Doble Grado en Matemáticas y Estadística

TRABAJO FIN DE GRADO

Reconocimiento de imágenes con TensorFlow desde R

Francisco de Paula Quirós Pérez

Sevilla, Junio de 2022

Índice general

Resumen

La presente investigación se refiere al aprendizaje automático, definido como un tipo de inteligencia artificial que proporciona a los ordenadores la capacidad de aprender centrándose en el desarrollo de programas informáticos y que haga que estos actúen de distinta forma en función de los datos expuestos.

La prioridad es utilizar herramientas que nos permitan desarrollar dicho aprendizaje en el reconocimiento y clasificación de imágenes y todo ello en torno a la introducción de las redes neuronales convolucionales, lo cual, supone un antes y un después en el concepto de automatización, como veremos. Conoceremos el funcionamiento de dichas redes desde dentro con la formulación matemática que se lleva a cabo en cada capa y cómo la lectura idónea ante cualquier anomalía en una neruona será desde la última capa a la primera.

Para un análisis en profundidad, necesitamos echar la vista atrás y observar cuánto ha avanzado la tecnología en este terreno desde mediados del siglo XX, desde el primer momento en el que se consideró oportuno un símil entre la red de neuronas de nuestro cerebro y el funcionamiento interno de programas informáticos.

La finalidad será, por tanto, avanzar en la historia de las redes neuronales desde sus inicios, saber cómo funcionan y qué grandes progresos se han hecho recientemente para, más tarde, exponer ejemplos sencillos de aplicación de redes neruonales convolucionales a la creación, entrenamiento, compilación y predicción de un modelo de redes neruonales capaz de clasificar imágenes, apoyados en paquetes de aprendizaje automático como Keras o TensorFlow y a través de R.

Abstract

The current investigation refers to machine learning, defined as a type of artificial intelligence that provides computers with the ability to learn by focusing on the development of programs and that makes them act differently based on the exposed computer data.

The priority is to use tools that allow us to develop this learning in the recognition and classification of images and all this around the introduction of convolutional neural networks, which supposes a before and a after in the concept of automation, as we will see. We will learn how these networks work from the inside with the mathematical formulation that is carried out in each layer and how the ideal reading for any anomaly in a neuron will be from the last layer to the first.

For an in-depth analysis, we need to look back and see how much technology has advanced in this field since the middle of the 20th century, from the first moment in which a simile between the network of neurons in our brain and the internal workings of computer programs of the brain was thought appropriate.

The purpose will be, therefore, to advance in the history of neural networks from their beginnings, to know how they work and what great progress has been made recently to, later, present simple examples of the application of convolutional neural networks to the creation, training, compilation and prediction of a neural network model capable of classifying images, supported by machine learning packages such as Keras or TensorFlow and through R.

Motivación

No es un secreto el hecho de que el aprendizaje automático y, en concreto, las redes neuronales artificiales están adquiriendo un papel muy importante en la actualidad tanto para el análisis estadístico como para el análisis predictivo. El objetivo principal es encontrar patrones que hagan imitar de la mejor manera posible el funcionamiento del cerebro humano a través de un ordenador.

Dada esta importancia mencionada, las empresas están aumentando sus inversiones en el sector del *Machine Learning*, ya que lo consideran de vital importancia para mantenerse competitivos en el mercado. Y es que una de las conclusiones a las que llegó ***Algorithmia*** en su informe anual sobre tendencias empresariales en *machine learning*. El 83 % de las empresas encuestadas dijeron que sus presupuestos de IA y ML habían aumentado. [2]

Avanzando en la era tecnológica cada vez se generan más datos y, lejos de lo que se pudiera pensar, al hablar de "grandes volúmenes de datos" no solo hacemos referencias a simples números, sino también a textos, imágenes o cualquier combinación de ellos.

En el doble grado de Matemáticas y Estadística se estudian algunas pinceladas sobre este profundo entramado y mi afán por conocer más sobre Redes Neuronales y, en concreto, sus numerosas utilidades para la clasificación y el reconocimiento de imágenes me han llevado a considerar este trabajo como la opción idónea.

Objetivos

El objetivo de este trabajo es el desarrollo de una herramienta que actúe como clasificador de imágenes con el fin de saber manipular una serie de datos de entrenamiento, realizar los distintos "test" sobre dichos conjuntos entrenados y ser capaces de hacer predicciones sobre nuevas imágenes administradas.

Para la consecución de dichos objetivos haremos uso de tres pilares fundamentales:

- El **aprendizaje automático** o *Machine Learning* y más concretamente, las Redes Neuronales Convulucionales, ya que podríamos reducir esta tarea a resolver un problema de redes neuronales, donde construiremos un modelo, configuraremos las distintas capas, lo compilaremos, entrenaremos y evaluaremos, estudiando aquella forma de optimizar su arquitectura e hiperparámetros asociados evitando así cualquier problema de sobreajuste.
- **R y el entorno de desarrollo RStudio.** Será nuestro principal software de trabajo, donde importaremos los *datasets*, nos encargaremos de explorar y pre-procesar los datos y llevaremos a cabo las distintas acciones mencionadas en el punto anterior. ***"R"*** como lenguaje de programación es muy versátil e intuitivo debido a los múltiples paquetes estadísticos que ofrece que nos ayudarán como instrumento de clasificación.
- **Tensorflow.** Entre los paquetes y librerías que mencionábamos de *R*, aparece *Tensorflow* y *Keras*. Esta plataforma de código abierto nos permitirá trabajar de la mano de R y RStudio ofreciéndonos todo tipo de detalles sobre aprendizaje automático con cuantiosas bibliotecas, actuando en un segundo plano y que nos servirá para implementar los algoritmos de aprendizaje automático y visualizar los modelos de clasificación propuestos.

Índice de figuras

Capítulo 1

Introducción al aprendizaje automático (Machine Learning)

1.1. Contexto histórico

Hoy en día sabemos que el aprendizaje automático utiliza algoritmos matemáticos para aprender y analizar datos con el objetivo de hacer predicciones y tomar decisiones en un futuro. Nos permite comunicarnos con los ordenadores, conducir coches de manera autónoma, predecir desastres naturales o incluso tratando de combatir ataques terroristas.

Pero ¿cuál es el origen real de *Machine Learning* y cuáles fueron sus primeros hitos?

Este concepto apareció en 1950 de la mano de Alan Turing con su libro *"¿Puede pensar una máquina? (Can machine think?)"*, que proponía la hipótesis de que si una máquina es capaz de convencer a un humano de que no es en realidad un humano, habría alcanzado la inteligencia artificial. [33] Esta fue la llamada "Prueba de Turing".

Fue en 1957 cuando Frank Rosenblatt diseñó la primera red neuronal para ordenadores, conocida en la actualidad como *Modelo Perceptrón*. [8] Este algoritmo fue diseñado para clasificar entradas visuales considerando un hiperplano, separando el espacio en dos regiones y asignando una clase a cada una de ellas.

En 1959, Bernard Widrow y Marcian Hoff (quien, tiempo después, participaría en el invento del microprocesador) crearon dos modelos de redes neuronales llamados *Adaline*, que era capaz de detectar patrones binarios, y *Madaline*, para desarrollar filtros adaptativos que eliminen el eco de las líneas telefónicas. [38]

En 1967, apareció el algoritmo del "vecino más cercano", que posteriormente permitiría a los ordenadores el reconocimiento de patrones básicos, muy usado en la evaluación de la competencia en un territorio comparando la distribución espacial de cierta especie animal o vegetal y para el control de los cambios que se producen a lo largo del tiempo en una determinada área. [12]

Ya en la década de los 90, el trabajo en aprendizaje automático tomó una nueva vertiente, pasando de un enfoque basado en el conocimiento a un enfoque basado en datos. Científicos comenzaron a crear programas para analizar grandes volúmenes de datos y tratar de obtener resultados y aprender de las conclusiones.

Acercándonos algo más a la actualidad, en 2016, el programa de Google *AlphaGo*, se convirtió en el primer programa informático capaz de derrotar a un jugador profesional usando una combinación de *Machine Learning* y técnicas de árboles de búsqueda, conocido como "Árbol de búsqueda de Monte Carlo".

1.2. Redes neuronales

Las redes neuronales se han convertido en la familia de algoritmos de aprendizaje automático más populares de los últimos años, siendo el reconocimiento de imágenes uno de sus principales usos.

Para profundizar sobre su funcionamiento necesitamos entender estos sistemas como la interacción de muchas partes simples trabajando conjuntamente, denominadas neuronas.

La neurona es la principal fuente de procesamiento con la que trabajaremos, de manera similar que una neurona biológica, recibe estímulos externos a través de las conexiones de entrada, realiza una serie de cálculos internos y se genera un valor de salida, es decir, a grandes rasgos no es más que una función matemática.

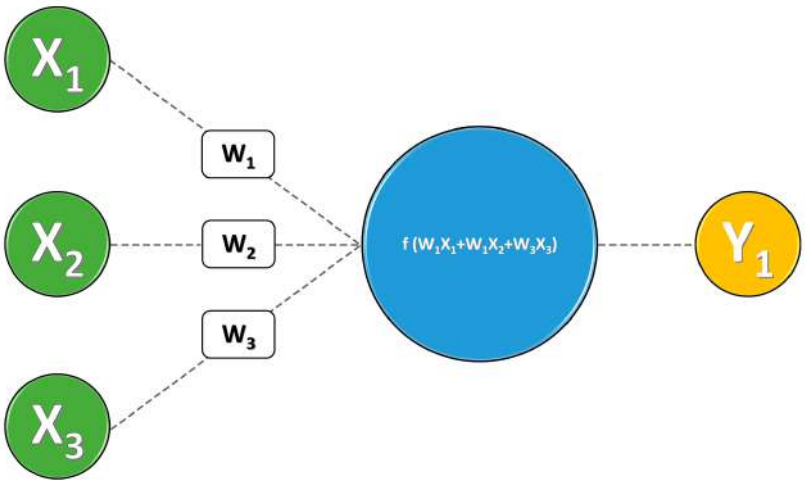


Figura 1.1: Funcionamiento de una neurona. Fuente: Elaboración propia

La neurona internamente usa los valores de entrada realizando una suma ponderada de ellos que viene dada por un "peso" que se le asigna a cada una de las conexiones de entrada. Estos pesos serán los parámetros de nuestro modelo.

Podríamos decir que una neurona es, al fin y al cabo, un modelo de regresión en la que se define una recta o hiperplano y cuya pendiente se ve modificada en función de nuestros parámetros.

$$y = w_1 x_1 + w_2 x_2 + b$$

Cuando la complejidad de los datos aumenta, un modelo de una neurona nos imposibilita la tarea de separar linealmente una nube de puntos. Una solución planteada es la agregación de más neuronas, formando así lo que realmente se conoce como "red neuronal".

Una forma de agrupar las neuronas es por capas, donde las neuronas de la capa N recibirán la información de la capa anterior $N-1$ y los cálculos que realicen los pasarán a la capa siguiente $N+1$. Denominaremos "capa de entrada" y "capa de salida" a aquellas donde se encuentren las variables de entrada y de salida respectivamente y a las capas internas las llamaremos "capas ocultas".

Todo este procesamiento de la información nos otorga una gran ventaja, que la red tenga la capacidad de aprender "conocimiento jerarquizado", un conjunto de modelos para representar las relaciones aparentemente estructurales entre datos, información, conocimiento y sabiduría.

Matemáticamente estamos concatenando diferentes operaciones de regresión lineal. El problema es que el efecto de sumar muchas operaciones de regresión lineal equivale a una única operación, es decir, toda la estructura buscada sobre la red colapsa en una neurona. Para evitar este problema, aparece en escena la "función de activación", donde la suma ponderada antes mencionada es evaluada sobre esta función, añadiendo deformaciones no lineales a nuestro valor de salida para poder encadenar de manera efectiva la computación de varias neuronas. Esta función de activación toma el papel de umbral para comparar los resultados y asignar valores en la salida.

$$Y_1 = f(w_1x_1 + w_2x_2 + w_3x_3)$$

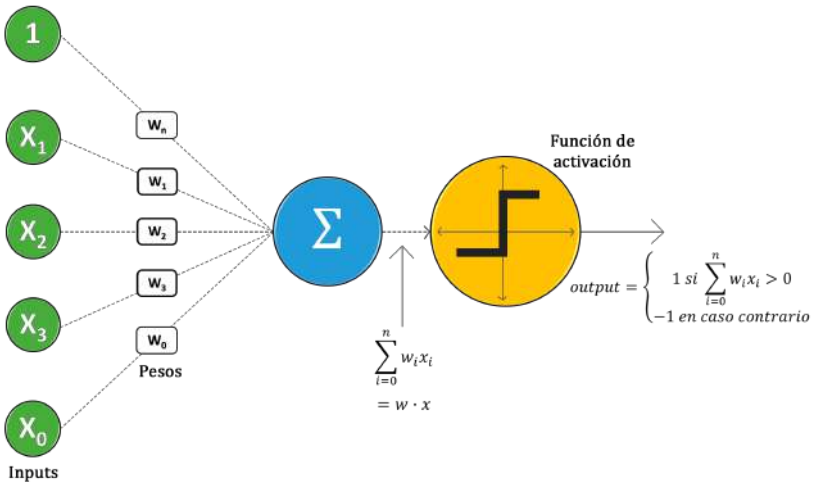


Figura 1.2: Funcionamiento de una red neuronal. Fuente: Elaboración propia

Una vez entendido cómo las redes neuronales son capaces de modelar información compleja, nuestro principal objetivo será que la propia red sea la que aprenda por sí sola, ajustando así los parámetros a partir de los datos.

1.2.1. Retropropagación o Propagación hacia atrás *(Backpropagation)*

Aunque desde los años 50 se conocía el "Perceptrón" y se creía que sería el algoritmo que resolvería una gran cantidad de problemas, pronto se pondrían de manifiesto algunas de sus limitaciones. Como vimos anteriormente una arquitectura de una sola neurona solo es capaz de resolver problemas lineales, siendo necesario concatenar más neuronas para poder abarcar problemas no lineales más complejos. Es aquí donde aparece el problema, el algoritmo de aprendizaje automático utilizado para entrenar al perceptrón no era extensible a otros tipos de redes más complejas, es decir, queríamos redes neuronales, pero no sabíamos cómo entrenarlas. Marvin Minsky y Seymour Papert publicaron en 1969 el libro *"Perceptrons"*, donde demostraron matemáticamente esta problemática. [25]

El gran hallazgo para la inteligencia artificial y la popularidad de las redes neuronales llegaría en el año 1985 de la mano de David E. Rumelhart, Geoffrey E. Hinton y Ronald J. Williams, con la publicación del trabajo *"Learning representations by back-propagating errors"*. [29] Demostraron experimentalmente cómo usando un nuevo algoritmo de aprendizaje se podría conseguir que una red neuronal autoajustara sus parámetros para así aprender una representación interna de la información que estaba procesando, el nombre de este algoritmo fue *"Backpropagation"*.

1.2.1.1. Descenso del Gradiente

Al ir modificando los parámetros de cualquier modelo va variando también el error asociado. Introducimos entonces la "Función de coste", que nos informará de cuál es el error para cada combinación de parámetros.

Todo parece sencillo cuando esta función de coste es una función convexa [ver figura 1.3] y, ya que cumple la propiedad que nos dice que, de encontrar un punto mínimo, podemos estar seguros de que dicho punto será un mínimo global de la función. El problema aparece cuando la función es no convexa, debido a que ya no se cumple dicha propiedad, es decir, podemos encontrarnos con múltiples puntos mínimos y, además, podemos tener otras zonas de la función con pendiente nula, ya sean máximos locales, puntos de inflexión o puntos de silla, lo que nos conduce a un sistema de ecuaciones de ineficiente resolución.

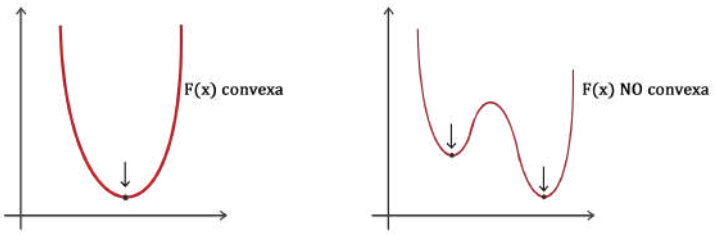


Figura 1.3: Función convexa y no convexa. Fuente: Elaboración propia

Para minimizar el error se suele usar el método del "Descenso del gradiente", con el que conseguimos una buena estrategia para ajustar los parámetros de nuestro modelo.

