas las capas. Hacemos uso de la función **compile()**. En ella necesitaremos una función de pérdida, la cual usaremos "categorical_crossentropy", usada para clasificación multiclase donde cada imagen pertenece a una única clase de todas las posibles. Durante el entrenamiento, evaluaremos el modelo con el método *accuracy*. Por último, optimizaremos el modelo con la función **optimizer_sgd()**, que es un método de gradiente descendente estocástico.

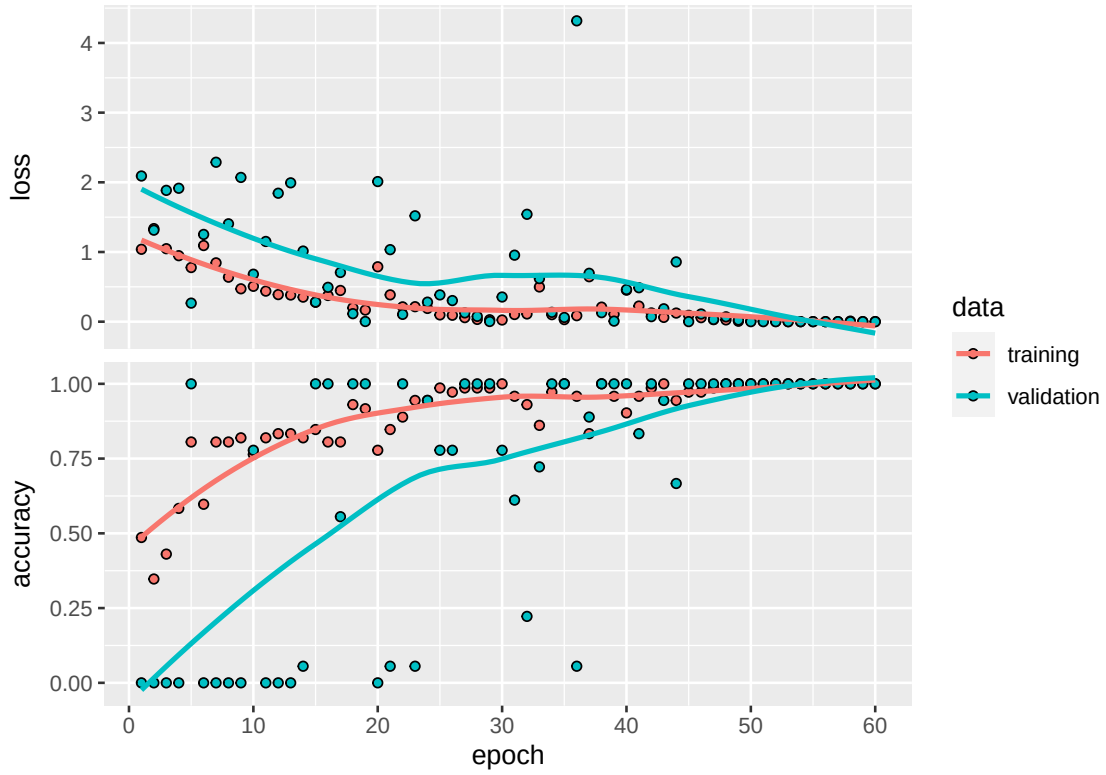
```
modelo <- keras_model_sequential()

modelo %>%
  layer_conv_2d(filters = 32, kernel_size = c(3,3),
                activation = 'relu', input_shape = c(100,100,3)) %>%
  layer_conv_2d(filters = 32, kernel_size = c(3,3),
                activation = 'relu') %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_dropout(rate = 0.25) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3,3),
                activation = 'relu') %>%
  layer_conv_2d(filters = 64, kernel_size = c(3,3),
                activation = 'relu') %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_dropout(rate = 0.25) %>%
  layer_flatten() %>%
  layer_dense(units = 256, activation = 'relu') %>%
  layer_dropout(rate = 0.25) %>%
  layer_dense(units = 3, activation = 'softmax') %>%
  compile(loss = 'categorical_crossentropy',
          optimizer = optimizer_sgd(learning_rate = 0.01, decay = 1e-6,
                                    momentum = 0.9, nesterov = T),
          metrics = c('accuracy'))
```

