Doble Grado en
Matemáticas y Estadística

TRABAJO FIN DE GRADO

REDES CONVOLUCIONALES

Carmelo Bonilla Carrión

Sevilla, Junio de 2020

Tutor: Rafael Pino Mejías

Índice general

Resumen III
Abstract IV
Índice de Figuras V
Índice de Cuadros VII

1. Introducción **1**

2. Redes de Neuronas **3**
2.1. Redes de Neuronas Artificiales 4
2.2. Entrenamiento de una red neuronal 10
2.2.1. Forward Propagation 11
2.2.2. Backward Propagation 12
2.2.3. Actualización de los pesos 12

3. Redes de Neuronas Convolucionales (CNN) **15**
3.1. Introducción 15
3.2. Visión artificial 15
3.2.1. Convolución 16
3.2.1.1. Convolución entre volúmenes 18
3.2.1.2. Aprendizaje de filtros de convolución 19
3.3. Estructura y propiedades de las CNN 20
3.4. Componentes en la arquitectura de una CNN 21
3.4.1. Capa convolucional (*Convolutional layer*) 21
3.4.2. Capa de agrupación (*Pooling layer*) 23
3.4.3. Capa de unidades lineales rectificadas ReLu (*Rectified Linear Units Layer*) 25
3.4.4. Capa completamente conectada (*Full connected layer*) 26
3.4.5. Capa de pérdida (*Loss layer*) 26
3.5. Hiperparámetros 27
3.6. Arquitecturas de CNN 28
3.7. Convoluciones 1×1 32
3.8. Regularización 33

4. Aplicaciones de CNNs en R **35**
4.1. Librería Keras 35
4.1.1. Funciones 35
4.1.1.1. layer_conv_1d() 36
4.1.1.2. layer_conv_2d() 36
4.1.1.3. layer_embedding() 36

4.1.1.4. layer_activation() 37
4.1.1.5. layer_dropout() 37
4.1.1.6. layer_golbal_max_pooling_1d() 37
4.1.1.7. layer_max_pooling_2d() 37
4.1.1.8. layer_dense() 37
4.1.1.9. layer_flatten() 38
4.2. Ejemplo práctico I 38
4.3. Ejemplo práctico II 41

Bibliografía **47**

Resumen

El Aprendizaje Profundo (Deep Learning) está teniendo un gran desarrollo en la actualidad gracias a la evolución de los ordenadores y al uso de GPUs. Debido a este desarrollo, las redes de neuronas están apareciendo con más fuerza que nunca desde que surgieron por primera vez en los años 50. Dentro de estas redes de neuronas se encuentran las redes convolucionales, que serán el tema central de este Trabajo de Fin de Grado, las cuales se encargan de resolver problemas de clasificación, localización e identificación de objetos dentro de imágenes.

Este trabajo comienza con un estudio general de las redes de neuronas artificiales así como de sus distintas arquitecturas. Seguidamente se estudiará con un mayor detalle las redes de neuronas convolucionales. En este aspecto, el trabajo se centra en conocer las características de este tipo de redes, su funcionamiento, sus distintas arquitecturas y cómo realizan la operación de convolución, que es lo que realmente caracteriza a estas redes. En la parte final del trabajo, se llevarán a cabo aplicaciones en lenguaje R para la clasificación de imágenes a partir de las redes de neuronas convolucionales, utilizando la librería Keras.

Abstract

Currently, Deep Learning is witnessing a great development thanks to the evolution of computers and the use of GPUs. As a consequence, neural networks are gathering more stream than ever, since their first appearance during the 1950s. Among these, convolutional networks can be found, which will constitute the focus of this dissertation. Convolutional networks are in charge of solving classification, localisation and identification issues of objects within images.

This dissertation starts with a general review of artificial neural networks and their architectures, followed by a more detailed study of convolutional neural networks. To this respect, this dissertation will focus on getting to know the characteristics of these types of networks, their operation, their different architectures, as well as how they perform the convolution operation, which is what characterises these networks. At the end of the document, theory will be put into practice in the R language for image classification, based on convolutional neural networks, by using Keras library.

Índice de figuras

1.1. Machine Learning y Deep Learning como subconjuntos de la IA. 1

2.1. Estructura de una neurona biológica. 3
2.2. Red neuronal artificial. 5
2.3. Elementos de una neurona artificial. 5
2.4. Funciones de activación. 6
2.5. Perceptrón. 7
2.6. Perceptrón multicapa. 7
2.7. Red neuronal recurrente. 8
2.8. Red neuronal convolucional. 8
2.9. Máquinas de Boltzmann. 9
2.10. Redes de creencias profundas. 9
2.11. Red neuronal con 3 capas 11

3.1. Extracción del borde de una imagen mediante convolución. 18
3.2. Visualización de una CNN. 20
3.3. Arquitectura de una CNN. 21
3.4. Imagen ejemplo de un 1. 22
3.5. Ejemplo de un filtrado. 22
3.6. Max-pooling: filtro 2×2 y S=2. 24
3.7. Max-pooling. 24
3.8. Max-pooling con 3 mapas de características. 25
3.9. Arquitectura de LeNet. 29
3.10. Arquitectura LeNet5. 29
3.11. Arquitectura GoogLeNet 30
3.12. Arquitectura AlexNet 31
3.13. Arquitectura VGGNet 31

Índice de cuadros

3.1. Hiperparámetros de la convolución entre volúmenes. 27

Capítulo 1

Introducción

La **Inteligencia Artificial o IA** nació en la década de 1950, y se define como el proceso de automatizar las tareas intelectuales que son realizadas por los seres humanos. Es un campo que abarca el **aprendizaje automático (*Machine Learning*)** y el **aprendizaje profundo (*Deep Learning*)**.