La clave de este método se basa en evaluar el error del modelo en un punto aleatorio de partida y calcular las derivadas parciales en dicho punto. Con esto obtenemos un vector de direcciones que nos indicaría la pendiente de la función hacia donde el error se incrementa, lo que conocemos como "gradiente". Basta entonces con iterar en la dirección contraria al gradiente, reduciendo el error del modelo.

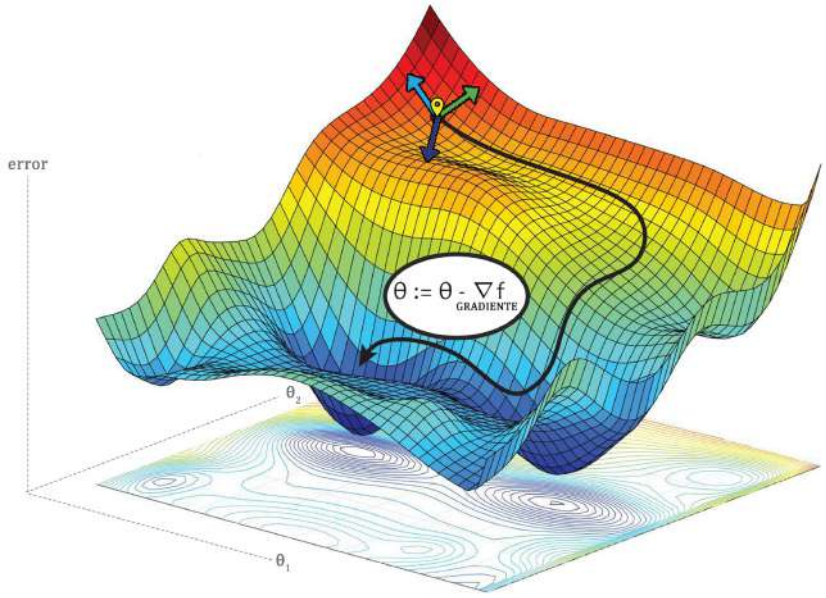


Figura 1.4: Algoritmo de descenso del gradiente. Fuente: Elaboración propia

En el ámbito de las redes neuronales el concepto de gradiente es similar, cómo varía el coste a medida que variamos los parámetros. Pero la forma en la que variar un parámetro puede afectar al resultado final y, por tanto, al coste de la red neuronal, es algo más complicado. Es más, el efecto de un parámetro se ve afectado por el valor del resto de parámetros de capas posteriores, haciendo que las derivadas parciales a la hora de obtener el gradiente sean cada vez más complejas.

Es en este preciso instante donde aparece el algoritmo de "Backpropagation". El proceso consistirá en optimizar la función de coste usando este algoritmo para obtener el vector de gradientes dentro de la complejidad de la arquitectura de la red neuronal.

Es conveniente hacer un estudio de cuanta "responsabilidad" tiene cada neurona en la aparición de algún error en la red. Este análisis tiene sentido realizarlo hacia atrás, desde la señal de error hacia las primeras capas. Esto se debe a que, en una red neuronal, el error de las capas anteriores depende directamente del error de las capas posteriores. Así podemos medir el nivel de influencia de cada neurona en el resultado final de manera eficiente, mediante la retropropagación de errores, operando de manera recursiva capa tras capa desplazando el error hacia atrás. Partimos del error final analizando la última capa, podremos hacer un reparto sobre la influencia de cada neurona en el error final y lo usaremos principalmente para saber cuánto hay que modificar cada parámetro en dicha neurona. Una vez hemos imputado los errores a las neuronas de dicha capa, podemos repetir el mismo proceso de antes como si este fuera el error final de nuestra red, es decir, asumiendo que esta es nuestra nueva última capa. Gracias a esto, cuando hayamos llegado a la primera capa habremos obtenido cual es el error para cada neurona y para cada uno

de sus parámetros solamente propagando una única vez el error hacia atrás, lo cual hace de este algoritmo una estrategia muy eficiente.

Antes de existir esta técnica, el cálculo de la responsabilidad de cada neurona se hacía siguiendo una estrategia de "fuerza bruta", preguntándose para cada neurona cómo variaba el coste cuando se introducía un pequeño cambio. En la práctica, esto requería computar un gran número de veces pasos hacia adelante en nuestra red siendo así muy ineficiente de entrenar. Por el contrario, lo propuesto en el algoritmo de "Backpropagation" permite obtener todos esos errores con un único pase hacia atrás. Esos errores son los que usaremos para calcular las derivadas parciales de cada parámetro de la red, conformando así el vector gradiente que es lo que necesita el algoritmo del descenso del gradiente para minimizar el error y, por tanto, entrenar a la red.

1.2.1.2. Un enfoque de formulación matemática

En un caso más práctico, si disponemos de una red neuronal, tendrá sus parámetros inicializados de forma aleatoria, siendo entonces el resultado obtenido para cualquier input una variable aleatoria y, al compararlo con el valor real, probablemente la predicción no será del todo correcta y la función de coste le asignará un error muy elevado. Este error lo usaremos para entrenar a la red.

- ¿Cómo varía el coste ante un cambio del parámetro ω?

$$y = w_1 x_1 + w_2 x_2 + b$$

Recordemos que en una red neuronal teníamos dos tipos de parámetros, los pesos ω y el término de sesgo b, por tanto, tendremos que calcular $\frac{\partial C}{\partial \omega}$ y $\frac{\partial C}{\partial b}$

Como hemos visto, empezamos a trabajar hacia atrás, calculando las derivadas parciales de la última capa. Suponiendo que nuestra red tiene L capas, $\frac{\partial C}{\partial \omega^L}$, $\frac{\partial C}{\partial b^L}$

- ¿Cómo realizamos estos cálculos?

En el funcionamiento de la neurona, el parámetro ω participaba en una suma ponderada a la que nos referiremos como $Z^L = W^L X + b^L$. Esta suma ponderada sería evaluada por la función de activación $a\left(Z^L\right)$ y el resultado de las activaciones de la neurona en la última capa conformarían el resultado de la red que luego sería evaluada por la función de coste $C\left(a\left(Z^L\right)\right)$ para determinar así el error de la red.

Obtenemos, por tanto, una composición de funciones de la que calcularemos las derivadas parciales directamente mediante la regla de la cadena.

$$\frac{\partial C}{\partial \omega^L} = \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial \omega^L} \qquad \frac{\partial C}{\partial b^L} = \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial b^L}$$

$$Z^L = W^L a^{L-1} + b^L \qquad C\left(a^L\left(Z^L\right)\right)$$

Cabe destacar que estas derivadas tienen un cálculo sencillo y directo. Por ejemplo, si tomamos el Error Cuadrático Medio como nuestra función de coste,

$$C\left(a_j^L\right) = \frac{1}{2}\sum_j \left(y_j - a_j^L\right)^2$$

y queremos hallar cómo varía el coste de la red al modificar la activación de las neuronas en la última capa, estamos hallando realmente la derivada de la activación con respecto al output de la red neuronal,

$$\frac{\partial C}{\partial a_J^L} = \left(a_j^L - y_j\right)$$

Por otra parte, si queremos averiguar cómo varía el output de la neurona cuando variamos la suma ponderada de dicha neurona, tomando la función sigmoide como función de activación de nuestra red, por ejemplo,

$$a^L(z^L) = \frac{1}{1+e^{-z^L}}$$

$$\frac{\partial a^L}{\partial z^L} = a^L(z^L)\cdot(1-a^L(z^L))$$

Finalmente, si queremos conocer cómo varía la suma ponderada "z" con respecto a una variación de los parámetros "ω", los pesos, y "b", el sesgo, calculamos

$$z^L = \sum_i a_i^{L-1}\omega_i^L + b^L$$

$$\frac{\partial z^L}{\partial b^L} = 1 \qquad\qquad \frac{\partial z^L}{\partial \omega^L} = a^{L-1}$$

La derivada respecto del término de sesgo es 1, pues es independiente y su derivada es constante. Respecto a ω, es el valor de entrada a la neurona que realiza esa conexión para la cual el parámetro hace referencia. En este caso, los valores de entrada de la neurona a^{L-1} se corresponden con la salida de las neuronas de la capa anterior, la capa $L-1$.

Haciendo uso de la composicion de derivadas parciales del principio de la sección,

$$\frac{\partial C}{\partial \omega^L} = \frac{\partial C}{\partial a^L}\cdot\frac{\partial a^L}{\partial z^L}\cdot\frac{\partial z^L}{\partial \omega^L}$$

tenemos que $\frac{\partial C}{\partial z^L} = \frac{\partial C}{\partial a^L}\cdot\frac{\partial a^L}{\partial z^L}$ representa cómo varía el error en función del valor de z que es la suma ponderada calculada dentro de la neurona, es decir, en qué grado se modifica el error o coste cuando se produce un pequeño cambio en la suma de la neurona. Cuanto mayor sea dicha derivada, mayor sensibilidad tendrá el resultado final ante un ligero cambio en el valor de la neurona. Por el contrario, cuanto más pequeño sea el valor de la derivada, menos afectará al valor de la red una modificación del valor de la suma. Es decir, el valor de $\frac{\partial C}{\partial z^L}$ nos informará sobre el grado de "responsabilidad" de la neurona en el resultado final y, por tanto, en el error. Es por ello que interpretaremos esta derivada como el **error imputado a la neurona** y que representaremos mediante δ^L. Simplificando y reestructurando nuestra expresión inicial en función del error de las neuronas de la capa L, obtenemos

$$\delta^L = \frac{\partial C}{\partial a^L}\cdot\frac{\partial a^L}{\partial z^L}$$

$$\frac{\partial C}{\partial b^L} = \delta^L \cdot \frac{\partial z^L}{\partial b^L} = \delta^L \cdot 1 = \delta^L$$

$$\frac{\partial C}{\partial \omega^L} = \delta^L \cdot \frac{\partial z^L}{\partial \omega^L} = \delta^L a^{L-1}$$

Hemos deducido, por tanto, tres expresiones diferentes que nos permiten obtener las derivadas parciales deseadas para la última capa.

Aunque a priori pueda parecer complicado el procedimiento ya que solo hemos operado en las neuronas de la última capa, faltando así las $L-1$ capas restantes de la red, realmente es ahora donde el algoritmo de *backpropagation* alcanza su esplendor.

Si queremos ahora calcular los parámetros de la capa anterior $L-1$, repetimos el procedimiento

$$\frac{\partial C}{\partial \omega^{L-1}} = \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial a^{L-1}} \cdot \frac{\partial a^{L-1}}{\partial z^{L-1}} \cdot \frac{\partial z^{L-1}}{\partial \omega^{L-1}} = \delta^L \cdot \frac{\partial z^L}{\partial a^{L-1}} \cdot \frac{\partial a^{L-1}}{\partial z^{L-1}} \cdot a^{L-2}$$

$$\frac{\partial C}{\partial b^{L-1}} = \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L} \cdot \frac{\partial z^L}{\partial a^{L-1}} \cdot \frac{\partial a^{L-1}}{\partial z^{L-1}} \cdot \frac{\partial z^{L-1}}{\partial b^{L-1}} = \delta^L \cdot \frac{\partial z^L}{\partial a^{L-1}} \cdot \frac{\partial a^{L-1}}{\partial z^{L-1}} \cdot 1$$

Es decir, en la expresión anterior, lo único que necesitamos calcular es $\frac{\partial z^L}{\partial a^{L-1}}$, que representa la variación de la suma ponderada de una capa cuando se varía el *output* de una neurona en la capa previa. Esta derivada es también fácil de obtener y no es más que la matriz de parámetros que conecta ambas capas W^L.

Lo que estamos realizando, a fin de cuentas, es desplazar el error de una capa a la capa anterior, distribuyendo el error en función de cuáles son las ponderaciones de las conexiones, mediante operaciones sucesivas.

En resumen, los pasos a seguir recorriendo todas las capas de la red, serían los siguientes:

1. Cómputo del error de la última capa

$$\delta^L = \frac{\partial C}{\partial a^L} \cdot \frac{\partial a^L}{\partial z^L}$$

2. Retropropagamos el error a la capa anterior

$$\delta^{l-1} = W^L \delta^L \cdot \frac{\partial a^{l-1}}{\partial z^{l-1}}$$

3. Calculamos las derivadas de la capa usando el error

$$\frac{\partial C}{\partial b^{l-1}} = \delta^{l-1} \qquad\qquad \frac{\partial C}{\partial \omega^{l-1}} = \delta^{l-1} a^{l-2}$$

Así, hemos conseguido obtener todos los errores y las derivadas parciales de nuestra red haciendo uso exclusivamente de esas cuatro expresiones, que nos informan de cómo tenemos que usar el error de la capa anterior para calcular el error en la capa en la que nos situamos.

1.3. Minería de datos

La minería de datos se encarga de descubrir patrones en grandes cantidades de datos, por eso es usado principalmente en la estadística y en las ciencias computacionales. Forma parte, como etapa de análisis, del proceso de *"knowledge discovery in databases"* o KDD una vez que hemos seleccionado y depurado toda información irrelevante.

- ¿Cómo realiza el proceso de encontrar patrones?

Es importante hacer uso de clasificadores los cuales son modelos que describen cómo se comporta una base de datos, asociando una etiqueta en función del tipo de dato encontrado en la información. Esta técnica es aplicada en algoritmos de inteligencia artificial como regresión lineal, árboles de decisión y como clasificador de imágenes mediante redes neuronales.
Para un adecuado etiquetado de la información, al clasificador se le otorgan unos datos de entrenamiento que posteriormente serán evaluados para conocer la fiabilidad de la clasificación de cada dato.

Las aplicaciones de la búsqueda de patrones en el mundo real son muy variadas, por ejemplo, se puede estudiar la efectividad de un determinado medicamento en medicina dependiendo de ciertos datos de pacientes o un banco puede predecir la fiabilidad de una persona a la hora de pedir un préstamo.

Podríamos, por tanto, resumir el proceso de *"data mining"* en cuatro etapas principales:

1. Determinación de objetivos
2. Procesamiento de datos
3. Determinación del modelo
4. Análisis de resultados

- ¿Cuál es la unión entre la minería de datos y las redes neuronales?

Aunque los métodos basados en redes neuronales no han sido comúnmente usados para tareas de *"data mining"* debido a que a menudo producen modelos incomprensibles para el ser humano o que requieren mucho tiempo para su entrenamiento, se han ofrecido algoritmos y enfoques facilitando este trabajo, como *"rule extraction"* o "extracción de reglas".

En dicho enfoque, la minería de datos permite extraer reglas simbólicas concisas de una red neuronal. La red se entrena primero para lograr una tasa de precisión determinada, luego, las conexiones redundantes de la red se eliminan mediante un algoritmo de poda de la red, se analizan los valores de activación de las unidades ocultas y se generan reglas de clasificación a partir del resultado de este análisis.

A fin de cuentas, son las habilidades de reconocimiento de patrones y estimación las que hacen que las redes neuronales artificiales (RNA) tengan una utilidad mayúscula en la minería de datos.

A medida que los volúmenes de datos aumentan, se vuelve más necesario la optimización y automatización del procesamiento.

Las tareas de la minería de datos se pueden dividir en dos categorías: Minería descriptiva y predictiva de datos. La minería descriptiva proporciona información para entender lo que sucede dentro de los datos sin una idea predeterminada. En cambio, la minería predictiva permite al usuario enviar registros con valores de campo desconocidos y el sistema averiguará los valores desconocidos basándose en patrones anteriormente descubiertos. La acción más común en la minería de datos es la clasificación, reconocer patrones que describe un grupo al que pertenece un elemento examinando elementos existentes que ya han sido clasificados e infiriendo un conjunto de reglas.

La predicción es la construcción y el uso de un modelo para evaluar la clase de un objeto sin etiquetar o evaluar el valor o rangos de valores que es probable obtener.

Por todo esto, la minería de datos en redes neuronales es el pilar fundamental para la clasificación de imágenes. Este proceso de construir una red, entrenarla y evaluar datos de distintas imágenes tratando de encontrar patrones que nos permita diferenciar y caracterizar elementos, es lo que precisamente realizaremos más adelante con imágenes.

1.4. Redes neuronales convolucionales para la clasificación de imágenes

1.4.1. Introducción

Las redes neuronales convolucionales funcionan de una forma similar a cómo lo hace el ojo humano.