Por último, una vez construida la red y compiladas las capas, hay que entrenar el modelo. Tiene que aprender a clasificar las imágenes de vehículos partiendo de las 90 imágenes que funcionan como datos de entrenamiento. Para ello usaremos la función **fit()**. Vamos a realizar 60 iteraciones, con 32 muestras en cada actualización, utilizando el 20 % de los datos para validación. Vemos también el gráfico que muestra el progreso de la función de pérdida y la bondad de ajuste mientras se entrenaba la red.

```
history <- modelo %>%
  fit(train, trainlabels, epochs = 60, batch_size = 32,
      validation_split = 0.2)

plot(history)
```



5.2.4. Evaluación y predicción

Dejamos 3 imágenes sin utilizar, una de cada clase, que serán los datos test sobre los que realizaremos la evaluación del modelo, y además, realizaremos una predicción para comprobar que la red efectivamente funciona.

Realizamos la evaluación de nuestro modelo para los datos test. El objetivo es que *loss* se aproxime todo lo posible a 0 y *accuracy* lo haga a 1. Mientras más cercanos sean a esos valores, más fiable será la red.

```
##       loss   accuracy
## 5.1657e-06 1.0000e+00
```

Para terminar obtenemos una tabla de probabilidades que nos dirá la probabilidad de que cada imagen se corresponda a cada clase.

Tabla 5.3: Probabilidad de predicción

	Avión	Coche	Bicicleta
Img_1	100	0	0
Img_2	0	100	0
Img_3	0	0	100

Con estas probabilidades obtenemos que:

```
## La primera imagen es clasificada como: Avión
```

```
## La segunda imagen es clasificada como: Coche
```

```
## La tercera imagen es clasificada como: Bicicleta
```

Y ya por último comprobamos si las predicciones son las correctas.

Tabla 5.4: Predicción vs Real

	Avión	Coche	Bicicleta
Avión	1	0	0
Coche	0	1	0
Bicicleta	0	0	1

Observamos finalmente que sí son correctas; coincide la predicción con la realidad. Luego afirmamos que el modelo CNN es fiable.

Capítulo 6

Conclusiones

En este trabajo se ha realizado una introducción teórica al Deep Learning y a sus herramientas, las Redes Neuronales. Dentro de estas, se ha profundizado más en las Redes Neuronales Convolucionales (CNN). Además se han realizado 2 ejercicios prácticos para ejemplificar lo explicado en la parte teórica.

El Deep Learning es una rama de la Inteligencia Artificial, dentro del campo del Maching Learning. Se trata de un método de aprendizaje automático, que consta de sucesivas capas de aprendizaje. Las estructuras de aprendizaje por capas que se utilizan en el Deep Learning son las llamadas Redes Neuronales. Hay diferentes tipos de Redes Neuronales, cada cual usada para unas actividades específicas. En este trabajo, como ya se ha dicho anteriormente, se ha profundizado en las CNN, aunque cualquiera de las demás Redes Neuronales podrían ser igual de interesantes para un futuro estudio.

Las Redes Neuronales Convolucionales se usan principalmente para el reconocimiento de imagen. Utilizan filtros, matrices que se multiplican a la matriz de píxeles de la imagen original, para detectar colores y marcas, por ejemplo líneas horizontales y verticales. Esta red aprende de analizar una serie de imágenes de entrenamiento. Posteriormente se puede comprobar su bondad de ajuste con unos datos de test.

Con el objetivo de visualizar cómo funcionan estas redes en imágenes reales, se han realizado dos ejercicios prácticos. En el primero se ha modelado una red que aprenda a clasificar números escritos a mano. El segundo utiliza imágenes de 3 tipos de vehículos y el modelo debe aprender a diferenciarlos. En la práctica de los números, no siempre consigue clasificar al 100 %, suele dar problemas el número 9; sin embargo en la segunda práctica siempre consigue una clasificación perfecta de los vehículos.

Por último, este proyecto deja abiertas muchas líneas futuras de trabajo. La primera, y la más evidente, podría ser estudiar qué cambios son necesarios en la estructura de la red de la práctica de los números para mejorar su clasificación. Además, se ha profundizado en las CNN, los demás modelos de Redes Neuronales también podrían ser objeto de estudio de futuros trabajos.