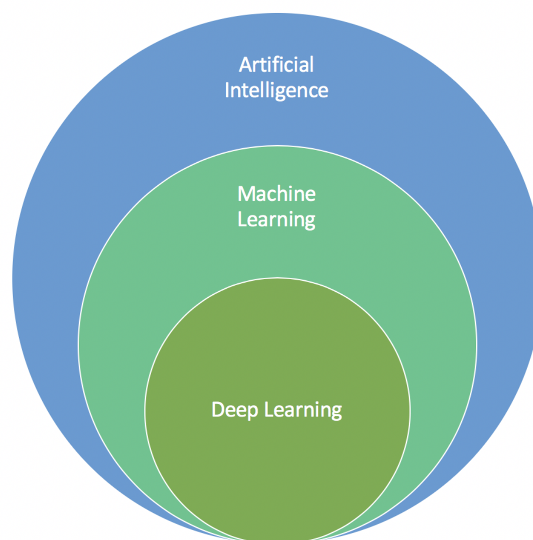


Figura 1.1: Machine Learning y Deep Learning como subconjuntos de la IA.

El aprendizaje automático es la ciencia que permite que los ordenadores funcionen con determinadas acciones sin haber sido programados específicamente para ello y representa una de las áreas fundamentales de la inteligencia artificial. Los algoritmos deben perfeccionarse para ser lo más precisos posibles a la hora de realizar una tarea y, para ello, la máquina es entrenada con una gran cantidad de datos dando la oportunidad a estos algoritmos de ser perfeccionados.

El aprendizaje automático descubre reglas para ejecutar una tarea de procesamiento de datos, dados una serie de ejemplos. Por tanto, para el aprendizaje automático es necesario:

- Datos de entrada.
- Ejemplos que la salida espera. Por ejemplo, si se introduce una imagen de un automóvil, lo que se espera es que el ordenador clasifique esa imagen como un

automóvil.

- Una forma de ver si el algoritmo está haciendo bien su trabajo. Es decir, ver en qué medida se está equivocando o acertando el algoritmo para así poder mejorarlo. Esto es lo que se llama aprendizaje.

El aprendizaje profundo es una rama del aprendizaje automático y se basa en un subconjunto de algoritmos del aprendizaje automático que intentan modelar niveles altos de abstracción en los datos, generalmente estos algoritmos tienen múltiples etapas de procesamiento, con una estructura normalmente compleja y cuyas etapas se componen de una serie de transformaciones no lineales. El tema de este trabajo, las convoluciones, se usan con gran frecuencia en el aprendizaje profundo para aplicaciones de la visión artificial.

Las redes de neuronas profundas, las cuales son un instrumento muy importante para implementar el aprendizaje profundo, están teniendo mucho éxito en los últimos años, y en particular, las redes de neuronas convolucionales son las que tratan de resolver problemas como la clasificación de imágenes a través de un ordenador.

Capítulo 2

Redes de Neuronas

Las redes de neuronas son una herramienta muy potente utilizada en el campo de la inteligencia artificial.

Las redes de neuronas artificiales son modelos matemáticos que simulan las interconexiones y el funcionamiento de las neuronas biológicas. Las neuronas biológicas están formadas por:

- Dendritas: fibras que transportan información en forma de señal eléctrica, desde el exterior hasta el núcleo.
- Sinapsis: son los puntos de conexión entre las neuronas. Las neuronas reciben la información en la sinapsis a la cual están conectadas las dendritas.
- Núcleo: recibe las señales de las dendritas, las procesa y produce una respuesta que envía al axón.
- Axón: parte terminal de la neurona que se puede conectar a la sinapsis de otras neuronas.

El aprendizaje de las neuronas biológicas se basa en la modificación a lo largo del tiempo de la señal de salida generada por el núcleo, de acuerdo a ciertos tipos de señales de entrada. Las neuronas se especializan en reconocer estímulos durante su vida.

La estructura de la neurona se muestra en la siguiente figura:

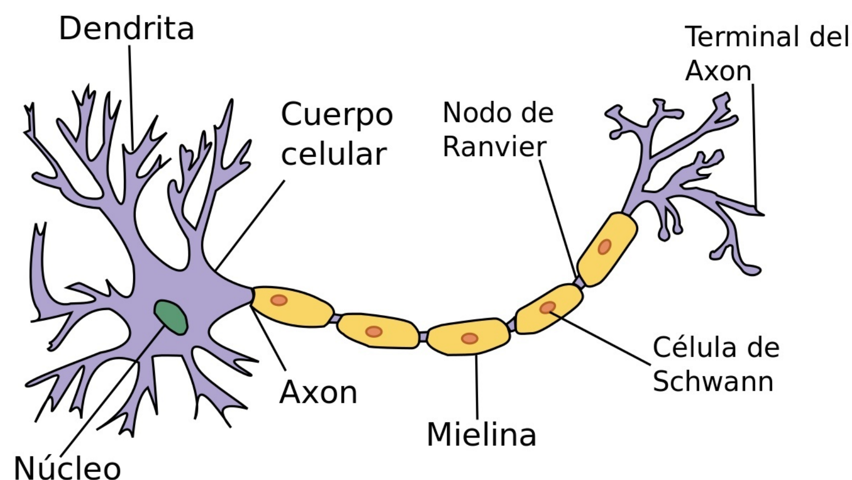


Figura 2.1: Estructura de una neurona biológica.