Si nos encontramos ante la imagen de una cara, somos capaces de describirla a la perfección con solo mirarla; esto es porque escaneamos la imagen y detectamos la presencia de ciertos elementos que sabemos, gracias a nuestro aprendizaje, que conforman un rostro (distinguimos entre ojos, nariz, boca, etc).

Y, ¿cómo sabemos detectar un ojo?, de la misma forma, hemos detectado determinados elementos como la pupila, las pestañas o las cejas. Así, de manera sucesiva, si reproducimos los pasos que realiza nuestro córtex visual, nos encontramos ante un procesamiento en cascada, donde primero se identifican aquellos elementos básicos y generales y donde en posteriores capas esto se combina para generar patrones cada vez más complejos.

Esta neurociencia fue la fuente de inspiración para investigadores como Yann LeCun quien en 1989 introdujo el primer diseño de una red neuronal convolucional para la detección de números escritos en cheques bancarios. [20]

Una red neuronal convolucional es, por tanto, un tipo de red neuronal cuyo diseño ha sido pensado para sacar partido a la estructura espacial de una imagen, es decir, con una red neuronal multicapa clásica, lo que estaremos haciendo es introducir el valor de cada píxel como si de una variable independiente se tratara, es decir, la red neuronal multicapa solo vería un vector plano de píxeles. Pero sabemos que esta independencia de píxeles no es algo habitual en una imagen, normalmente el valor de un píxel va a estar ligado al de sus píxeles vecinos tanto en su ancho como en el alto y esto es lo que hace que surjan

estructuras, formas y patrones que analizadas correctamente nos pueden beneficiar a la hora de entender una imagen. Con esta idea surgen las redes neuronales convolucionales.

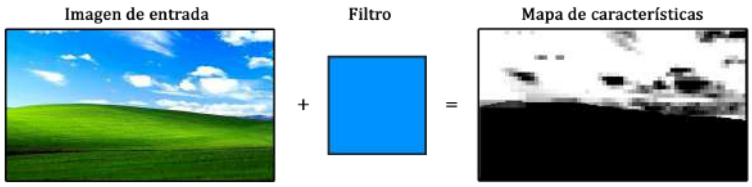


Figura 1.5: Escaneo del filtro generando un mapa de características. Elaboración propia

Este tipo de redes se caracterizan por aplicar un tipo de capa donde se realiza una operación matemática conocida como convolución. Una convolución aplicada sobre una imagen es una operación que, modificando los valores de los píxeles, es capaz de producir otra imagen. Concretamente, cada píxel nuevo que vayamos a generar se calculará colocando una matriz de números, que denominaremos "flitro" o "kernel", sobre la imagen original y multiplicaremos y sumaremos los valores de cada píxel vecino para obtener el nuevo valor. Desplazando el filtro y realizando estas operaciones por todo el recorrido de la imagen, obtendremos esta nueva imagen. Dependiendo, por tanto, de cómo configuremos los parámetros de nuestro filtro, podremos obtener una imagen u otra.

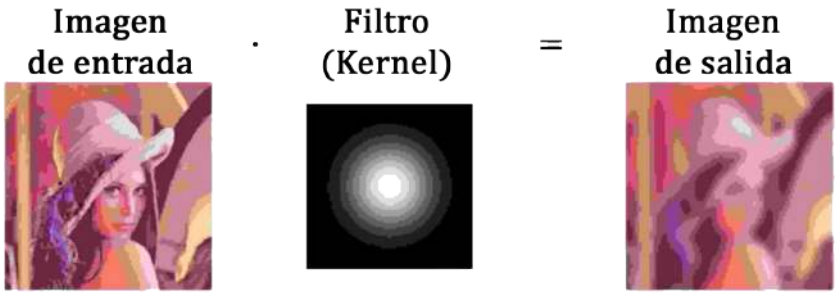


Figura 1.6: Convolución aplicada sobre una imagen

Esta operación de convolución sobre una imagen puede detectar detalles diferentes en función de los valores del filtro que definamos. Estos valores no los proporcionamos nosotros directamente, sino que es la propia red neuronal la que irá aprendiendo para poder hacer mejor su tarea. **Aprender estos filtros para detectar patrones es el principal trabajo de la red neuronal convolucional.** A cada una de las imágenes generadas se le conoce como mapa de características ya que actúa como un mapa donde se nos indica en qué parte de la imagen se ha detectado la característica buscada por dicho filtro.

En otras palabras, el resumen del proceso sería el siguiente: Se introduce una imagen, se le aplican una serie de convoluciones y se genera un conjunto de mapas de características.

El gran potencial de este tipo de redes se encuentra en que esta operación se va a realizar secuencialmente, donde el *output* de una de las capas se va a convertir en el *input* de la siguiente.

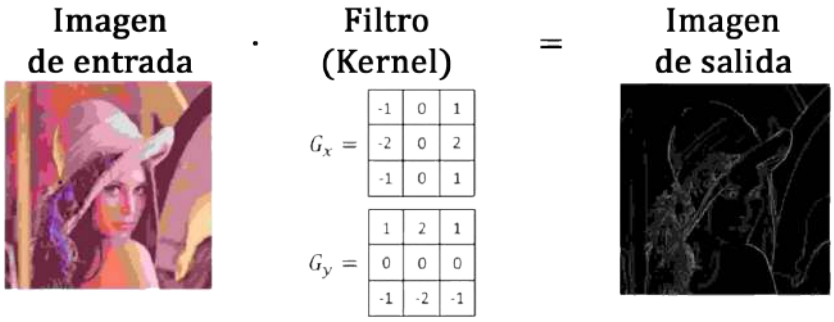


Figura 1.7: Otro ejemplo de convolución

Una cosa similar ocurre cuando tenemos una imagen a color con los tres canales *"RGB"*, que se podrían interpretar como si tuviéramos tres mapas de características diferentes donde se han detectado elementos en rojo, en verde y en azul.

Podemos utilizar filtros de tamaños 3x3, 5x5 o 7x7 píxeles para ir escaneando poco a poco nuestra imagen en búsqueda de patrones, pero ¿es posible que un filtro tan pequeño pueda detectar detalles tan complejos como un ojo humano?

La respuesta la encontramos en la sucesión de capas, en el *"deep"* del *"Deep Learning"*, en la profundidad de la arquitectura de nuestra red. Lo que sucede es que la operación de convolución cada vez se va a ir haciendo más potente, donde antes teníamos una región de 9 píxeles, nuestro filtro lo ha convertido en un único píxel de información. Aplicando de nuevo, una convolución sobre estos mapas de características, en realidad estaremos accediendo a cada vez más información espacial de la imagen original.

Una operación de convolución por sí sola no puede detectar cosas más complejas que bordes y patrones muy simples, pero volver a hacer detecciones sobre las detecciones anteriores permite componer cada vez patrones más complejos.

El diseño de una arquitectura convolucional se suele representar 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; el número de mapas de características que vamos detectando va en aumento. Es al final de este "embudo convolucional" llegaremos a un punto en el que habremos detectado todos los patrones necesarios, contaremos con mapas de características que podremos introducir como *inputs* independientes dentro de una red neuronal multicapa que acabará por tomar la decisión de averiguar qué es o que hay en la imagen.

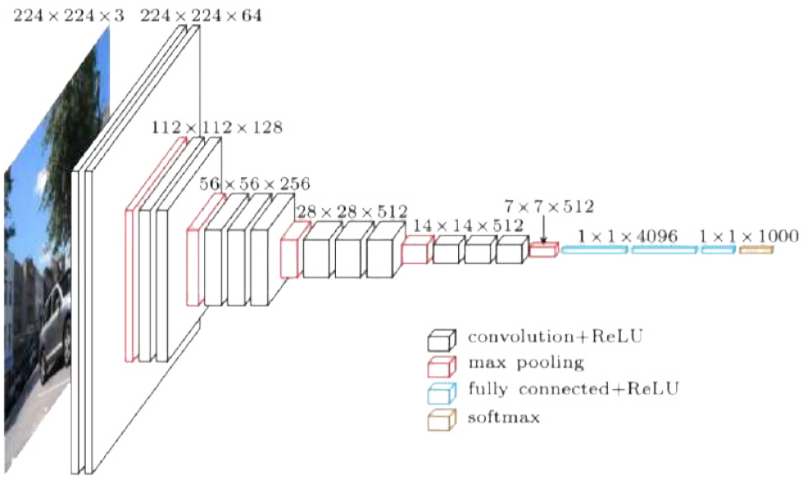


Figura 1.8: Embudo convolucional

1.4.2. Aplicación a la clasificación de imágenes

Dentro del ámbito de las redes neuronales, nos serán de mayor utilidad las redes neuronales convolucionales, pero ¿por qué?

Para la clasificación de imágenes no basta con una red neuronal regular, es cierto que podrá predecir con una alta probabilidad el tipo de prenda de ropa que se está mostrando de una imagen, por ejemplo. Pero en realidad, esta precisión es exclusiva de un set de datos concreto, es decir, introduciendo una imagen cualquiera de ropa diferente a la de los datos entrenados, no será capaz de identificarla con facilidad. Es más, si modificamos el propio conjunto de datos de entrada con el que desarrollamos la red (aumentando o disminuyendo el tamaño de una imagen, por ejemplo), también nos encontraríamos con problemas a la hora de clasificar.

Podríamos concluir entonces que las redes neuronales hacen un fantástico trabajo de clasificación y predicción de imágenes siempre y cuando se establezca previamente un marco de datos y características que poder comparar con los datos que se hayan usado en el periodo de entrenamiento de la red.

¿Qué novedad introducen las redes neuronales convolucionales? Estas redes se van a encargar de extraer características que les permitan identificar imágenes. El objetivo es no ceñirse a una imagen concreta, memorizarla, entrenar una red entorno a ella y buscar la clasificación de otra imagen siempre y cuando sea muy similar a la memorizada. Más bien pretendemos realizar un proceso de recepción, extracción de características y a partir de ahí clasificar en capas posteriores.

El proceso consiste en recibir una imagen que tenga una resolución de, por ejemplo, 28x28 píxeles. Por tanto, tendremos en nuestra red una capa de entrada de 784 neuronas, una por cada píxel y, a parte de capas ocultas propias de la red, una capa de convolución y agrupación para extraer las características particulares de la imagen para poder clasificarla posteriormente con las capas regulares.

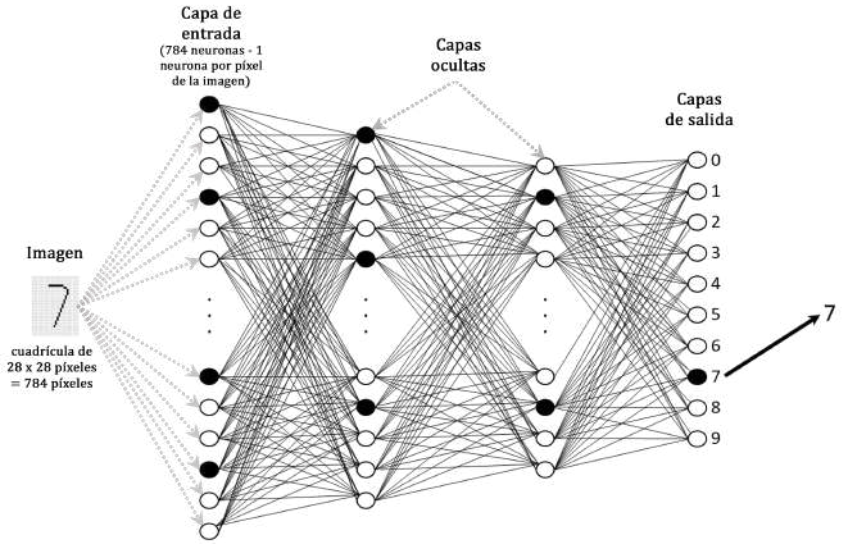


Figura 1.9: Red neuronal convolucional para el tratamiento de píxeles

Capítulo 2

Bibliotecas y *"Frameworks"* que nos serán de utilidad

2.1. Keras

Keras es una biblioteca de código abierto (con licencia MIT) escrita en Python, un lenguaje de programación dominante en el mundo del aprendizaje automático, que se basa principalmente en el trabajo de François Chollet, un desarrollador de Google, en el marco del proyecto ONEIROS (Open-ended Neuro-Electronic Intelligent Robot Operating System) [30]. El objetivo de la biblioteca es acelerar la creación de redes neuronales: para ello, Keras no funciona como un *framework* independiente, sino como una **interfaz de programación de aplicaciones de uso intuitivo** (API) que permite acceder a varios ***frameworks* de aprendizaje automático** y desarrollarlos. Entre los *frameworks* compatibles con Keras, se incluyen Theano, Microsoft Cognitive Toolkit (anteriormente CNTK) y TensorFlow.

Esta interfaz es una excelente manera de comenzar a implementar el aprendizaje automático y, específicamente, las redes neuronales profundas ya que permite una experimentación rápida y se enfoca en su sencillez en la utilización. Es una opción popular tanto para principiantes o aficionados como para desarrolladores profesionales y proporciona módulos incorporados para todos los cálculos de redes neuronales. Al mismo tiempo, los cálculos que implican tensores, gráficos de cálculo o sesiones, pueden hacerse a medida utilizando la API del núcleo de TensorFlow, lo que provoca una flexibilidad y un control total sobre nuestra aplicación y nos permite implementar nuestras ideas en un tiempo relativamente corto.

Entre las características anteriormente mencionadas de Keras, destacamos:

- Eficiente funcionamiento tanto en la CPU como en la GPU.
- Ofrece una API consistente que proporciona la información necesaria cuando se produce un error.
- Podemos personalizar las funcionalidades de nuestro código en gran medida.
- Permite trabajar tanto con redes neuronales convolucionales (CNN) como con redes neuronales recurrentes (RNN), para una variedad de aplicaciones como Visión Computacional y el análisis de series temporales, respectivamente.

- Soporta arquitecturas de red arbitrarias, poniendo a disposición de los usuarios la posibilidad de compartir modelos y capas.

Como interfaz universal para las plataformas de Machine Learning, Keras se utiliza actualmente en una gran variedad de proyectos en el campo de la inteligencia artificial, a mediados de 2018, esta biblioteca contaba con más de 250.000 usuarios individuales, y este número ha aumentado mucho después de su inclusión en el software TensorFlow.

Gracias a la libertad de elección del *framework* subyacente, la gratuidad de las licencias y su independencia de plataforma, Keras es de gran ayuda para todo tipo de aplicaciones profesionales de redes neuronales tanto en la industria como en la investigación. Por ejemplo, empresas conocidas como Netflix, Uber o Yelp, así como organizaciones como la NASA o el Centro Europeo para la Investigación Nuclear (CERN), utilizan Keras en sus proyectos.

2.2. Herramientas para la clasificación de imágenes: Tensorflow

2.2.1. Introducción

TensorFlow es una librería de código libre para *Machine Learning* desarrollado por Google para satisfacer sus necesidades a partir de redes neuronales artificiales y, como tal, está basado en redes neuronales de aprendizaje profundo.

En cuanto a su arquitectura, está creado en C++ y Python, puede correr en múltiples procesadores y tarjetas gráficas y proporciona una API en Python.

En la actualidad, TensorFlow se encuentra entre las librerías y herramientas más populares de *Machine Learning*, ofreciendo una gran cantidad de contribuciones en GitHub.

Gracias a TensorFlow la implementación de modelos en la disciplina de aprendizaje automático es mucho menos compleja de lo que era históricamente. Se ha facilitado enormemente el proceso de adquisición de datos, el entrenamiento de modelos, la realización de predicciones y el perfeccionamiento de los resultados futuros.

En 2011, Google Brain desarrolló una biblioteca de *Machine Learning* propia para uso interno de Google, llamada DistBelief. Ésta se utilizó sobre todo para los negocios principales de Google, como la búsqueda y anuncios. En 2015, para acelerar los avances en Inteligencia Artificial, Google decidió lanzar TensorFlow como una biblioteca de código abierto, apareció entonces TensorFlow Beta y en 2016, Google anunció las Unidades de Procesamiento de Tensores (TPU), estos tensores son los pilares fundamentales de las aplicaciones de TensorFlow y las TPU son especialmente diseñadas para las operaciones de *Deep Learning*.

En febrero de 2017 se lanzó TensorFlow 1.0, marcando un hito ya que esta versión define la API pública con una capacidad de producción estable. Viendo los rápidos avances en las tecnologías móviles el equipo de TensorFlow anunció en mayo de 2017, TensorFlow Lite; una biblioteca para el desarrollo del *Machine Learning* en dispositivos móviles. Finalmente, en diciembre de 2017, Google presentó KubeFlow, una plataforma de código abierto que permite operar y desplegar modelos TensorFlow en *"Kubernetes"*.