Apéndice A

Apéndice: Código R completo

A.1. Práctica Números

```
rm(list = ls())
```

Librerías

```
library(keras)
library(tensorflow)
library(tidyverse)
library(png)
library(EBImage)
library(kableExtra)
```

Datos

```
mnist <- dataset_mnist()
x_train <- mnist$train$x
y_train <- mnist$train$y
x_test <- mnist$test$x
y_test <- mnist$test$y

x_train <- x_train/255
x_test <- x_test/255

dim(x_train) <- c(60000, 28, 28, 1)
dim(x_test) <- c(10000, 28, 28, 1)
```

```
dim(x_train)
```

```
dim(x_test)
```

```
y_train <- to_categorical(y_train, 10)
y_test <- to_categorical(y_test, 10)
```

Modelo

```
modelo <- keras_model_sequential()

modelo %>%
  layer_conv_2d(kernel_size = c(3,3), filters = 32,
                activation = 'relu',
                input_shape = c(28, 28, 1)) %>%
  layer_conv_2d(kernel_size = c(3,3), filters = 64,
                activation = 'relu') %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_dropout(rate = 0.25) %>%
  layer_flatten() %>%
  layer_dense(units = 128, activation = 'relu') %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 10, activation = 'softmax')

modelo %>%
  compile(loss = 'categorical_crossentropy',
          optimizer = optimizer_adadelta(),
          metrics = c('accuracy'))
```

```
history <- modelo %>%
  fit(x_train, y_train, epochs = 10,
      batch_size = 128, validation_split = 0.2)

plot(history)
```

Evaluación

```
modelo %>%
  evaluate(x_test, y_test)
```

Prueba

```
pic2 <- c('datos/img/num_propio_0.png',
          'datos/img/num_propio_1.png',
          'datos/img/num_propio_2.png',
          'datos/img/num_propio_3.png',
          'datos/img/num_propio_4.png',
```

```
          'datos/img/num_propio_5.png',
          'datos/img/num_propio_6.png',
          'datos/img/num_propio_7.png',
          'datos/img/num_propio_8.png',
          'datos/img/num_propio_9.png')
test <- list()
for (i in 1:10) {test[[i]] <- readPNG(pic2[i])}

for (i in 1:10) {test[[i]] <- test[[i]][,,1]}

test_t <- list()
for (i in 1:10) {
  test_t[[i]] <- t(apply(test[[i]], 2, rev))
}
for (i in 1:10) {
  test_t[[i]] <- t(apply(test_t[[i]], 1, rev))
}

par(mfrow = c(2, 5))
for (i in 1:10) {
  display(test_t[[i]])
}
par(mfrow = c(1,1))

for (i in 1:10) {
  test[[i]] <- EBImage::resize(test[[i]], 28, 28)
}

test <- EBImage::combine(test)

test <- aperm(test, c(3, 1, 2))

y <- tile(test, 10)

dim(test) <- c(10, 28, 28, 1)
```

```
dim(test)
```

```
testy <- c(0:9)
testlabels <- to_categorical(testy)
```

```
prob_test <- modelo %>% predict(test)
imagen <- c("num_0", "num_1", "num_2", "num_3", "num_4",
            "num_5", "num_6", "num_7", "num_8", "num_9")
numeros <- c(0:9)
rownames(prob_test) <- imagen
colnames(prob_test) <- numeros
```

```
prob_test <- round(prob_test, 4)
kable(prob_test*100, "latex", caption = "Probabilidad de predicción") %>%
  kable_styling(latex_options = c("striped", "hold_position"))
```

```
pred_test <- apply(prob_test, 1, which.max)-1

respuesta <- rep(NA, 10)
for (i in 1:10) {
  if (pred_test[i]==i-1) {
    respuesta[i] <- numeros[i]
  } else {
    respuesta[i] <- pred_test[i]
  }
}

compara <- table(Predicted = pred_test, Actual = testy)
kable(compara, caption = "Predicción vs Real") %>%
  kable_styling(latex_options = c("striped", "HOLD_position"))
```