Las neuronas artificiales se basan en la estructura de la neurona biológica y utilizan funciones matemáticas de valor real para simular su comportamiento. Las analogías con la neurona biológica son:

- Dendritas D: número de entradas que la neurona acepta.
- Sinapsis $w_i, i = 0, 1, .., D-1$: pesos asociados a las dendritas. Estos son valores que van cambiando a lo largo del tiempo, especializando a la neurona. Se indica el valor de entrada del vector D-dimensional $x = (x_0, x_1, .., x_{D-1})$, la operación efectuada en la sinapsis es:

$$x_i w_i \ \ \forall i \in [0, D-1].$$

- Núcleo: función que une los valores provenientes de la sinapsis y define el comportamiento de la célula. Para simular el comportamiento de la neurona biológica, es decir, activarse solamente en presencia de ciertos estímulos, se utilizan funciones no lineales.
- Axón: el valor disponible como entrada para otras neuronas.

2.1. Redes de Neuronas Artificiales

El elemento fundamental de una red de neuronas artificial es una neurona o elemento de procesamiento. Es un elemento muy simple que se encarga de transformar señales que recibe de entrada en una única salida. Estas entradas, pueden proceder de otras neuronas o bien pueden ser entradas a la red de neuronas artificial desde el exterior. De igual forma que estas entradas pueden ser de otras neuronas, la salida, puede transmitirse a otras neuronas o ser la salida de la red.

Las redes de neuronas artificiales se usan para estimar o aproximar funciones que puedan depender de una gran cantidad de entradas, muchas de las cuales pueden no conocerse.

La señales de entrada se encuentran moduladas por un factor, llamado peso, que gradúa la importancia de la conexión existente entre la neurona receptora y el emisor de la señal (generalmente otra neurona). El valor del peso es ajustable en función de la experiencia en cuestión y esto hace que las redes de neuronas artificiales sean un instrumento adaptable a varios tipos de entrada y tengan la capacidad de aprender.

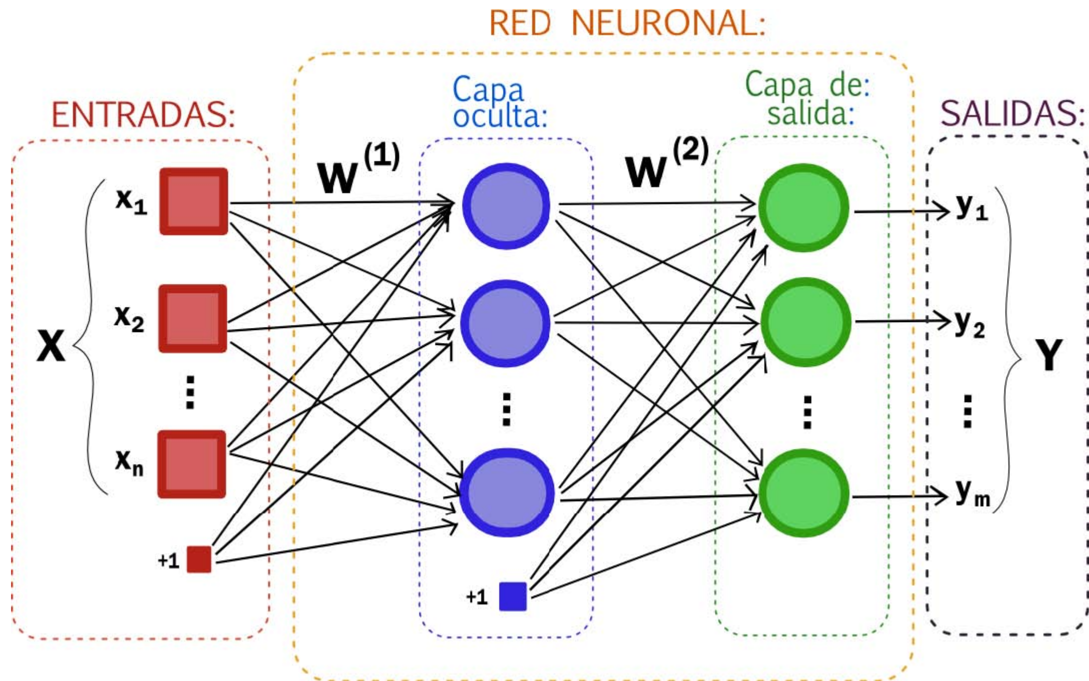


Figura 2.2: Red neuronal artificial.

Las Redes de Neuronas Artificiales (RNA) son un modelo matemático simplificado e inspirado en cómo el sistema nervioso procesa la información. Funciona utilizando a la misma vez un número elevado de unidades de procesamiento interconectadas que parecen versiones abstractas de neuronas.

Según [10], la distribución de neuronas dentro de la red se realiza formando capas, con un número determinado de dichas neuronas en cada una de ellas. A partir de su situación dentro de la red, se pueden distinguir tres tipos de capas:

- **De entrada**: es la capa que recibe directamente la información proveniente de las fuentes externas de la red.
- **Ocultas**: son internas a la red y no tienen contacto directo con el entorno exterior. El número de niveles ocultos puede estar entre cero y un número elevado. Las neuronas de las capas ocultas pueden estar interconectadas de distintas maneras, lo que determina, junto con su número, las distintas topologías de redes neuronales.
- **De salida**: transfieren información de la red hacia el exterior.

Se dirá que una red está totalmente conectada si todas las salidas desde un nivel llegan a todos y cada uno de los nodos del siguiente nivel.