En marzo de 2018, Google anunció TensorFlow.js 1.0, que permite a los desarrolladores implementar modelos de *Machine Learning* utilizando JavaScript. En julio de 2018, Google lanzó TPU Edge, diseñado para ejecutar modelos usando TensorFlow Lite en los smatphones.

Fue en septiembre de 2019 cuando se produjo el lanzamiento oficial de TensorFlow 2.0, la versión principal actual, aunque previamente, en mayo de ese mismo año apareció TensorFlow Graphics para abordar cuestiones relacionadas con el renderizado gráfico y el modelado 3D.

Con la versión 2.0, la cual agilizó muchos de los inconvenientes de la construcción de redes neuronales, TensorFlow finalmente adoptó a Keras como la principal API oficial de alto nivel para construir, entrenar y evaluar redes neuronales. También se mejoraron las herramientas de carga y procesamiento de datos y se proporcionaron nuevas características.

2.2.2. Funcionamiento

TensorFlow permite a los desarrolladores crear gráficos de flujo de datos, estructuras que describen cómo los datos se mueven a través de un gráfico. Cada nodo del gráfico representa una operación matemática y cada conexión o arista entre nodos es una matriz de datos multidimensional o tensor.

TensorFlow proporciona todo al programador a través del lenguaje Python, que proporciona formas convenientes de expresar cómo las abstracciones de alto nivel pueden ser acopladas. Los nodos y los tensores en TensorFlow son objetos de Python y las aplicaciones de TensorFlow son a su vez aplicaciones de Python.

Las bibliotecas de transformaciones que están disponibles a través de TensorFlow están escritas como binarios C++ de alto rendimiento. Python solo dirige el tráfico entre las piezas y proporciona abstracciones de programación de alto nivel para conectarlas entre sí.

Las aplicaciones de TensorFlow se pueden ejecutar en casi cualquier objetivo que sea conveniente, como dispositivos iOS y Android o CPUs y GPUs. Si se utiliza en la propia nube de Google, se puede ejecutar TensorFlow en la Unidad de Procesamiento (TPU) de Google para una mayor aceleración. Sin embargo, los modelos resultantes creados por TensorFlow pueden desplegarse en la mayoría de los dispositivos en los que se utilizan para realizar predicciones.

TensorFlow se inspira principalmente en su uso para clasificar, descubrir predicciones, identificar patrones y aplicar percepciones. Se ha utilizado en aplicaciones de *Machine Learning* y en la parte de producción de Google para desarrollar una solución optimizada; aplicaciones como el cuidado de la salud, los productos de Google, los medios sociales y los anuncios.

El software TensorFlow sigue actualizándose y tiene un rápido crecimiento en los próximos años, considerándose como el futuro de la modelización de *Machine Learning*.

Las bibliotecas auxiliares de TensorFlow ayudan a depurar y visualizar los modelos. El mayor beneficio que ofrecen es la abstracción, en lugar de ocuparse de los detalles de implementación de los algoritmos, el desarrollador puede centrarse en la lógica general de la aplicación.

2.2.3. Uso de Tensorflow desde R

Para la parte práctica de este trabajo implementaremos el portal de Tensorflow en R. Para ello necesitaremos instalar dos paquetes principales con sus respectivas librerías: Tensorflow y Keras.

Entre las ventajas del uso de Tensorflow mediante R destaca el hecho de poder cargar sets de datos de grandes dimensiones, convirtiéndolo en una gran librería de computación, y la existencia de numerosos algoritmos de optimización incorporados que no requieren que todos los datos se encuentren en la RAM.

Estamos acostumbrados a que, para poder trabajar con algoritmos de optimización, tenemos que gestionar la situación de modo que todo el conjunto de datos se halle en la RAM. Sin embargo, ahora podemos tener un conjunto de datos que ocupen 10 gigabytes, por ejemplo, otorgar a la construcción de nuestro modelo pequeñas muestras a la vez y garantizar que la optimización funcionará correctamente.

2.2.4. Tensores

El término al que más se hace referencia en este tipo de modelados es “tensor”. Los tensores no dejan de ser vectores o matrices multidimensionales con los que hemos trabajado habitualmente y será nuestra forma de almacenar los datos.

Los tensores 4D son los que nos resultarán de interés en nuestro trabajo, ya que es la forma de agrupar datos de imágenes. Aunque podamos pensar en una imagen como un tensor 3D teniendo en cuenta la altura, anchura y profundidad (que son los canales de color rojo, verde y azul,“RGB”); pero si consideras cada una de las muestras, terminas teniendo un tensor de cuatro dimensiones.

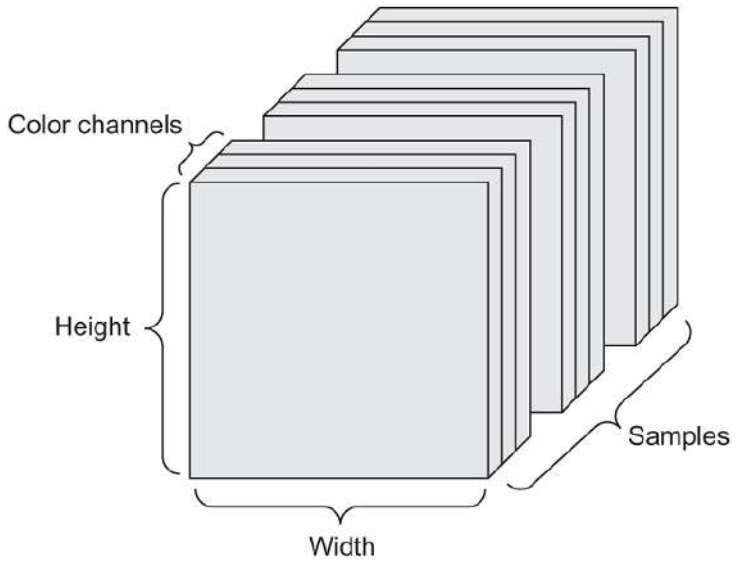


Figura 2.1: Imagen como tensor 4D

2.2.5. ¿Cómo se ejecuta realmente Tensorflow?

Los programas con Tensorflow no son un simple *script* de R que se compila, sino que se construye un grafo de flujo de datos que funciona como una especie de matriz de adición o multiplicación de un término de sesgo, tomando gradientes o aplicando algún tipo de optimización. En definitiva, se construyen funciones que operan con los datos juntándolos todos dentro de un grafo.

Trabajando con Tensorflow el usuario no programa directamente este grafo, sino que desarrolla un modelo en código R y el grafo se genera automáticamente de dicho código.

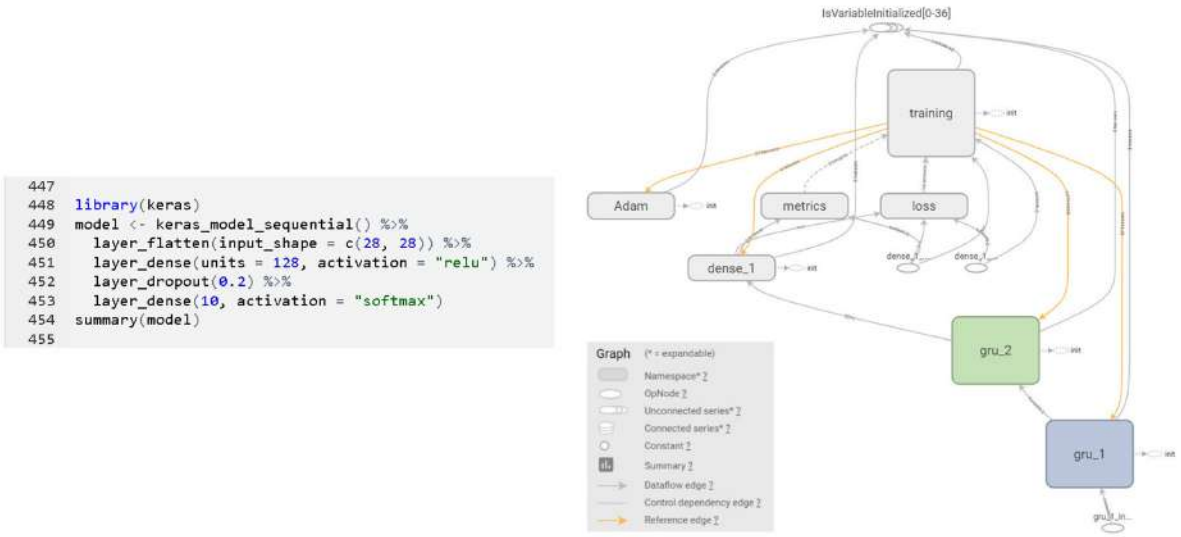


Figura 2.2: Grafo generado a partir de código R

2.2.6. Librería *Keras*

La librería *Keras* nos será de gran utilidad en R ya que, nos permite preprocesar los datos de entrada remodelándolos, escalándolos y tranformándolos en tensores.

Posteriormente podremos definir propiamente nuestro modelo de red, lo que básicamente nos dice cuáles son nuestras capas y cómo se van a comportar. Una vez definido, procedemos a compilar el modelo, lo que nos indicará cuál es la función de pérdida y de optimización que usaremos durante el entrenamiento.

La piedra angular de nuestro trabajo con keras se resume por tanto en dos principales actividades: definir capas y compilarlas.

2.2.7. Fase de entrenamiento del modelo de red neuronal

El proceso de entrenamiento se realiza mediante la función `fit` donde, con la información preprocesada, el modelo irá aprendiendo de manera iterativa para luego validarlo intentando disminuir el sobreajuste, es decir, intentando que el modelo no se limite únicamente a memorizar los datos para que nos dé como resultado una función poco útil ya que sólo sera válida para dichos datos en concreto.

A medida que la red va aprendiendo, podremos observar como el propio procedimiento, la precisión y la pérdida mejoran.

2.2.8. Fase de evaluación y predicción

Una vez se haya definido y entrenado el modelo, para cerciorarnos de que todo ha funcionado correctamente, procederemos a evaluar la red, lo cual consiste en utilizar un conjunto de datos que se habría mantenido al margen en un principio y con el que el modelo nunca hubiese tenido contacto, fuese un conjunto de datos "nuevo" para la red. Es en ese momento donde se realiza el test final de validación y obtendríamos los resultados de cómo de buenas pueden ser las predicciones que se realicen generalizando en imágenes distintas.

Capítulo 3

Construcción de un modelo con tensorflow desde R para clasificar imágenes

3.1. Conjunto de datos de Moda MNIST

Para este ejemplo, entrenaremos una red neuronal para la clasificación de imágenes de ropa. En primer lugar, importaremos un dataset compuesto de 70.000 imágenes en escala de grises y divididas según el tipo de prenda que sea. Estas prendas individuales son imágenes de 28x28 píxeles.

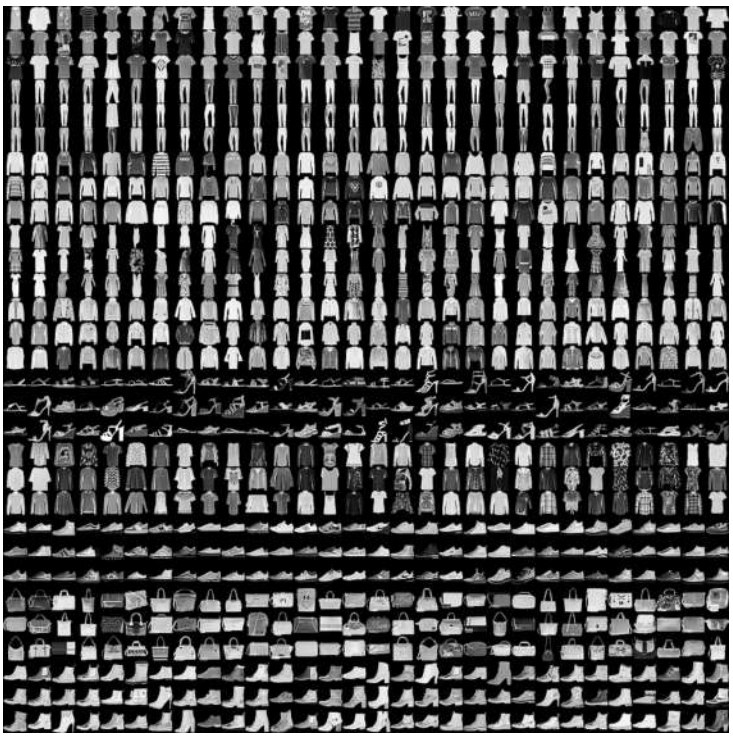


Figura 3.1: Muestra de imágenes del conjunto Fashion-MNIST

De las 70.000 imágenes importadas, destinaremos 60.000 para el entrenamiento y apren-

dizaje de la red y las 10.000 restantes para la evaluación final de la red y medir la precisión real del algoritmo a la hora de clasificar las imágenes.

```
library(tensorflow)
library(keras)

fashion_mnist <- dataset_fashion_mnist()

c(train_images, train_labels) %<-% fashion_mnist$train
c(test_images, test_labels) %<-% fashion_mnist$test
```

Al cargar el set de datos "fashion_mnist" recibimos cuatro listas, tanto las imágenes (`train$x` y `test$x`) como las etiquetas (`train$y` y `test$y`) de entrenamiento y test. Con esta orden separamos los datos referidos al entrenamiento y los datos referidos al test, guardándolos en un nuevo "array" que une los valores almacenados en `x` e `y`, bajo el nombre `train_images` y `train_labels` para el entrenamiento y `test_images` y `test_labels` para el test, haciéndolo así más intuitivo y fácil de localizar.

```
dim(train_images)
```

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

```
dim(test_images)
```

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

```
train_labels[1:20]
```

```
##  [1] 9 0 0 3 0 2 7 2 5 5 0 9 5 5 7 9 1 0 6 4
```

Como decíamos, las imágenes se pueden ver como una matriz de datos de 28x28 píxeles, donde cada píxel viene expresado en una escala de 0 a 255. Esto se debe a que cada píxel de una imagen almacena la información de su tono o luminosidad, donde el tono negro es el valor 0 y el valor blanco es el más alto (normalmente 255 en escala de grises).