A.2. Práctica Vehículos

```
rm(list = ls())
```

Librerías

```
library(keras)
library(tensorflow)
library(tidyverse)
library(EBImage)
library(kableExtra)
```

Datos

```
pic1 <- c('datos/img/p01.jpg', 'datos/img/p02.jpg', 'datos/img/p03.jpg',
          'datos/img/p04.jpg', 'datos/img/p05.jpg', 'datos/img/p06.jpg',
          'datos/img/p07.jpg', 'datos/img/p08.jpg', 'datos/img/p09.jpg',
          'datos/img/p10.jpg', 'datos/img/p11.jpg', 'datos/img/p12.jpg',
          'datos/img/p13.jpg', 'datos/img/p14.jpg', 'datos/img/p15.jpg',
          'datos/img/p16.jpg', 'datos/img/p17.jpg', 'datos/img/p18.jpg',
          'datos/img/p19.jpg', 'datos/img/p20.jpg', 'datos/img/p21.jpg',
          'datos/img/p22.jpg', 'datos/img/p23.jpg', 'datos/img/p24.jpg',
          'datos/img/p25.jpg', 'datos/img/p26.jpg', 'datos/img/p27.jpg',
          'datos/img/p28.jpg', 'datos/img/p29.jpg', 'datos/img/p30.jpg',
          'datos/img/c01.jpg', 'datos/img/c02.jpg', 'datos/img/c03.jpg',
```

```
          'datos/img/c04.jpg', 'datos/img/c05.jpg', 'datos/img/c06.jpg',
          'datos/img/c07.jpg', 'datos/img/c08.jpg', 'datos/img/c09.jpg',
          'datos/img/c10.jpg', 'datos/img/c11.jpg', 'datos/img/c12.jpg',
          'datos/img/c13.jpg', 'datos/img/c14.jpg', 'datos/img/c15.jpg',
          'datos/img/c16.jpg', 'datos/img/c17.jpg', 'datos/img/c18.jpg',
          'datos/img/c19.jpg', 'datos/img/c20.jpg', 'datos/img/c21.jpg',
          'datos/img/c22.jpg', 'datos/img/c23.jpg', 'datos/img/c24.jpg',
          'datos/img/c25.jpg', 'datos/img/c26.jpg', 'datos/img/c27.jpg',
          'datos/img/c28.jpg', 'datos/img/c29.jpg', 'datos/img/c30.jpg',
          'datos/img/b01.jpg', 'datos/img/b02.jpg', 'datos/img/b03.jpg',
          'datos/img/b04.jpg', 'datos/img/b05.jpg', 'datos/img/b06.jpg',
          'datos/img/b07.jpg', 'datos/img/b08.jpg', 'datos/img/b09.jpg',
          'datos/img/b10.jpg', 'datos/img/b11.jpg', 'datos/img/b12.jpg',
          'datos/img/b13.jpg', 'datos/img/b14.jpg', 'datos/img/b15.jpg',
          'datos/img/b16.jpg', 'datos/img/b17.jpg', 'datos/img/b18.jpg',
          'datos/img/b19.jpg', 'datos/img/b20.jpg', 'datos/img/b21.jpg',
          'datos/img/b22.jpg', 'datos/img/b23.jpg', 'datos/img/b24.jpg',
          'datos/img/b25.jpg', 'datos/img/b26.jpg', 'datos/img/b27.jpg',
          'datos/img/b28.jpg', 'datos/img/b29.jpg', 'datos/img/b30.jpg')
train <- list()
for (i in 1:90) {train[[i]] <- readImage(pic1[i])}

pic2 <- c('datos/img/p111.jpg', 'datos/img/c111.jpg', 'datos/img/b111.jpg')
test_original <- list()
for (i in 1:3) {test_original[[i]] <- readImage(pic2[i])}

par(mfrow = c(6, 5))
for (i in c(1:10, 31:40, 61:70)) {
  plot(train[[i]])
}
par(mfrow = c(1,1))
```

```
for (i in 1:90) {
  train[[i]] <- resize(train[[i]], 100, 100)
}
test <- list()
for (i in 1:3) {
  test[[i]] <- resize(test_original[[i]], 100, 100)
}
```

```
train <- EBImage::combine(train)

test <- EBImage::combine(test)
```

```
train <- aperm(train, c(4, 1, 2, 3))
test <- aperm(test, c(4, 1, 2, 3))
```

```
dim(train)
```

```
trainy <- c(rep(0, 30), rep(1, 30), rep(2, 30))
testy <- c(0, 1, 2)
```

```
trainlabels <- to_categorical(trainy)
testlabels <- to_categorical(testy)
```