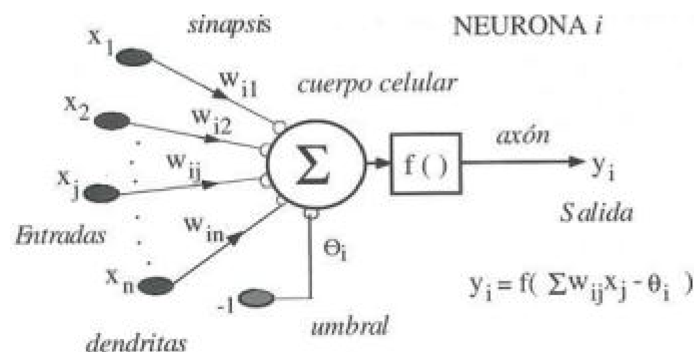


Figura 2.3: Elementos de una neurona artificial.

En la figura anterior, podemos apreciar los elementos de la neurona. Según [11], en ella, la suma de las n entradas x_j, ponderadas con los pesos sinápticos w_{ij}, genera la entrada ponderada total o "potencial postsináptico" de la neurona i. Posteriormente, se aplica una función de activación o transferencia a la diferencia entre el "potencial postsináptico" y el umbral θ_i, obteniéndose la salida de la neurona, y_i, donde $y_i(t) = f(\sum_{j=1}^{n} w_{ij}x_j - \theta_i)$. La función de activación $f()$ se suele considerar determinista y en la mayor parte de los modelos es monótona creciente y continua. La forma $y_i(t) = f(x)$ de las funciones de activación más empleadas en las redes de neuronas artificiales se muestran en la siguiente figura, donde x representa el potencial postsináptico e y el estado de activación.

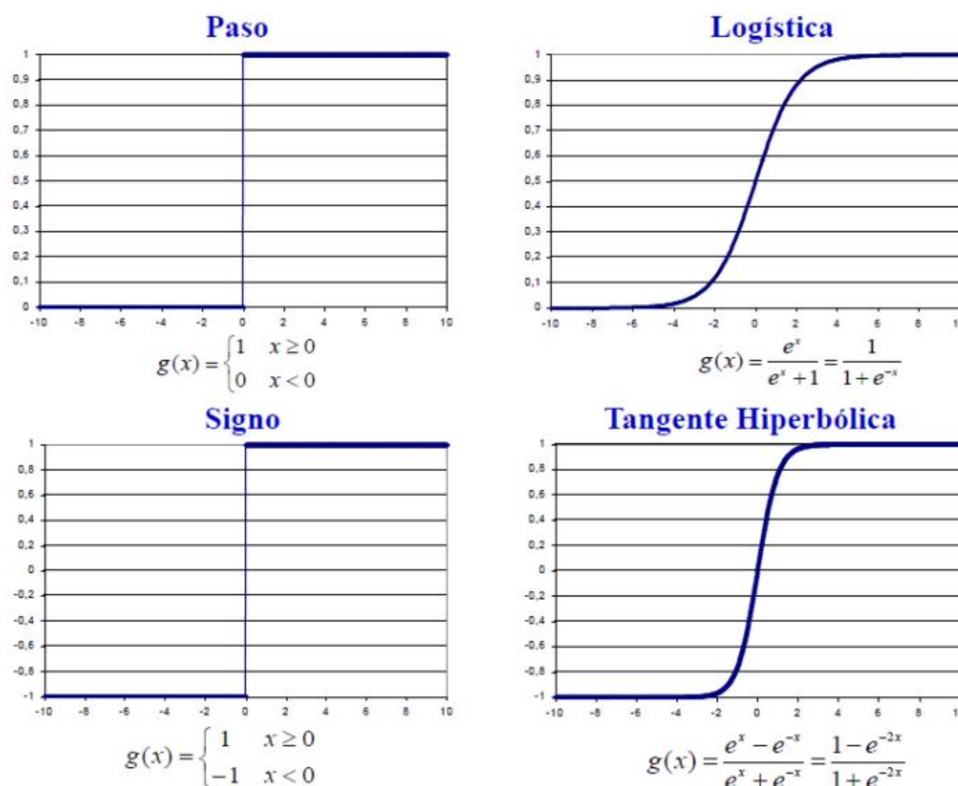


Figura 2.4: Funciones de activación.

En las redes de neuronas convolucionales, la función de activación que se usa es la *rectifier*, utilizadas en las capas ReLu, en la cual profundizaremos en el siguiente capítulo. Esta función se define simplemente como $f(x) = max(0, x)$, es decir, permiten el paso de todos los valores positivos sin cambiarlos, pero los valores negativos los asigna a cero.

Según [11], las neuronas artificiales pueden estar estructuradas en capas interconectando neuronas con diferentes pesos asociados, con la posibilidad de que haya interacción entre neuronas de una misma capa o no. Cualquier red podría ser representada por una matriz de pesos que contiene todos los pesos de dicha red. Si w_{ij} es positivo indica que la relación entre las neuronas es excitadora, es decir, siempre que la neurona i esté activada la neurona j recibirá una señal que tenderá a activarla. Si w_{ij} es negativo la sinapsis será inhibidora, en este caso, si la neurona i está activada enviará una señal que desactivará la neurona j. Finalmente, si w_{ij} es 0, se supone que no hay conexión entre ambas.