Ya que las imágenes del dataset están divididas en 10 categorías, asociamos a cada prenda un dígito del 0 al 9 que usaremos como etiqueta:

Dígito	Prenda
0	Camiseta
1	Pantalón
2	Jersey
3	Vestido
4	Abrigo
5	Sandalia

Dígito	Prenda
6	Camisa
7	Zapatilla de deporte
8	Bolso
9	Bota

Creamos un vector donde almacenamos el nombre de cada tipo de ropa.

```
class_names = c('Camiseta',
                'Pantalón',
                'Jersey',
                'Vestido',
                'Abrigo',
                'Sandalia',
                'Camisa',
                'Zapatilla de deporte',
                'Bolso',
                'Bota')
```

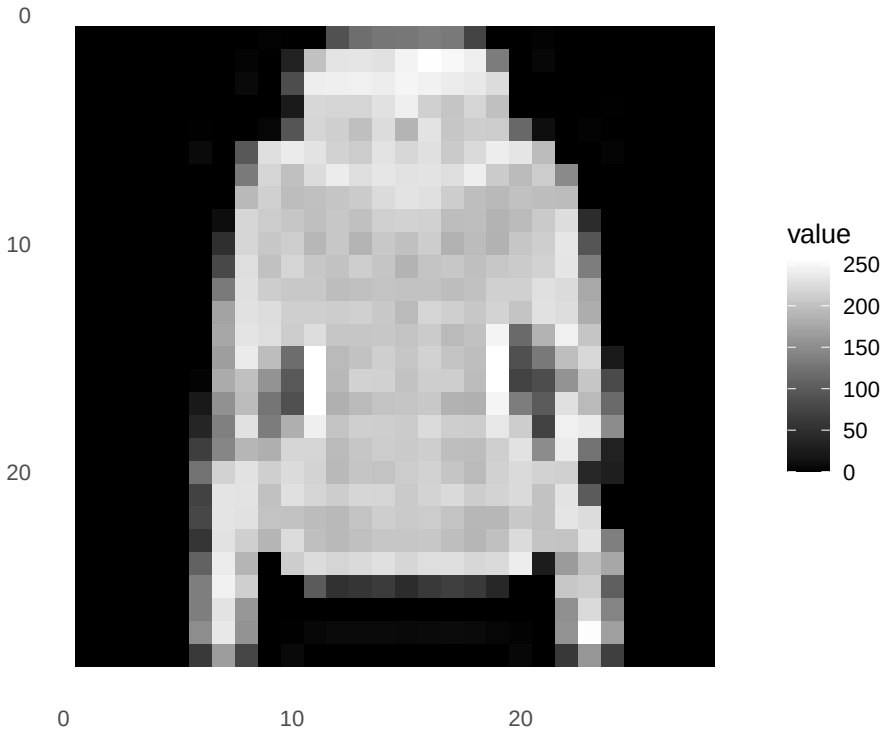
3.1.1. Preprocesamiento de los datos

Antes de trabajar con nuestra red, inspeccionamos una imagen cualquiera del conjunto de datos de entrenamiento. Establecemos una semilla para obtener una penda mediante un muestreo entre las 60.000 imágenes de entrenamiento con la orden `sample` y la representamos con una escala desde el 0 para el negro hasta el 255 para el blanco.

```
library(tidyr)
library(ggplot2)
set.seed(1234567)

image_1 <- as.data.frame(train_images[sample(1:60000,1), , ])
colnames(image_1) <- seq_len(ncol(image_1))
image_1$y <- seq_len(nrow(image_1))
image_1 <- gather(image_1, "x", "value", -y)
image_1$x <- as.integer(image_1$x)

ggplot(image_1, aes(x = x, y = y, fill = value)) +
  geom_tile() +
  scale_fill_gradient(low = "black", high = "white", na.value = NA) +
  scale_y_reverse() +
  theme_minimal() +
  theme(panel.grid = element_blank())   +
  theme(aspect.ratio = 1) +
  xlab("") +
  ylab("")
```

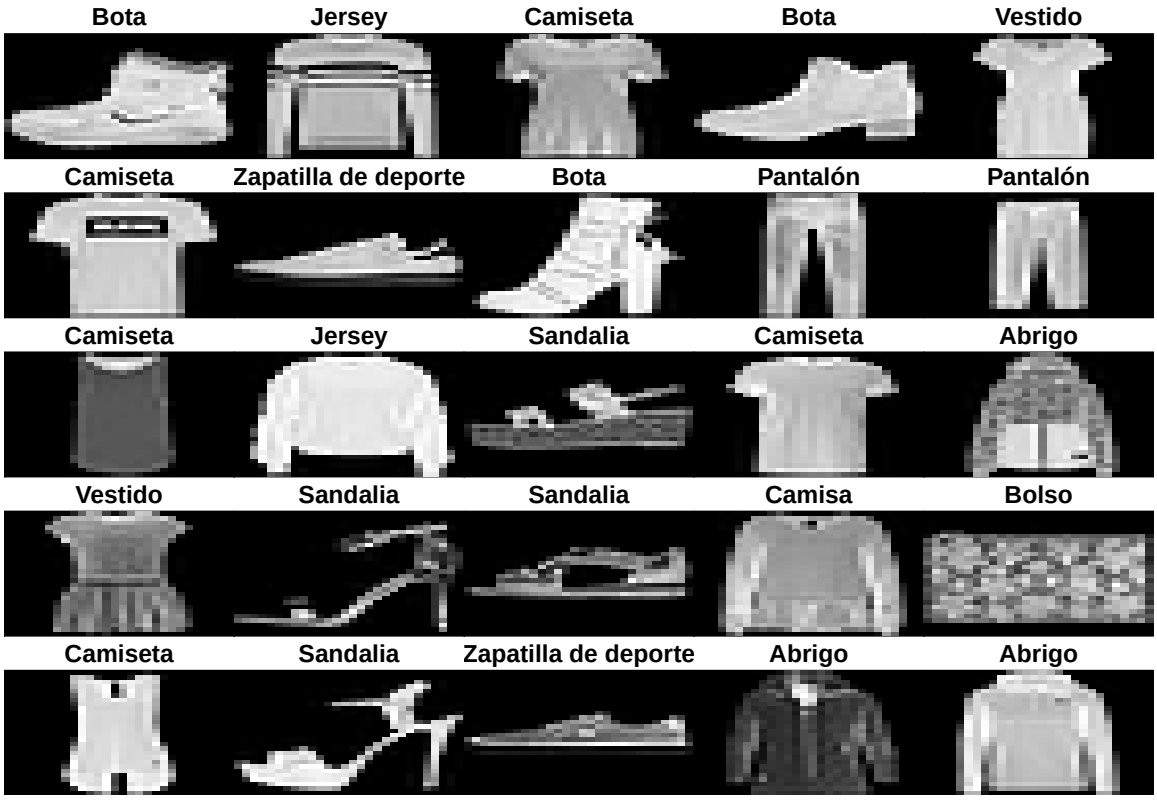


Vemos como el rango de valores de los píxeles van desde el 0 hasta el 255. Aunque estos valores de píxeles se pueden presentar directamente en los modelos de redes neuronales en su formato sin procesar, esto puede resultar ineficiente y lento en el modelado. En cambio, puede resultar muy beneficioso preparar los valores de los píxeles de la imagen antes del modelado, simplemente escalando los valores al rango 0-1 para centrar y estandarizar los valores. Esto se puede lograr dividiendo todos los valores de píxeles por el valor de píxel más grande; es decir 255, tanto en el conjunto de entrenamiento como en el conjunto test.

```
train_images <- train_images / 255
test_images <- test_images / 255
```

Mostramos las 25 primeras imágenes de nuestro set de datos, comprobando que la información es correcta y que las etiquetas se corresponden con su prenda asociada.

```
par(mfcol=c(5,5))
par(mar=c(0, 0, 1.5, 0), xaxs='i', yaxs='i')
for (i in 1:25) {
  img <- train_images[i, , ]
  img <- t(apply(img, 2, rev))
  image(1:28, 1:28, img, col = gray((0:255)/255), xaxt = 'n', yaxt = 'n',
        main = paste(class_names[train_labels[i] + 1]))
}
```



3.1.2. Construcción del modelo

3.1.2.1. Configuración de las capas

El componente básico de una red neuronal es la capa. Las capas extraen representaciones de los datos que se introducen en ellas.

La mayor parte del aprendizaje profundo consiste en encadenar capas simples. La mayoría de las capas tienen parámetros que se aprenden durante el entrenamiento.

```
model <- keras_model_sequential()
model %>%
  layer_flatten(input_shape = c(28, 28)) %>%
  layer_dense(units = 128, activation = 'relu') %>%
  layer_dense(units = 10, activation = 'softmax')
```

La primera capa del modelo transforma el formato de las imágenes en un vector de dos dimensiones de 28 por 28 = 784 píxeles, lo que denominamos capa de aplanamiento.

Después de que los píxeles sean "aplanados", la red la conforman dos capas densas. La primera capa densa tiene 128 neuronas. La segunda (y última) capa es una capa softmax de 10 nodos; devuelve una matriz de 10 puntuaciones de probabilidad que suman 1. Cada nodo contiene una puntuación que indica la probabilidad de que la imagen actual pertenezca a una de las clases de 10 dígitos.

3.1.2.2. Compilación del modelo

Antes de que el modelo esté listo para el entrenamiento, necesita algunos ajustes más. Estos se agregan durante el paso de compilación del modelo:

- Función de pérdida: mide la precisión del modelo durante el entrenamiento. Queremos minimizar esta función para "dirigir" el modelo en la dirección correcta.
- Optimizador: así es como se actualiza el modelo en función de los datos que ve y su función de pérdida.
- Métricas: se utilizan para monitorear los pasos de entrenamiento y prueba. Usamos para ello la precisión, es decir, la fracción de imágenes que se clasifican correctamente.

```
model %>% compile(
  optimizer = 'adam',
  loss = 'sparse_categorical_crossentropy',
  metrics = c('accuracy')
)
```

3.1.2.3. Entrenamiento del modelo

Para entrenar el modelo de red neuronal hacemos lo siguiente:

- Introducimos el conjunto de datos de entrenamiento, preprocesado previamente, tanto los vectores de imágenes como los vectores de etiquetas.
- El modelo aprende a asociar las etiquetas a las imágenes.
- Realizamos predicciones sobre el conjunto test de datos. Verificamos que las predicciones se corresponden con las etiquetas teóricas.

Establecemos primeramente 5 periodos de entrenamiento sobre los 60.000 datos de entrenamiento, mostrando para cada etapa la función de pérdida, la precisión y el tiempo que invierte la muestra en cada etapa.

```
model %>% fit(train_images, train_labels, epochs = 5, verbose = 2)
```

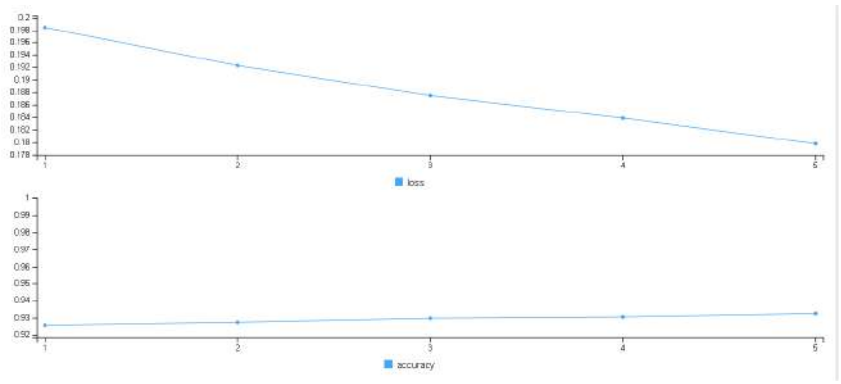


Figura 3.2: Gráfico que muestra la disminución de la función de pérdida y el aumento en la precisión de la red

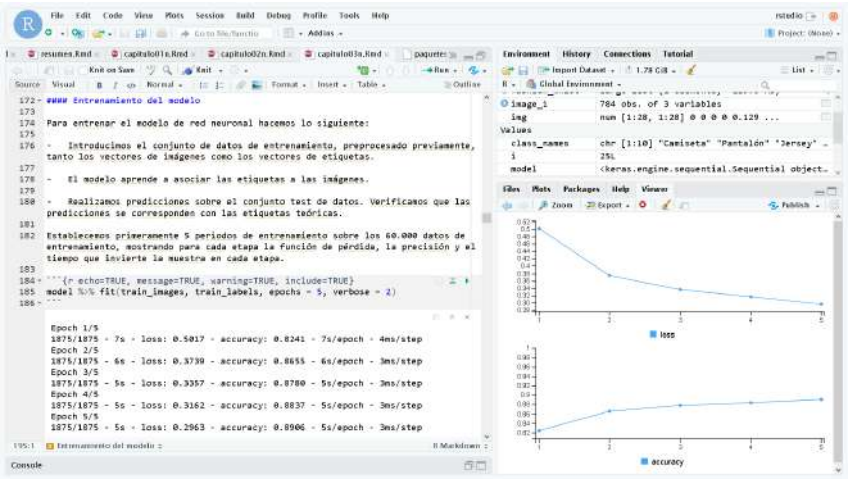


Figura 3.3: Captura del entrenamiento del modelo en cada etapa y resultados. Fuente: Elaboración propia

A medida que el modelo se entrena, la función de pérdida disminuye y la precisión aumenta. En este ejemplo alcanzamos una precisión de 0.89 (que equivale a un 89 %). El tiempo transcurrido en cada periodo es de unos 6 segundos, alcanzando entonces los 30 segundos totales.

```
model %>% fit(train_images, train_labels, epochs = 10, verbose = 2)
```

Si empleamos ahora 10 etapas para entrenar el modelo vemos como el tiempo total transcurrido aumenta proporcionalemente hasta los 60 segundos.

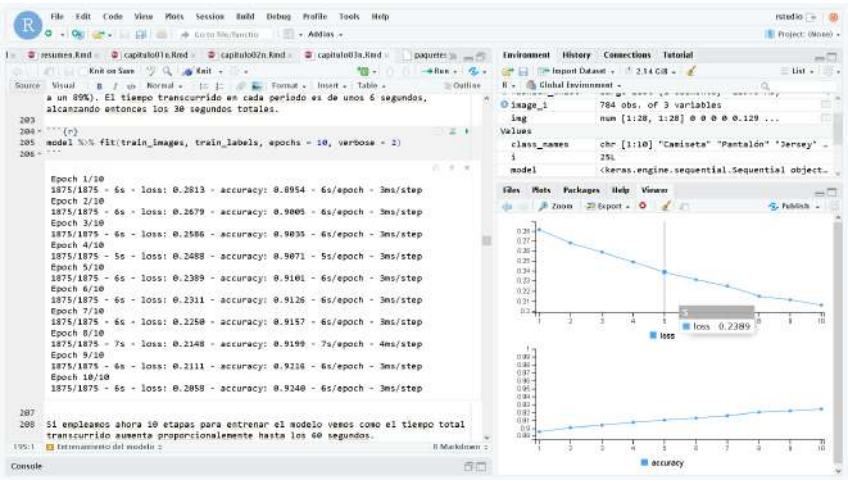


Figura 3.4: Mayor tiempo de compilación cuando se introducen más etapas. Fuente: Elaboración propia

Aunque, como vemos en la Figura 3.2, un aumento en el número de etapas mejora los resultados de pérdida y precisión, también hace que el entrenamiento del modelo sea más lento e ineficiente. Es por ello que tenemos que encontrar el punto de equilibrio óptimo entre eficiencia y fiabilidad.

3.1.2.4. Evaluación de la precisión

Comparamos ahora los resultados, evaluando el modelo sobre el conjunto test.

```
score <- model %>% evaluate(test_images, test_labels, verbose = 0)
score
```

```
##      loss  accuracy
## 0.3429737 0.8825000
```

Observamos como la precisión es algo menor cuando la red actúa sobre los datos test. Esta pequeña diferencia se debe a un concepto llamado "sobreajuste", por el cual el modelo está tan ajustado a los datos de entrenamiento que luego complica el hecho de generalizar la evaluación sobre los datos de test.

Aunque no es una disparidad llamativa o preocupante ya que nos encontramos ante un modelo de red neuronal simple, es cierto que en modelos más complejos se tiende más a este sobreajuste ya que el algoritmo aprende ciertas características específicas pero no patrones más generales.

Un problema de sobreajuste también puede venir dado por disponer de pocos datos sobre los que actuar.

3.1.2.5. Realización de predicciones

Una vez que tenemos el modelo entrenado, podemos usarlo para realizar predicciones sobre las imágenes.

```
predictions <- model %>% predict(test_images)
predictions[1, ]
```

```
##  [1] 2.273699e-09 4.145728e-12 2.736063e-09 3.925237e-13 3.763194e-10
##  [6] 5.049190e-04 3.090618e-09 3.017159e-02 8.112063e-09 9.693235e-01
```

Una predicción es un vector de 10 números que miden la fiabilidad del modelo de que la imagen en cuestión pertenezca a uno de los 10 artículos de ropa.