Modelo

```
modelo <- keras_model_sequential()

modelo %>%
  layer_conv_2d(filters = 32, kernel_size = c(3,3),
                activation = 'relu', input_shape = c(100,100,3)) %>%
  layer_conv_2d(filters = 32, kernel_size = c(3,3),
                activation = 'relu') %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_dropout(rate = 0.25) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3,3),
                activation = 'relu') %>%
  layer_conv_2d(filters = 64, kernel_size = c(3,3),
                activation = 'relu') %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_dropout(rate = 0.25) %>%
  layer_flatten() %>%
  layer_dense(units = 256, activation = 'relu') %>%
  layer_dropout(rate = 0.25) %>%
  layer_dense(units = 3, activation = 'softmax') %>%
  compile(loss = 'categorical_crossentropy',
          optimizer = optimizer_sgd(learning_rate = 0.01, decay = 1e-6,
                                    momentum = 0.9, nesterov = T),
          metrics = c('accuracy'))
```

```
history <- modelo %>%
  fit(train, trainlabels, epochs = 60, batch_size = 32,
      validation_split = 0.2)

plot(history)
```

Evaluación y predicción

```
par(mfrow = c(1, 3))
for (i in 1:3) {
  plot(test_original[[i]])
}
par(mfrow = c(1,1))
```

```
modelo %>%
  evaluate(test, testlabels)
```

```
prob_test <- modelo %>% predict_on_batch(test)
imagen <- c("Img_1", "Img_2", "Img_3")
vehiculos <- c("Avión", "Coche", "Bicicleta")
rownames(prob_test) <- imagen
colnames(prob_test) <- vehiculos
prob_test <- round(prob_test, 4)
kable(prob_test*100, "latex", caption = "Probabilidad de predicción") %>%
  kable_styling(latex_options = c("striped", "hold_position"))
```

```
pred_test <- apply(prob_test, 1, which.max)-1

respuesta <- rep(NA, 3)
for (i in 1:3) {
  if (pred_test[i]==i-1) {
    respuesta[i] <- vehiculos[i]
  }
}

cat("La primera imagen es predecida como:" , respuesta[1], "\n")
cat("La segunda imagen es predecida como:" , respuesta[2], "\n")
cat("La tercera imagen es predecida como:" , respuesta[3], "\n")
```

```
compara <- table(Predicted = pred_test, Actual = testy)
rownames(compara) <- respuesta
colnames(compara) <- respuesta
kable(compara, caption = "Predicción vs Real") %>%
  kable_styling(latex_options = c("striped", "HOLD_position"))
```

Bibliografía

JJ Allaire and François Chollet. *keras: R Interface to 'Keras'*, 2022. URL https://keras.rstudio.com. R package version 2.8.0.9000.

JJ Allaire, Dirk Eddelbuettel, Nick Golding, and Yuan Tang. *tensorflow: R Interface to TensorFlow*, 2016. URL https://github.com/rstudio/tensorflow.

François Chollet. *Deep Learning with R.* Manning Publications Co., first edition, 2018.

Abhijit Ghatak. *Deep Learning with R.* Springer, first edition, 2019.

Taweh Beysolow II. *Introduction to Deep Learning using R.* Aperss, first edition, 2017.

Pedro L. Luque-Calvo. *Escribir un Trabajo Fin de Estudios con R Markdown*, 2017.

Pedro L. Luque-Calvo. *Cómo crear Tablas de información en R Markdown*, 2019.

Rafael Pino Mejías. *Tema 5: Redes Neuronales Aritificiales.* Asignatura: Estadística Computacional II, Curso 21/22.

Michael Nielsen. *Neural Networks and Deep Learning.* Determination Press, first edition, 2019.

RStudio Team. *RStudio: Integrated Development Environment for R.* RStudio, Inc., Boston, MA, 2015. URL http://www.rstudio.com/.