En función de cómo se realiza la interconexión entre neuronas, se pueden tener las siguientes arquitecturas [5]:

- **Perceptrón**, es la forma más simple de red neuronal y la base de los modelos más avanzados que se han desarrollado en el aprendizaje profundo. Normalmente se utilizan en problemas de clasificación donde se necesita dar datos referentes a etiquetas de observaciones (binarias o multinomiales) basadas en los datos de entrada. Estos valores de entrada se envían directamente a la capa de salida después de que se multipliquen por pesos y un sesgo es añadido a la suma acumulada, la cual se pone en una función de activación, que no es más que una función que define la salida. Cuando esa salida se encuentra por encima o por debajo de un cierto umbral, se determina la salida final.

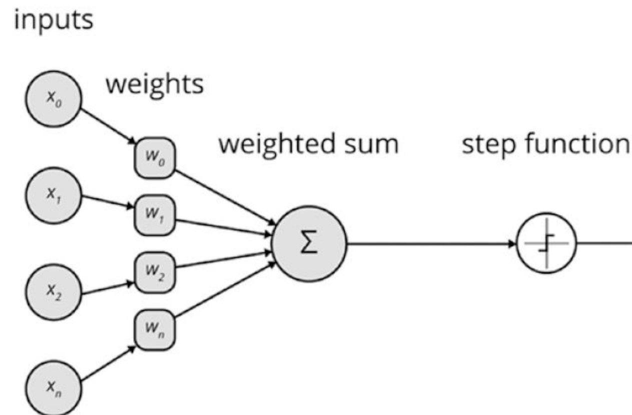


Figura 2.5: Perceptrón.

- **Perceptrón multicapa**, es muy similar al perceptrón, presenta múltiples capas que se encuentran interconectadas de forma que constituyen una red neuronal de retroalimentación. Cada neurona en una capa tiene conexiones dirigidas a neuronas de otra capa.

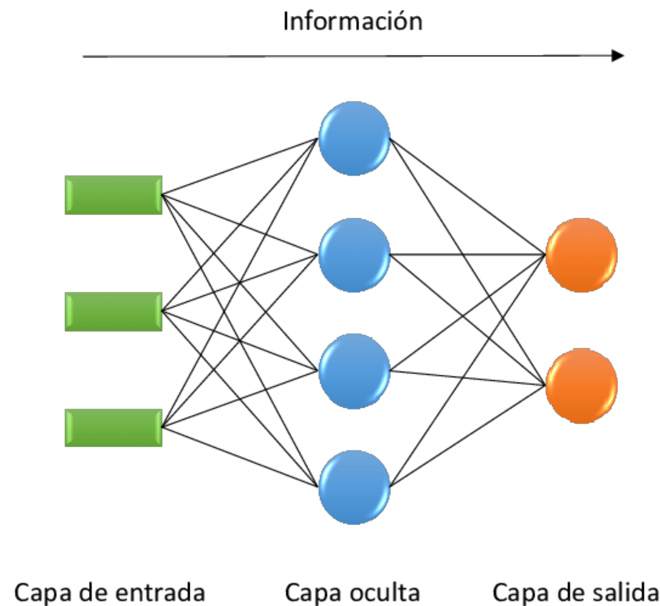


Figura 2.6: Perceptrón multicapa.

Uno de los factores más importantes en este modelo y en el perceptrón, es el algoritmo de propagación hacia atrás (*back-propagation*), un método común para el entrenamiento de redes neuronales. La propagación hacia atrás pasa el error calculado en la capa de

salida hacia la capa de entrada, para poder apreciar la contribución de cada capa al error y alterar así la red en consecuencia. Aquí, se usa un algoritmo de descenso del gradiente para determinar el grado que los pesos deberían de cambiar en cada iteración. Se explicará con mayor detalle posteriormente.

- **Redes recurrentes (RNN)**, son modelos de redes de neuronas artificiales donde las conexiones entre las unidades forman un ciclo dirigido. Un ciclo dirigido es una secuencia donde el camino a lo largo de los vértices y bordes está completamente determinado por el conjunto de aristas que se ha utilizado y, por tanto, tiene la apariencia de seguir un orden específico. Son generalmente utilizadas para el reconocimiento de voz y de escritura.

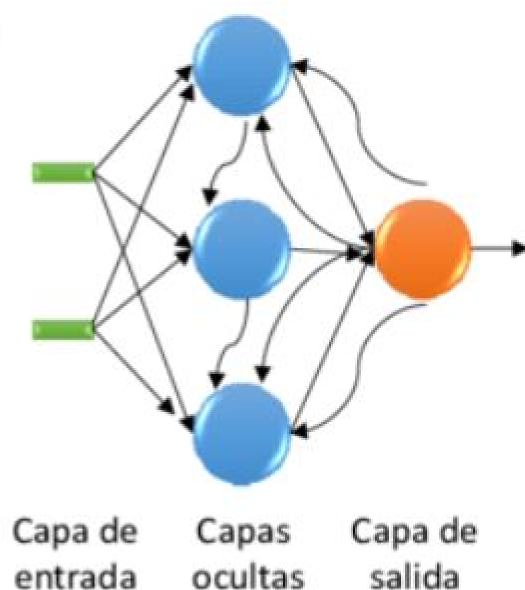


Figura 2.7: Red neuronal recurrente.

- **Redes convolucionales (CNN)**, son modelos que se utilizan en mayor frecuencia para el procesamiento de imágenes y visión para ordenadores. Están diseñadas de tal manera que imitan la estructura de la corteza visual animal. Las redes de neuronas convolucionales tienen neuronas dispuestas en tres dimensiones: anchura, altura y profundidad. Aquí, las neuronas de una capa solamente están conectadas a una pequeña región de la capa anterior.