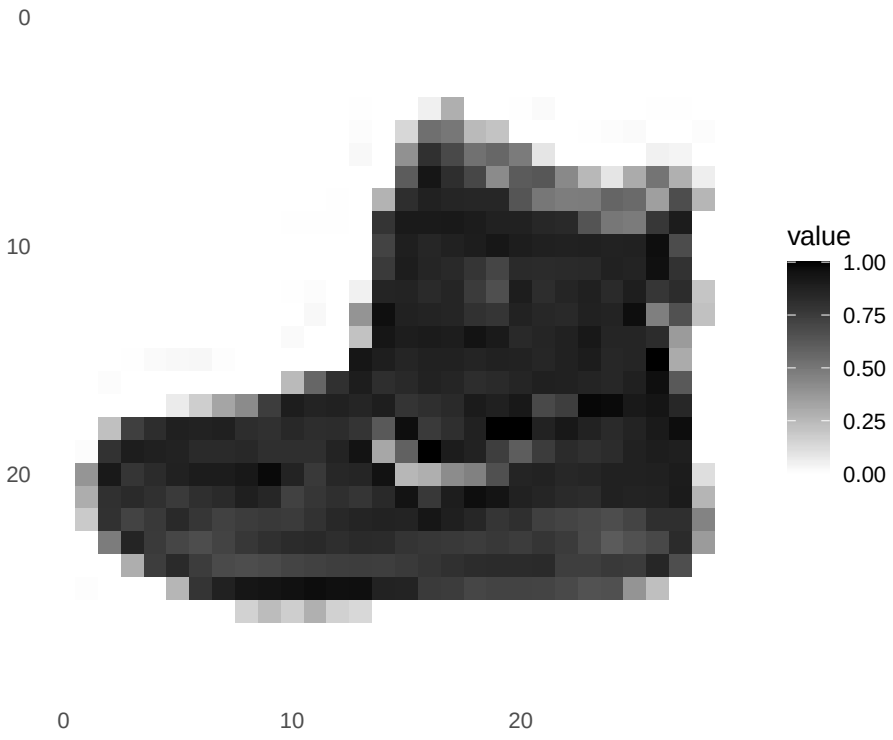
```
which.max(predictions[1, ])
```

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

```
class_names[which.max(predictions[1, ])]
```

```
## [1] "Bota"
```

```
image_1 <- as.data.frame(train_images[1, , ])
colnames(image_1) <- seq_len(ncol(image_1))
image_1$y <- seq_len(nrow(image_1))
image_1 <- gather(image_1, "x", "value", -y)
image_1$x <- as.integer(image_1$x)
ggplot(image_1, aes(x = x, y = y, fill = value)) +
  geom_tile() +
  scale_fill_gradient(low = "white", high = "black", na.value = NA) +
  scale_y_reverse() +
  theme_minimal() +
  theme(panel.grid = element_blank())   +
  theme(aspect.ratio = 1) +
  xlab("") +
  ylab("")
```



En este caso, la predicción de la primera prenda se corresponde con el artículo codificado con el número 10, es decir, con la bota.

Proyectemos ahora algunas imágenes con sus predicciones realizadas. Las predicciones correctas aparecen escritas en verde, mientras que en rojo aparecen aquellas que han sido confusas para la red y no ha podido predecirlas de manera correcta.

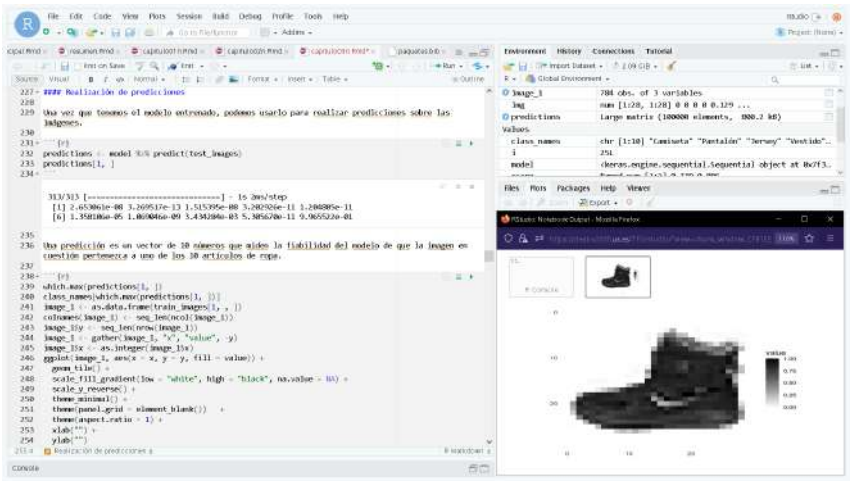
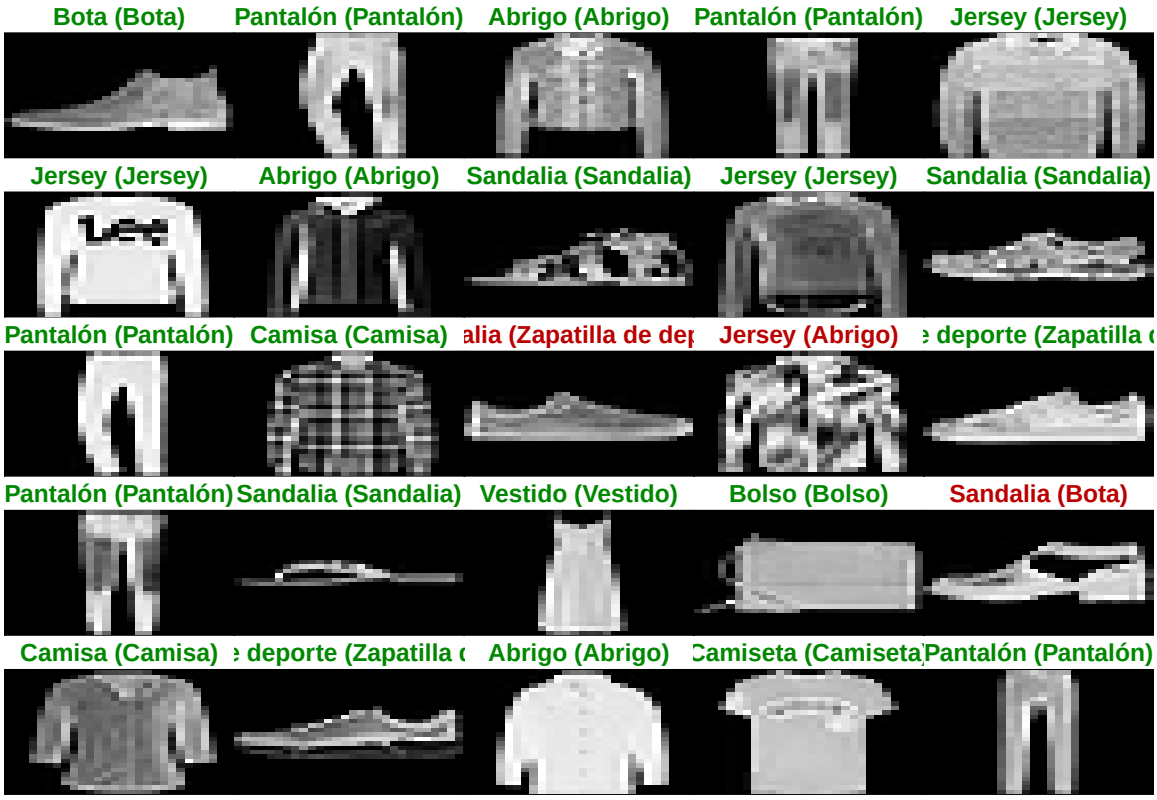


Figura 3.5: Captura del código de R en funcionamiento y predicción. Fuente: Elaboración propia

```
par(mfcol=c(5,5))
par(mar=c(0, 0, 1.5, 0), xaxs='i', yaxs='i')
for (i in 1:25) {
```

```
  img <- test_images[i, , ]
  img <- t(apply(img, 2, rev))
  # subtract 1 as labels go from 0 to 9
  predicted_label <- which.max(predictions[i, ]) - 1
  true_label <- test_labels[i]
  if (predicted_label == true_label) {
    color <- '#008800'
  } else {
    color <- '#bb0000'
  }
  image(1:28, 1:28, img, col = gray((0:255)/255), xaxt = 'n', yaxt = 'n',
        main = paste0(class_names[predicted_label + 1], " (",
                      class_names[true_label + 1], ")"),
        col.main = color)
}
```



Vemos como aparece la etiqueta a la que se ha predecido y entre paréntesis el verdadero artículo que es. Aunque se han producido ciertos errores, debido a que la precisión no era perfceta, observamos que estos han sido por la confusión de dos prendas bastante similares, difícil de apreciar hasta para el ojo humano. Esto es, predicción de sandalia cuando en realidad se trata de una bota o una zapatilla de deporte y predicción de jersey cuando el verdadero artículo es un abrigo.

3.2. Set de imágenes CIFAR-10

Para este ejemplo realizaremos un procedimiento similar al anterior mediante una red neuronal convolucional y utilizando el dataset **CIFAR-10** (Canadian Institute For Advanced Research), una colección de imágenes que son usualmente usadas para entrenar los algoritmos de aprendizaje automático y visión por ordenador. Contiene 60.000 imágenes a color de 32x32 píxeles divididas en 10 diferentes clases mutuamente excluyentes: aviones, coches, pájaros, gatos, ciervos, perros, ranas, caballos, barcos y camiones; habiendo 6.000 imágenes de cada clase. Estas imágenes se encuentran divididas en 50.000 imágenes de entrenamiento y 10.000 imágenes test.

```
cifar <- dataset_cifar10()
```

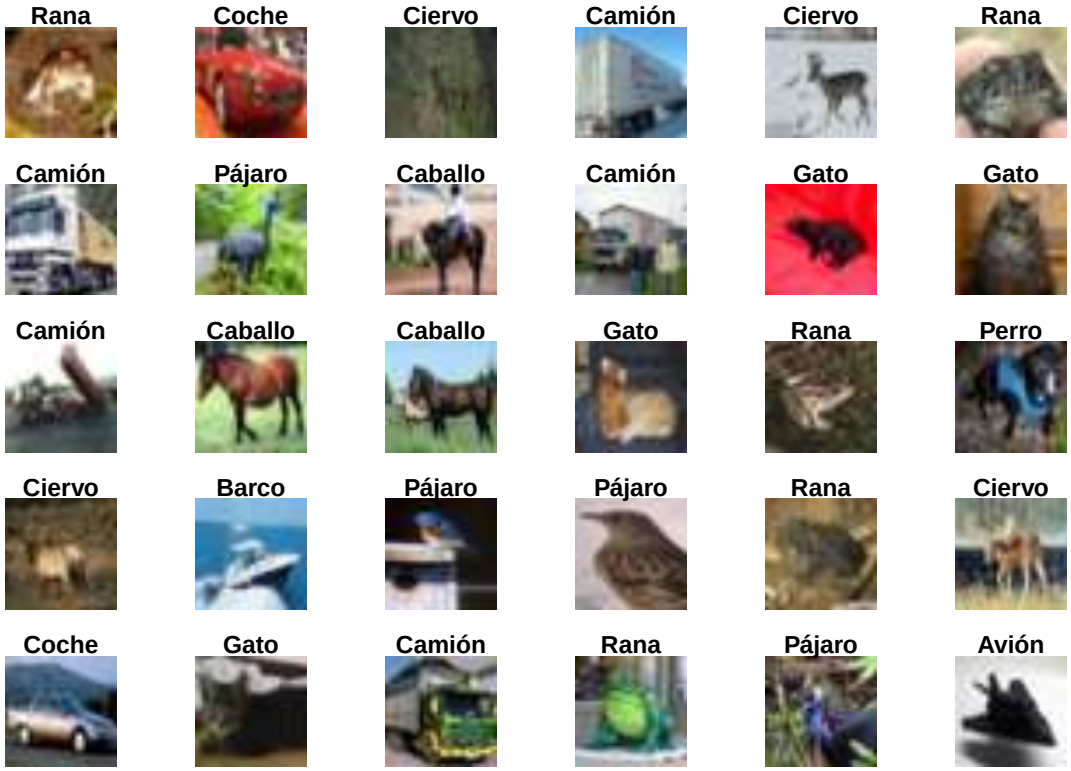
3.2.1. Proyección de los datos

Mostremos las 25 primeras imágenes del set, donde aparece el nombre de la clase a la que pertenece cada una.

```
class_names <- c('Avión', 'Coche', 'Pájaro', 'Gato', 'Ciervo',
                 'Perro', 'Rana', 'Caballo', 'Barco', 'Camión')

index <- 1:30

par(mfcol = c(5,6), mar = rep(1, 4), oma = rep(0.2, 4))
cifar$train$x[index,,,] %>%
  purrr::array_tree(1) %>%
  purrr::set_names(class_names[cifar$train$y[index] + 1]) %>%
  purrr::map(as.raster, max = 255) %>%
  purrr::iwalk(~{plot(.x); title(.y)})
```



3.2.2. Creación de la red convolucional

Como valores de entrada, la red neuronal convolucional usa *tensores de forma*, la altura, anchura y canales de color de la imagen (R,G,B). Estos tensores serán representados como vectores tridimensionales e introducidos a la primera capa.

```
model <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3,3), activation = "relu",
                input_shape = c(32,32,3)) %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3,3), activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2,2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3,3), activation = "relu")
```

La base de la red está formada por una sucesión de Capas de Convolución (Conv2D) y Agrupación Máxima (MaxPooling2D).

La capa de convolución crea un núcleo que se convoluciona con la entrada de la capa para producir un tensor de salidas.

En el procesamiento de imágenes, el kernel es una matriz de convolución que se puede usar para desenfocar, agudizar, realzar o detectar bordes al hacer una convolución.

Por otro lado, la operación de agrupación máxima para datos espaciales reduce la muestra de la entrada a lo largo de sus dimensiones espaciales (alto y ancho) tomando el valor máximo sobre una ventana de entrada para cada canal. Esta ventana se desplaza a lo largo de cada dimensión.

Si exponemos la arquitectura del modelo,

```
summary(model)
```

```
## Model: "sequential_1"
## _________________________________________________________________________
##  Layer (type)                       Output Shape                    Param #
## =========================================================================
##  conv2d_2 (Conv2D)                  (None, 30, 30, 32)              896
##  max_pooling2d_1 (MaxPooling2D)     (None, 15, 15, 32)              0
##  conv2d_1 (Conv2D)                  (None, 13, 13, 64)              18496
##  max_pooling2d (MaxPooling2D)       (None, 6, 6, 64)                0
##  conv2d (Conv2D)                    (None, 4, 4, 64)                36928
## =========================================================================
## Total params: 56,320
## Trainable params: 56,320
## Non-trainable params: 0
## _________________________________________________________________________
```

Se puede ver que la salida de cada capa Conv2D y MaxPooling2D es un tensor de forma 3D (alto, ancho, canales). Las dimensiones de ancho y alto tienden a reducirse a medida que se profundiza en la red. El número de canales de salida para cada capa Conv2D está controlado por el primer argumento (por ejemplo, 32 o 64). Por lo general, a medida que se reducen el ancho y el alto, puede permitirse (computacionalmente) agregar más canales de salida en cada capa Conv2D.

3.2.3. Agregar capas densas

Para completar nuestro modelo, se transformará el último tensor de salida de la base convolucional en una o más capas densas para realizar la clasificación. Las capas densas toman vectores como entrada de una dimensión, mientras que la salida actual es un tensor 3D. Primero, aplanará la salida 3D a 1D, luego agregará una o más capas densas en la parte superior. CIFAR tiene 10 clases de salida, por lo que usa una capa densa final con 10 salidas y una activación softmax.

```
model %>%
  layer_flatten() %>%
  layer_dense(units = 64, activation = "relu") %>%
  layer_dense(units = 10, activation = "softmax")
```

Mostramos ahora la arquitectura completa del modelo

```
summary(model)
```

```
## Model: "sequential_1"
## _________________________________________________________________________
```

```
##  Layer (type)                        Output Shape                    Param #
## ================================================================================
##  conv2d_2 (Conv2D)                   (None, 30, 30, 32)              896
##  max_pooling2d_1 (MaxPooling2D)      (None, 15, 15, 32)              0
##  conv2d_1 (Conv2D)                   (None, 13, 13, 64)              18496
##  max_pooling2d (MaxPooling2D)        (None, 6, 6, 64)                0
##  conv2d (Conv2D)                     (None, 4, 4, 64)                36928
##  flatten_1 (Flatten)                 (None, 1024)                    0
##  dense_3 (Dense)                     (None, 64)                      65600
##  dense_2 (Dense)                     (None, 10)                      650
## ================================================================================
## Total params: 122,570
## Trainable params: 122,570
## Non-trainable params: 0
## --------------------------------------------------------------------------------
```

Observamos como las salidas de vectores 3D se han aplanado en vectores unidimensionales antes de pasar por dos capas densas.

3.2.4. Compilación y entrenamiento del modelo

Al igual que en el ejemplo anterior haremos una comparación de los tiempos de computación al ir modificando el número de etapas en el entrenamiento de la red.

Comenzando con 5 periodos:

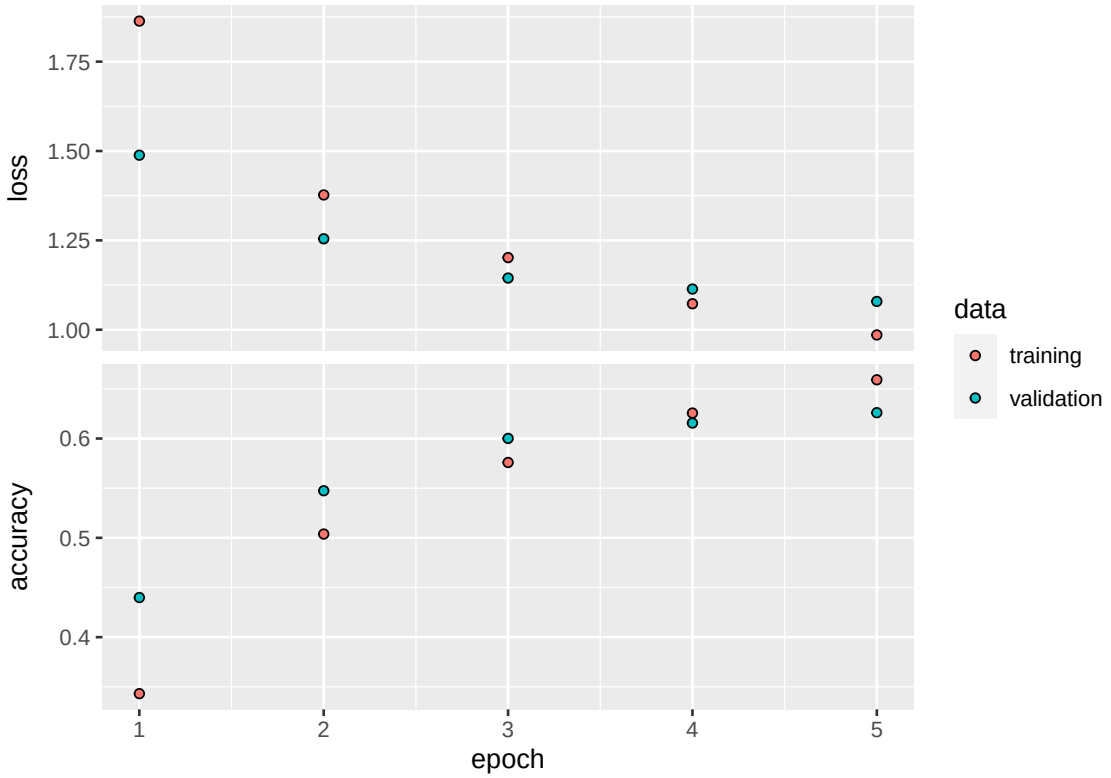
```
model %>% compile(
  optimizer = "adam",
  loss = "sparse_categorical_crossentropy",
  metrics = "accuracy"
)

history <- model %>%
  fit(
    x = cifar$train$x, y = cifar$train$y,
    epochs = 5,
    validation_data = unname(cifar$test),
    verbose = 2
  )
```

En este caso los resultados son más llamativos, alcanzando una media de 95 segundos por etapa y un total de 473 segundos (unos 8 minutos) para el entrenamiento global de la red.

3.2.5. Evaluación del modelo

```
plot(history, type = "l")
```




```
evaluate(model, cifar$test$x, cifar$test$y, verbose = 0)
```

```
##      loss accuracy
## 1.079085 0.626000
```

Alcanzamos una precisión en torno al 60 %, no es muy alta debido a la simplicidad de nuestra red.

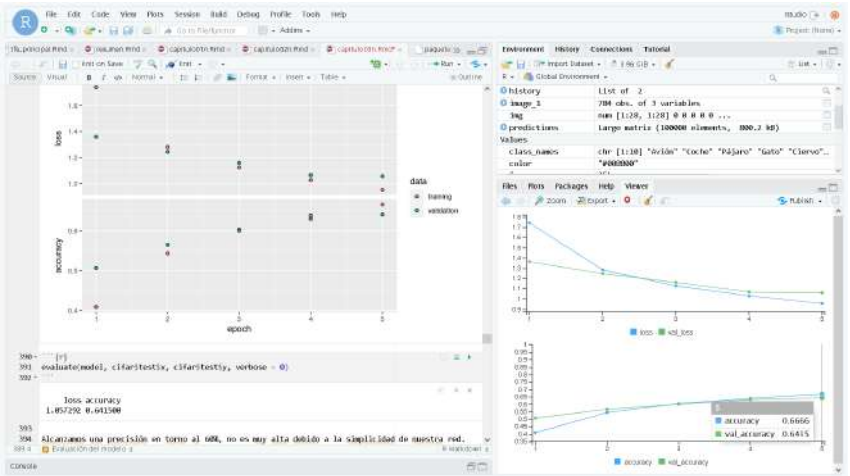


Figura 3.6: Evaluación del modelo desde R y gráfica de resultados. Fuente: Elaboración propia

3.3. Imágenes de números manuscritos

Como último ejemplo vamos a proceder a crear un modelo de red neuronal para el reconocimiento de dígitos manuscritos pertenecientes al dataset MNIST.

Estos dígitos son imágenes de 28x28 píxeles en escala de grises y aparecen divididos en un conjunto de entrenamiento de 60.000 imágenes y un conjunto test de 10.000 imágenes

```
# Paquetes
library(tensorflow)
library(purrr)
library(dplyr)
library(tibble)
library(readr)


# Mnist data set
mnist <- keras::dataset_mnist()
dim(mnist$train$x)
```

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

```
dim(mnist$test$x)
```

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

Estos son algunos ejemplos:

```
# para hacer un grid 12 x 12 imágenes
# 144 en total
imagenes <- 1:144

# Generando el grid de imágenes
imagenes %>%
  # para iterar sobre las primeras n imagenes
  map(
    ~mnist$train$x[.x, , ] %>%
      #transformar a dataframe
      as.data.frame() %>%
      # agregar el ide de la columna
      # para identificar la posición de cada pixel
      rowid_to_column(var = "y") %>%
      # reshape del dataframe
      gather("x", "value", -y) %>%
      # volviendo la variable x numerica
      mutate(x = parse_number(x)) %>%
      # gráfico de tiles
```

```
    ggplot(aes(x = x, y = y, fill = value)) +
    geom_tile(show.legend = FALSE) +
    # reorganizando el eje y
    scale_y_reverse() +
    scale_fill_gradient(low = "white", high = "black") +
    theme_void()
) %>%
cowplot::plot_grid(plotlist = .)
```

Aunque para el ojo humano son dígitos fácilmente distinguibles y sabríamos decir casi con total seguridad de qué número se trata en cada imagen, un ordenador tiene la difícil tarea de encontrar una serie patrones que los permita diferenciar con un porcentaje de precisión relativamente alto.

3.3.1. Preparación de los datos

Los valores de la variable `x` lo conforman vectores 3-d `(imágenes, ancho, alto)` que posteriormente serán "aplanados" a una sola dimensión, transformando los 28x28 píxeles en un vector de 784 píxeles de longitud.

Al igual que en modelos anteriores, necistamos reducir el conjunto de píxeles a una escala de 0 a 1 para facilitar el trabajo y estandarizar resultados.

```
# Cambiando la intensidad de los pixeles
# de una escala de 0 a 255 a una de 0 a 1
```

```
mnist$train$x <- mnist$train$x/255
mnist$test$x <- mnist$test$x/255
```

3.3.2. Definición del modelo

Estructuramos el modelo mediante la organziación de capas, utilizando el modelo secuencial como el más sencillo, mediante una sucesión lineal de capas.

```
library(keras)

model <- keras_model_sequential() %>%
  layer_flatten(input_shape = c(28, 28)) %>%
  layer_dense(units = 128, activation = "relu") %>%
  layer_dropout(0.2) %>% #Las entradas que no se establecen en 0 se escalan en
              #1/(1 - tasa) de modo que la suma de todas las entradas no cambia.
  layer_dense(10, activation = "softmax")
```

```
summary(model)
```

```
## Model: "sequential_2"
## ________________________________________________________________________________
##  Layer (type)                       Output Shape                    Param #
## ================================================================================
##  flatten_2 (Flatten)                (None, 784)                     0
##  dense_5 (Dense)                    (None, 128)                     100480
##  dropout (Dropout)                  (None, 128)                     0
##  dense_4 (Dense)                    (None, 10)                      1290
## ================================================================================
## Total params: 101,770
## Trainable params: 101,770
## Non-trainable params: 0
## ________________________________________________________________________________
```

Pasamos ahora a compilar el modelo con su correspondiente función de pérdida, optimización y métrica correspondiente.

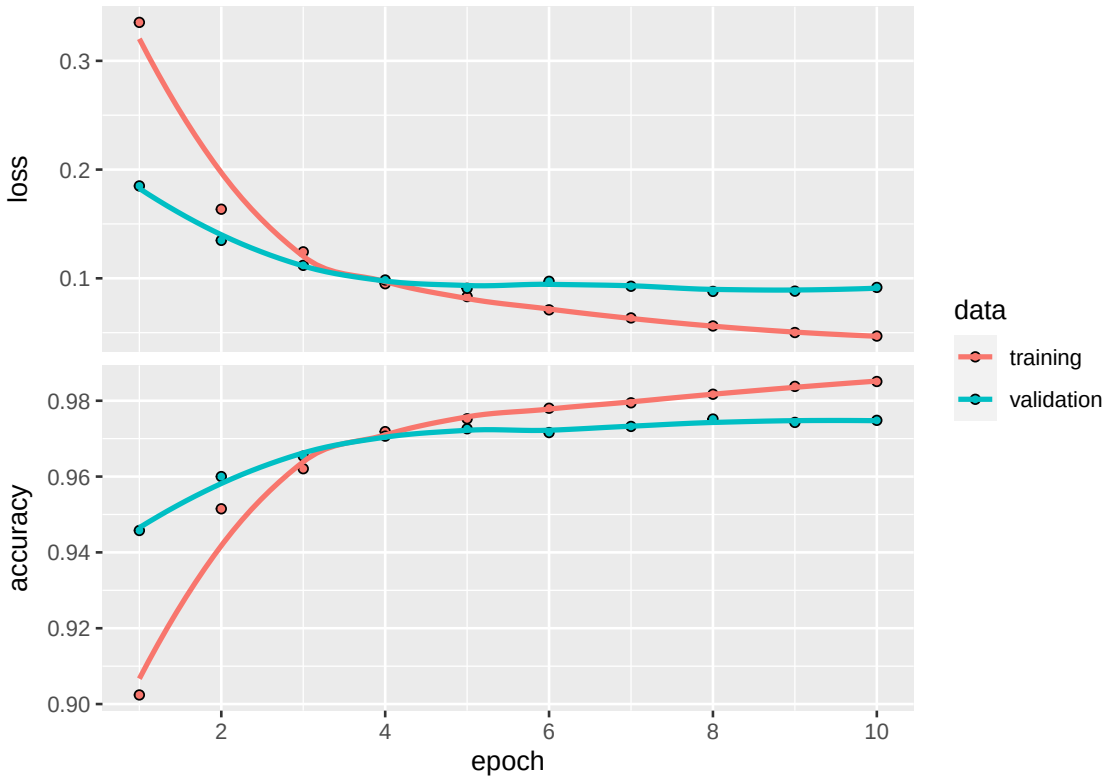
```
model %>%
  compile(
    loss = "sparse_categorical_crossentropy",
    optimizer = "adam",
    metrics = "accuracy"
  )
```

3.3.3. Entrenamiento y evaluación

Mediante la función `fit()` entrenamos el modelo introduciendo 10 periodos y estableciendo en 0,3 la fracción de datos de entrenamiento que serán utilizados como datos de validación.

El modelo separará esta fracción de los datos de entrenamiento, no los entrenará y evaluará la pérdida y cualquier métrica del modelo en estos datos al final de cada periodo.

```
history <- model %>%
  fit(
    x = mnist$train$x, y = mnist$train$y,
    epochs = 10,
    validation_split = 0.3,
    verbose = 2
  )
plot(history,type="l")
```



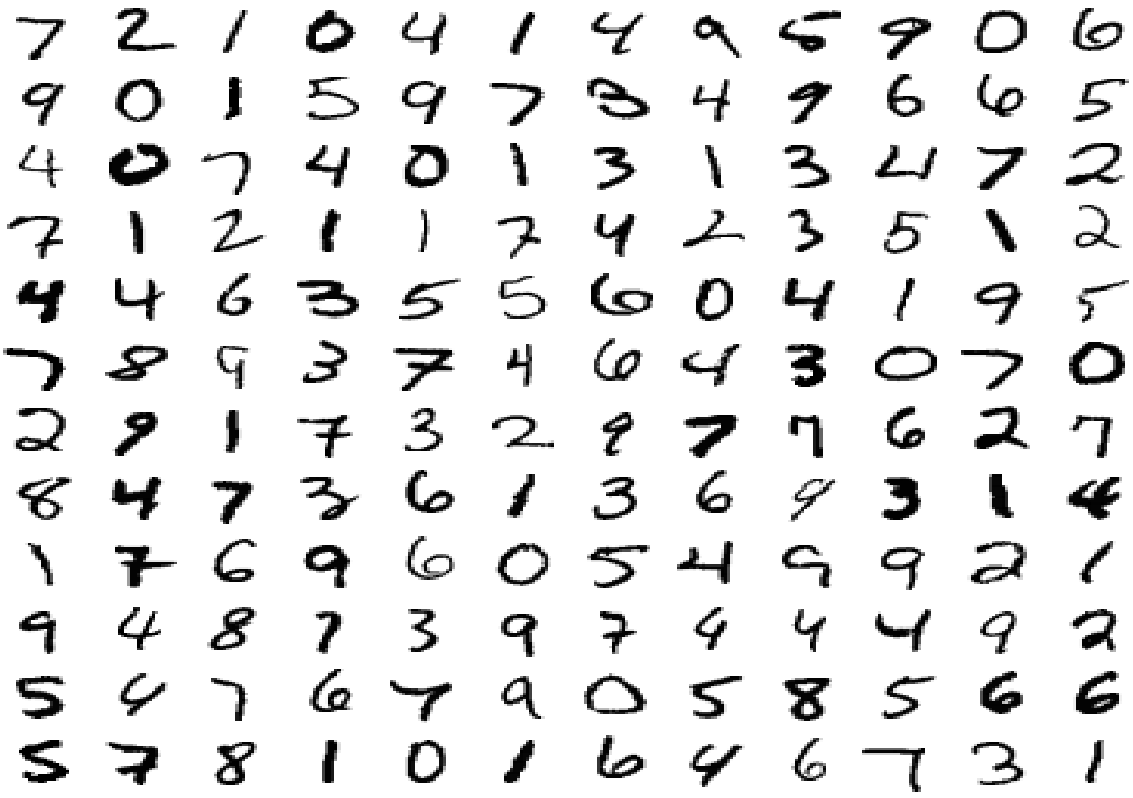
Observamos como es natural que, a medida que insertamos más periodos, mejora la precisión de entrenamiento hasta más de un 98 % y la de validación hasta más de un 97 %.

3.3.4. Predicciones en el conjunto test

Proyectamos las imágenes del conjunto test sobre las que queremos realizar las predicciones.

```r
# para hacer un grid 12 x 12 imágenes
# 144 en total
imagenes <- 1:144

# Generando el grid de imágenes
imagenes %>%
  # para iterar sobre las primeras n imagenes
  map(
    ~mnist$test$x[.x, , ] %>%
      #transformar a dataframe
      as.data.frame() %>%
      # agregar el ide de la columna
      # para identificar la posición de cada pixel
      rowid_to_column(var = "y") %>%
      # reshape del dataframe
      gather("x", "value", -y) %>%
      # volviendo la variable x numerica
      mutate(x = parse_number(x)) %>%
      # gráfico de tiles
      ggplot(aes(x = x, y = y, fill = value)) +
      geom_tile(show.legend = FALSE) +
      # reorganizando el eje y
      scale_y_reverse() +
      scale_fill_gradient(low = "white", high = "black") +
      theme_void()
  ) %>%
  cowplot::plot_grid(plotlist = .)
```



La función `predict` nos otorga las probabilidades de que cada imagen pertenezca a un dígito concreto del 0 al 9, por tanto, eligiendo aquel con mayor probabilidad, averiguaremos qué digito está prediciendo el modelo.

Mediante un bucle, podemos verificar si la predicción se corresponde con el dígito real. Si mostramos las primeras 12 predicciones, observamos que coincide con la primera fila de la imagen.

```
prediccion = model %>% predict(mnist$test$x)
for (i in c(1:12)) {
  write.table(rbind.data.frame(which.max(prediccion[i,])-1),
              col.names=FALSE, row.names=FALSE, quote=FALSE)
}
```

```
## 7
## 2
## 1
## 0
## 4
## 1
## 4
## 9
## 6
## 9
## 0
## 6
```

Cabe mencionar que a la hora de hacer el bucle hay que restar “1” al dígito de la predicción ya que la imagen del 0 está asociada a la etiqueta 1.