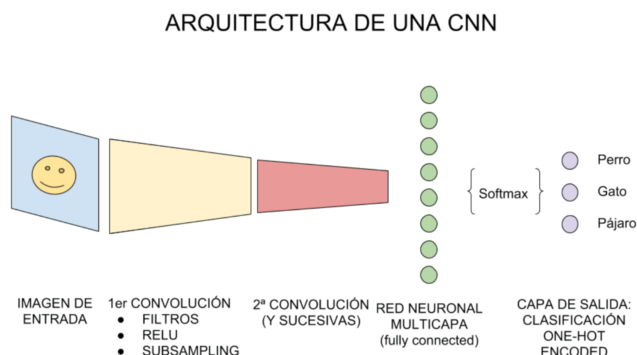


Figura 2.8: Red neuronal convolucional.

- **Máquinas de Boltzmann restringidas (RBM)**, son un tipo de modelo binario que tiene una arquitectura única, de modo que hay múltiples capas ocultas de variables aleatorias y una red de unidades binarias estocásticas acopladas simétricamente. Se componen de un conjunto de unidades visibles y de una serie de capas de unidades ocultas, aunque aquí no hay conexiones entre unidades de la misma capa. Las RBM pueden aprender representaciones internas, complejas y abstractas en tareas como el reconocimiento de objetos o de voz.

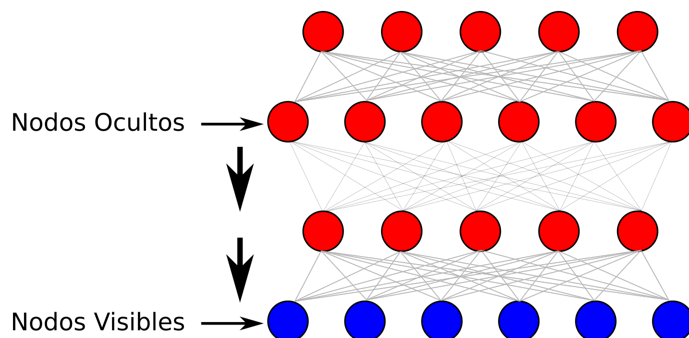


Figura 2.9: Máquinas de Boltzmann.

- **Redes de creencias profundas**, son similares a las de Boltzmann, excepto que la capa oculta de cada subred es la capa visible para la siguiente subred. Son en general un modelo gráfico generativo de múltiples capas de variables no observables con conexiones entre las capas pero no entre las unidades de cada capa.

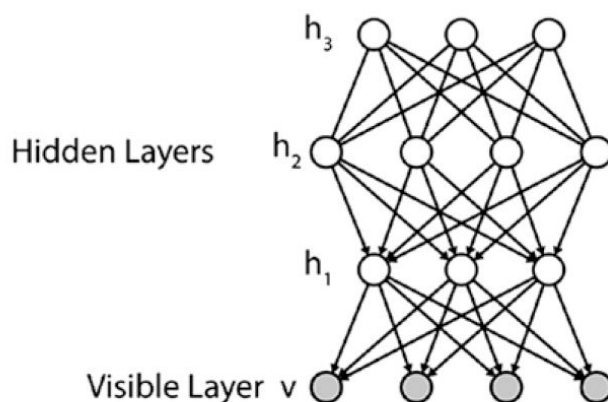


Figura 2.10: Redes de creencias profundas.

Un último elemento de las redes de neuronas artificiales es la regla de aprendizaje. El aprendizaje se define libremente como el significado de mejorar alguna tarea a través de la práctica. ¿Cómo sabe un ordenador si está mejorando y cómo mejorar? Hay diferentes respuestas posibles a estas preguntas, las cuales producen distintos tipos de aprendizaje automático.

Se puede decir al algoritmo la respuesta correcta para un determinado problema de forma que se ejecute de manera satisfactoria la próxima vez. Se espera que sólo haya que decir algunas respuestas correctas y luego pueda averiguar cómo obtenerlas para otros problemas. Alternativamente, se puede decir si la respuesta es correcta o errónea, pero no cómo encontrarla, para que tenga que buscar la respuesta idónea. Una variante es dar

una puntuación a la respuesta de acuerdo con lo satisfactoria que sea, en lugar de dar solamente una respuesta correcta o incorrecta. Finalmente, es posible que no se tengan las respuestas deseadas, queriendo únicamente que el algoritmo encuentre entradas que tengan algo en común.

Estas diferentes respuestas a las posibles preguntas proporcionan una forma útil de clasificar diferentes algoritmos:

- **Aprendizaje supervisado**: Se proporciona un conjunto de entrenamiento de ejemplos con los valores de la variable objetivo y en base a este conjunto de entrenamiento, el algoritmo se generaliza para responder de forma correcta a todas las entradas posibles. Esto también se llama aprender de los ejemplos.
- **Aprendizaje no supervisado**: No se conocen los valores de la variable objetivo, es el algoritmo el que intenta identificar similitudes entre las entradas para que las entradas que tienen algo en común se clasifiquen juntas. El enfoque estadístico para el aprendizaje no supervisado es conocido como estimación de densidad.
- **Aprendizaje reforzado**: Se encuentra entre el aprendizaje supervisado y el no supervisado. Al algoritmo se le informa si la respuesta es incorrecta pero no se le dice cómo corregirlo. Tiene que explorar y probar diferentes posibilidades hasta que descubra el modo de obtener la respuesta correcta. Al aprendizaje reforzado a veces se le llama aprender con un crítico, porque se le dicen las respuestas pero no la manera de mejorarlas.
- **Aprendizaje evolutivo**: La evolución biológica puede verse como un proceso de aprendizaje, organismos biológicos se adaptan para mejorar sus tasas de supervivencia y la posibilidad de tener descendencia en su ambiente. La idea es ver la forma de modelar esto en un ordenador, usando aptitudes, que corresponden a cuánto de buena es la solución actual.

2.2. Entrenamiento de una red neuronal

Uno de los métodos más conocidos y efectivos para entrenar redes de neuronas artificiales es el algoritmo de retropropagación para el error, el cual modifica sistemáticamante los pesos de las conexiones entre las neuronas de modo que la respuesta se vaya acercando cada vez más a la respuesta deseada. El entrenamiento de una red de neuronas artificiales se lleva a cabo en dos etapas diferentes: propagación hacia delante (*forward propagation*) y la propagación hacia atrás (*backward propagation*).

En la primera etapa se calculan todas las activaciones de las neuronas de la red, desde la primera hasta la última capa. Durante esta etapa, los valores de los pesos sinápticos son todos fijos, en la primera iteración se toman los valores por defecto. En la segunda etapa, la respuesta de la red se compara con la respuesta deseada, obteniendo así el error de la red. El error que se ha calculado, se propaga en dirección opuesta a la de las conexiones sinápticas. Al final de esta segunda etapa, los pesos se modifican según los errores calculados para minimizar la diferencia entre la salida actual y la deseada.

Se estudiarán a continuación ambas etapas con un mayor detalle.

2.2.1. Forward Propagation

Vamos a ver un ejemplo sobre la propagación hacia delante considerando la siguiente figura:

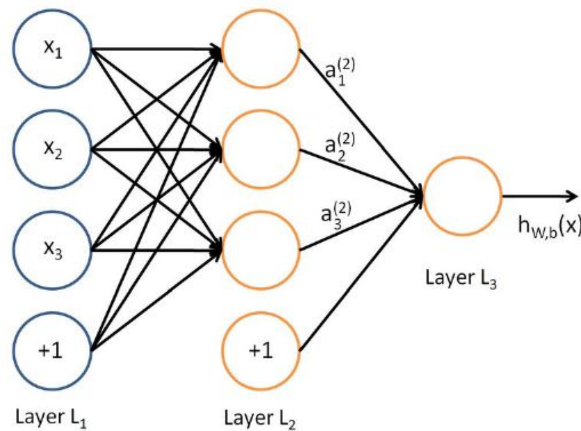


Figura 2.11: Red neuronal con 3 capas

Sea n_l el número de capas de la red, que en este caso se tiene que $n_l = 3$. Denotando L_l a cada capa donde $l = 1, .., n_l$, de esta forma se tiene que L_1 es la capa de entrada y L_{n_l} es la capa de salida. La red considerada tiene i parámetros $(W, b) = (W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)})$ donde:

- W^l_{ij} es el peso asociado a la conexión entre la neurona j de la capa l y la neurona j de la capa $l+1$ para $1 \leq i \leq 3,\ \ 1 \leq j \leq 3,\ \ 1 \leq l \leq 3$.
- $W^{(l)}$ representa la matriz de pesos entre las neuronas de la capa l y la capa $l+1$.
- a^l_i constituye la salida, es decir, la activación de la neurona en la capa l, en nuestro caso $a^l_i = x_i$ para $1 \leq i \leq 3,\ \ 1 \leq l \leq 3$.
- b^l_i es el sesgo asociado a la neurona i en la capa $l+1$, para $1 \leq i \leq 3,\ \ 1 \leq l \leq 3$.
- s_l corresponde al número de neuronas en la capa l sin contar las neuronas de sesgo, para $1 \leq i \leq 3$.

En el ejemplo se tiene que $W^{(1)} \in \mathbb{R}^{3\times3}$ y $W^{(2)} \in \mathbb{R}^{1\times3}$. La salida de la red ahora se puede expresar como se muestra en las siguientes ecuaciones donde la función $f()$ indica la función de activación de las neuronas.

$$\begin{aligned}
a^{(2)}_1 &= f(W^{(1)}_{11}x_1 + W^{(1)}_{12}x_2 + W^{(1)}_{13}x_3 + b^{(1)}_1) \\
a^{(2)}_2 &= f(W^{(1)}_{21}x_1 + W^{(1)}_{22}x_2 + W^{(1)}_{23}x_3 + b^{(1)}_2) \\
a^{(2)}_3 &= f(W^{(1)}_{31}x_1 + W^{(1)}_{32}x_2 + W^{(1)}_{33}x_3 + b^{(1)}_3) \\
h_{W,b}(x) = a^{(3)}_1 &= f(W^{(2)}_{11}a^{(2)}_1 + W^{(1)}_{22}a^{(2)}_2 + W^{(1)}_{23}a^{(2)}_3 + b^{(1)}_1).
\end{aligned}$$

Además, resulta que $h_{W,b}(x) = f(W^T\overrightarrow{x}) = f\left(\sum_{i=1}^{3} W_i x_i + b\right)$, o sea, una suma ponderada de las entradas incluyendo el sesgo para i.

Este proceso es el conocido como propagación hacia delante.