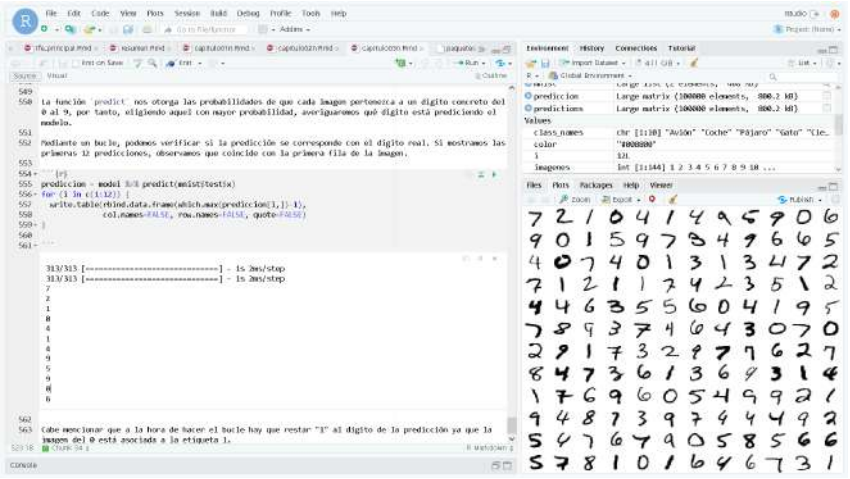


Figura 3.7: Se muestra las predicciones realizadas y los dígitos reales. Fuente: Elaboración propia

3.4. Conclusiones y líneas futuras de trabajo

A pesar de contar con el desarrollo de un modelo simple de red neuronal, hemos conseguido nuestro objetivo de reconocer y clasificar imágenes en una pequeña escala de profundidad.

El procedimiento, como vimos, es similar en los tres ejemplos, una primera fase de preparación de los datos, una segunda fase de construcción, entrenamiento y evaluación del modelo y una tercera y última fase de predicción, donde es más apreciable la efectividad de la red neuronal.

En conclusión podemos deducir que la precisión de nuestros modelos creados ha sido buena, se ha conseguido el objetivo buscado.

Cabe mencionar que los conjuntos de datos utilizados han sido obtenidos de la propia base de datos con la que cuenta R y que vienen preparados para una fácil manipulación. Esto nos ha permitido un fácil manejo de las imágenes a la hora de procesarlas e introducirlas en la red para ser entrenadas. Se podría decir que el “trabajo complicado” ya viene hecho por parte de R, contamos con datos depurados y divididos en dos conjuntos, uno para ser entrenado y otro para ser validado.

Como consecuencia de lo expuesto, la red creada está entrenada sobre una serie de imágenes concretas por lo que la introducción de imágenes nuevas, aunque estén relacionadas con la temática expuesta, probablemente no cumpla con los objetivos planteados. La investigación sobre redes neuronales convolucionales hace especial hincapié en este hecho y es que los avances en este campo se centran en la capacidad de aprender de una máquina por sí misma, en lugar de tener que memorizar un conjunto de imágenes ya que restringe su capacidad ante ligeras modificaciones.

Bien es cierto que este trabajo sirve como primer paso e introducción en este universo del que poco a poco se va conociendo más y cuyas expectativas futuras son máximas.

Como opinión personal considero de especial interés el desarrollo de una red neuronal convolucional más avanzada y desde el principio, donde las imágenes puedan ser fotos realizadas con una cámara, por ejemplo e introducidas por uno mismo, sin requerir para ello de fuentes de datos externas. Los ejemplos son muchos acerca de las necesidades de cualquier trabajador que le lleven a realizar este trabajo, desde saber qué tipo de ladrillo usar en contrucción para un arquitecto o conocer en detalle y al instante el tipo de planta o bacteria con la que está trabajando un biólogo.

Pero, sin duda alguna, lo que más he aprendido mediante el estudio de las redes neuronales y, en concreto de las redes neuronales convolucionales es todo aquello relacionado con avances en medicina y laboratorios. Me despierta gran interés profundizar más en la aplicación de este campo a la ayuda a personas realmente necesitadas. Hacen una gran labor por la humanidad al poder detectar y prever infartos a través de ecografías del corazón, realizar segmentaciones de imágenes del cerebro obtenidas mediante resonancias magnéticas, analizar angiografías y mamografías o poder detectar insuficiencias en el sistema inmune, entre otras muchas más adaptaciones. [10]

Apéndice A

Apéndice: Comentarios adicionales

A.1. Problemas de instalación

Al iniciar el trabajo encontré ciertas complicaciones al instalar en mi software de RStudio los paquetes de Keras y Tensorflow necesarios mediante la orden `install_tensorflow()`. La consola me remitía el mensaje de que dicha función no existía.

Investigando una posible solución encontré "Anaconda", una distribución libre y abierta para muchos lenguajes, entre ellos R y que busca facilitar y simplificar la gestión e implementación de paquetes. Desde la página oficial de Tensorflow ofrecen distintas alternativas para la instalación en R mediante la utilización de "Anaconda" y la librería `reticulate`, como muestran las siguientes líneas de código:

```
library(reticulate)
py_install("pandas")

conda_create("r-reticulate")

# Instalar "reticulate" y "scipy"
conda_install("r-reticulate", "scipy")

# Importar "scipy"
scipy <- import("scipy")

use_condaenv("r-reticulate")

# Crear un nuevo entorno
virtualenv_create("r-reticulate")

# Instalar "reticulate" y "scipy" desde el entorno virtual creado
virtualenv_install("r-reticulate", "scipy")

reticulate::py_install
```

```
install_tensorflow(version = "2.0.0")
```

Figura A.1: Métodos de instalación de TensorFlow en R. Fuente: Elaboración propia

Una vez probadas todas las alternativas que nos proporcionaba TensorFlow y mantener R actualizado, los mensajes de error persistían.

A.2. Estrategia determinada

Debido a la problemática comentada, mi recomendación personal a la hora de indagar en la investigación de este trabajo es utilizar RStudio Server, la versión web de RStudio. Esta versión es algo más flexible que la independiente. Las funciones internas son prácticamente iguales a las que ofrece RStudio Desktop pero nos permite usar RStudio desde cualquier máquina cliente a través de un navegador y conexión a internet.

Una de las ventajas que también posee esta estrategia es la posibilidad de un uso multiusuario pudiendo involucrar a varias personas en la implementación y depuración de proyectos de R, aunque no se permite la simultaneidad de inicio de sesión de varias cuentas.

El servidor de RStudio nos proporciona una interfaz basada en navegador para una versión de R que se ejecuta en un servidor Linux. Nos mantiene los cálculos y materiales seguros mediante un ususario y una contraseña pudiendo tener también más memoria y procesamiento que en una máquina de escritorio.

En mi equipo de trabajo cuento con la edición Windows Home, la cual puede que contribuyese a los problemas de instalación de paquetes. Esto no es un inconveniente a la hora de usar RStusio Server ya que el trabajo se realiza en un servidor remoto sin tener en cuenta el sistema operativo instalado en la máquina de trabajo.

Bibliografía

[1] Abadi, Martín; Agarwal, Ashish; Barham, Paul; Brevdo, Eugene; Chen, Zhifeng; Citro, Craig; Corrado, Greg S.; Davis, Andy; Dean, Jeffrey; Devin, Matthieu; Ghemawat, Sanjay; Goodfellow, Ian; Harp, Andrew; Irving, Geoffrey; Isard, Michael; Jia, Yangqing; Jozefowicz, Rafal; Kaiser, Lukasz; Kudlur, Manjunath; Levenberg, Josh; Mané, Dandelion; Monga, Rajat; Moore, Sherry; Murray, Derek; Olah, Chris; Schuster, Mike; Shlens, Jonathon; Steiner, Benoit; Sutskever, Ilya; Talwar, Kunal; Tucker, Paul; Vanhoucke, Vincent; Vasudevan, Vijay; Viégas, Fernanda; Vinyals, Oriol; Warden, Pete; Wattenberg, Martin; Wicke, Martin; Yu, Yuan y Zheng, Xiaoqiang (2015). «TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems».
https://www.tensorflow.org/. Software available from tensorflow.org.

[2] Algorithmia (Página web). «Lo que debes saber sobre el aumento de las inversiones en aprendizaje automático en las empresas».
https://www.cloudmasters.es/lo-que-debes-saber-sobre-el-aumento-de-las-inversiones\-en-aprendizaje-automatico-en-las-empresas/.

[3] Allaire, JJ y Chollet, François (2022). *keras: R Interface to 'Keras'*.
https://keras.rstudio.com. R package version 2.9.0.

[4] Allaire, JJ y Tang, Yuan (2022). *tensorflow: R Interface to 'TensorFlow'*.
https://github.com/rstudio/tensorflow. R package version 2.9.0.

[5] Allaire, JJ; Xie, Yihui; McPherson, Jonathan; Luraschi, Javier; Ushey, Kevin; Atkins, Aron; Wickham, Hadley; Cheng, Joe; Chang, Winston y Iannone, Richard (2022). *rmarkdown: Dynamic Documents for R*.
Https://github.com/rstudio/rmarkdown.

[6] Authors, The TensorFlow y RStudio (Blog). «Basic Image Classification».
https://tensorflow.rstudio.com/tutorials/beginners/basic-ml/tutorial_basic_classification/.

[7] Caparrini, Fernando Sancho (Página web). «Entrenamiento de Redes Neuronales: mejorando el Gradiente Descendente».
http://www.cs.us.es/~fsancho/?e=165.

[8] Chaos, Interactive (Página web). «El Perceptrón de Frank Rosenblatt».
https://interactivechaos.com/es/manual/tutorial-de-deep-learning/el-perceptron-de-frank-rosenblatt.

[9] Craven, Mark W. y Shavlik, Jude W. (1997). «Using neural networks for data mining». *Future Generation Computer Systems*, **13(2)**, pp. 211–229. ISSN 0167-739X. doi: https://doi.org/10.1016/S0167-739X(97)00022-8. https://www.sciencedirect.com/science/article/pii/S0167739X97000228. Data Mining.

[10] Delgado, Alberto (1999). «Aplicación de las Redes Neuronales en Medicina». *Revista de la Facultad de Medicina*, **47(4)**, p. 221–223. https://revistas.unal.edu.co/index.php/revfacmed/article/view/19460.

[11] for MailOnline, Victoria Woollaston (Página web). «Google releases TensorFlow: Search giant makes its artificial intelligence software available to the public». https://www.dailymail.co.uk/sciencetech/article-3311650/Google-releases-TensorFlow-Search-giant-makes-artificial-intelligence-software\-available-public.html.

[12] Gómez Ballester, Eva (2012). «Aportaciones a la mejora de la eficiencia de la búsqueda del vecino más cercano».

[13] Guide, IONOS Digital (Página web). «Keras: biblioteca de código abierto para crear redes neuronales». https://www.ionos.es/digitalguide/online-marketing/marketing-para-motores-de\-busqueda/que-es-keras/.

[14] Henry, Lionel y Wickham, Hadley (2020). *purrr: Functional Programming Tools*. Http://purrr.tidyverse.org, https://github.com/tidyverse/purrr.

[15] Hisour (Página web). «Aprendizaje automático». https://www.hisour.com/es/machine-learning-42773/.

[16] InteractiveChaos (Página web). «Backpropagation». https://interactivechaos.com/es/manual/tutorial-de-machine-learning/backpropagation.

[17] Kawaf, Tareef (Página web). «TensorFlow for R». https://www.rstudio.com/blog/tensorflow-for-r/.

[18] Krizhevsky, Alex (Página web). «CIFAR10 Dataset». https://www.tensorflow.org/datasets/catalog/cifar10.

[19] Kumar, Sandesh (Página web). «Redes neuronales de convolución en pocas palabras | Guía de la A a la Z». https://www.herevego.com/redes-neuronales-de-convolucion/.

[20] LeCun, Yann; Kavukcuoglu, Koray y Farabet, Clement (2010). «Convolutional networks and applications in vision», pp. 253–256. doi: 10.1109/ISCAS.2010.5537907.

[21] lisdatasolutions (Página web). «Deep Learning: clasificando imágenes con redes neuronales». https://www.lisdatasolutions.com/blog/deep-learning-clasificando-imagenes-con\-redes-neuronales/.

[22] Lu, Hongjun; Setiono, R. y Liu, Huan (1996). «Effective data mining using neural networks». *IEEE Transactions on Knowledge and Data Engineering*, **8(6)**, pp. 957–961. doi: 10.1109/69.553163.

[23] Luque Calvo, Pedro L. (2017). *Escribir un Trabajo Fin de Estudios con R Markdown*.
http://destio.us.es/calvo.

[24] Luque Calvo, Pedro L. (2021). «Página personal de Pedro L. Luque».
http://destio.us.es/calvo.

[25] Mycielski, Jan (1972). «Marvin minsky and seymour papert, perceptrons, an introduction to computational geometry». *Bulletin of the American Mathematical Society*, **78(1)**, pp. 12–15.

[26] Müller, Kirill y Wickham, Hadley (2022). *tibble: Simple Data Frames*. Https://tibble.tidyverse.org/, https://github.com/tidyverse/tibble.

[27] O'Shea, Keiron y Nash, Ryan (2015). «An introduction to convolutional neural networks». *arXiv preprint arXiv:1511.08458*.

[28] Rosa, Johan (Página web). «Tensorflow y keras con R».
https://www.johan-rosa.com/post/tensorflow-y-keras-con-r/.

[29] Rumelhart, David E; Hinton, Geoffrey E y Williams, Ronald J (1986). «Learning representations by back-propagating errors». *nature*, **323(6088)**, pp. 533–536.

[30] Sokoliuk, Anton; Kondratenko, Galyna; Sidenko, Ievgen; Kondratenko, Yuriy; Khomchenko, Anatoly y Atamanyuk, Igor (2020). «Machine Learning Algorithms for Binary Classification of Liver Disease», pp. 417–421. doi: 10.1109/PICST51311.2020.9468051.

[31] TensorFlow (Página web). «Plataforma de extremo a extremo de código abierto para el aprendizaje automático».
https://www.tensorflow.org/?hl=es-419.

[32] TensorFlow y RStudio (Blog). «Convolutional Neural Network, CNN».
https://tensorflow.rstudio.com/tutorials/advanced/images/cnn/.

[33] Warwick, Kevin y Shah, Huma (2016). «Can machines think? A report on Turing test experiments at the Royal Society». *Journal of experimental & Theoretical artificial Intelligence*, **28(6)**, pp. 989–1007.

[34] Wickham, Hadley (2016). *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York. ISBN 978-3-319-24277-4.
https://ggplot2.tidyverse.org.

[35] Wickham, Hadley; François, Romain; Henry, Lionel y Müller, Kirill (2022). *dplyr: A Grammar of Data Manipulation*. Https://dplyr.tidyverse.org, https://github.com/tidyverse/dplyr.

[36] Wickham, Hadley y Girlich, Maximilian (2022). *tidyr: Tidy Messy Data*. Https://tidyr.tidyverse.org, https://github.com/tidyverse/tidyr.

[37] Wickham, Hadley; Hester, Jim y Bryan, Jennifer (2022). *readr: Read Rectangular Text Data*. Https://readr.tidyverse.org, https://github.com/tidyverse/readr.

[38] Widrow, B. y Lehr, M.A. (1990). «30 years of adaptive neural networks: perceptron, Madaline, and backpropagation». *Proceedings of the IEEE*, **78(9)**, pp. 1415–1442. doi: 10.1109/5.58323.

[39] Wikipedia, la enciclopedia libre (Página web). «Minería de datos». https://es.wikipedia.org/wiki/Mineria_de_datos.

[40] Wilke, Claus O. (2020). *cowplot: Streamlined Plot Theme and Plot Annotations for 'ggplot2'*.
https://wilkelab.org/cowplot/. R package version 1.1.1.

[41] Xie, Yihui (2022). *knitr: A General-Purpose Package for Dynamic Report Generation in R*.
https://yihui.org/knitr/. R package version 1.39.

[42] Zaforas, Manuel (Blog). «TensorFlow, o cómo será el futuro de la Inteligencia Artificial según Google».
https://www.paradigmadigital.com/dev/tensorflow-sera-futuro-la-inteligencia\
-artificial-segun-google/.

[43] Álvaro Gonzalo (Página web). «¿Qué es el sobreajuste u overfitting y por qué debemos evitarlo?».
https://machinelearningparatodos.com/que-es-el-sobreajuste-u-overfitting-y-por-que\
-debemos-evitarlo/.