2.2.2. Backward Propagation

De forma general, la propagación hacia atrás utiliza un algoritmo de aprendizaje supervisado, llamado **algoritmo de retropropagación ("backpropagation algorithm")**. Permite cambiar el peso de las conexiones para así minimizar el valor de una determinada función de costo C. Considerando una sola muestra $(x^{(i)}, y^{(i)})$ del conjunto de entrenamiento, la función de costo puede escribirse como:

$$C(W, b; x^{(i)}, y^{(i)}) = \frac{1}{2}||h_{W,b}(x^{(i)}) - y^{(i)}||^2 \tag{2.1}$$

donde $h_{W,b}(x^{(i)})$ es la salida devuelta por la red, $x^{(i)}$ es la entrada e $y^{(i)}$ es la salida que se quiere y que es parte del conjunto de entrenamiento. En general, el conjunto de entrenamiento es un conjunto de m pares $(x^{(i)}, y^{(i)})$ con $i = 1, .., m$. De esta forma, la función de costo total es:

$$C(W, b) = \left[\frac{1}{m}\sum_{i=1}^{m} C(W, b; x^{(i)}, y^{(i)})\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}(W_{ji}^{(l)})^2 =$$

$$= \left[\frac{1}{m}\sum_{i=1}^{m}\left(\frac{1}{2}||h_{W,b}(x^{(i)}) - y^{(i)}||^2\right)\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}(W_{ji}^{(l)})^2 \tag{2.2}$$

El primer término de la fórmula (2.2) es la media de la suma de los errores al cuadrado, queriéndo decir por error a la diferencia entre el valor real de la salida y la salida deseada. El segundo término es un término de regularización y se llama pérdida de peso, que tiende a disminuir la cantidad del peso asignado y ayuda a la reducción de situaciones en las que se produce un sobreajuste.

Antes de comenzar a entrenar una red de neuronas artificiales, es bueno inicializar cada parámetro $W_{ij}^{(l)}$ con un valor aleatorio cercano a cero y luego elegir un algoritmo para minimizar la función dada en (2.2). Usualmente se usa el **algoritmo de descenso del gradiente**.

La propagación hacia atrás comienza desde la última capa de la red, que es la capa de salida y se propaga hacia atrás hasta la capa de entrada, calculando para cada neurona de cada capa el término de error utilizado para así medir la responsabilidad de cada neurona en un posible error en la salida de la red. La propagación hacia atrás se realiza después de una propagación hacia delante, donde se calculan todos los parámetros W y b.

2.2.3. Actualización de los pesos

Al utilizar los términos del error calculados en la propagación hacia atrás, se pueden calcular las derivadas parciales, es decir, los gradientes con respecto a W y a b de la función de costo C. A través de estas derivadas, se pueden actualizar el valor de los pesos. Una sola iteración del descenso del gradiente actualiza los parámetros W y b como se muestra en (2.3).

$$W_{ij}^{l} = W_{ij}^{l} - \alpha \frac{\partial}{\partial W_{ij}^{l}} C(W, b)$$

$$b_i^{(l)} = b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} C(W, b) \tag{2.3}$$

donde $C(W, b)$ viene dada en (2.2) siendo obvio que la función de costo debe de ser diferenciable, mientras que α es un parámetro conocido como tasa de aprendizaje, que se utiliza para especificar el grado de aprendizaje, o mejor dicho, la actualización respecto a los parámetros W y b.

Todo el proceso de la propagación hacia delante y la propagación hacia atrás con la correspondiente actualización de los pesos, mejora los parámetros W y b con el objetivo de minimizar la función de costo.

Capítulo 3

Redes de Neuronas Convolucionales (CNN)

3.1. Introducción

Desde 2012, uno de los avances más importantes en el aprendizaje profundo es el uso de las redes de neuronas convolucionales para obtener mejores resultados en el reconocimiento de imágenes.

Las redes de neuronas convolucionales (CNN) presentan múltiples capas para calcular la salida dado un conjunto de datos. El desarrollo de este modelo comenzó en la década de 1950, donde los investigadores Hubel y Wiesel modelaron la corteza visual animal.

En un artículo en el año 1968, ambos debatieron sus hallazgos, los cuales identificaron tanto células simples como complejas dentro de los cerebros de monos y gatos que ellos estudiaron. Observaron que las células simples tuvieron un rendimiento maximizado. Por otro lado, se observó que el cuerpo receptivo en las células complejas era considerablemente más grande, y sus salidas no se veían afectadas por las posiciones de los bordes dentro del cuerpo receptivo.

Además del reconocimiento de imágenes, para lo cual las redes de neuronas convolucionales fueron originalmente creadas, tienen otras aplicaciones considerables, como dentro de los campos del procesamiento natural del lenguaje y aprendizaje reforzado.

3.2. Visión artificial

La visión artificial (*CV : Computer Vision*) es un campo que se ocupa de la adquisición, procesamiento y extracción de información de imágenes. La visión artificial es un campo muy diverso que utiliza técnicas e ideas de las más diversas disciplinas científicas. La parte de la elaboración de imágenes utiliza técnicas de procesamiento de señales para poder realizar mejoras y/o cambios en la imagen original con el fin de hacerla más adecuada para extraer información. La extracción de los datos se puede considerar como la parte final de la visión artificial y es tarea de la minería de datos. Una vez la información ha sido extraída de la imagen, es posible usarla en los campos más dispares: orientación de robots, exploración automática de áreas mediante sistemas inteligentes, conducción automática de automóviles. . .

Tanto para procesar la imagen como para extraer información, la técnica más utilizada es la convolución en dos dimensiones. La información extraída viene almacenada en vectores de características, que se usan como elementos que representan el objeto identificado. Esto último es el punto de encuentro entre el procesamiento de imágenes, la minería de datos y el aprendizaje automático.

3.2.1. Convolución

En el campo de la teoría de la señal, la convolución ocurre entre dos señales continuas unidimensionales f(t) y g(t). Para comprender el significado de la operación de convolución entre señales es necesario introducir el concepto de sistema lineal de tiempo invariante (LTI). La teoría de la señal ofrece las herramientas para analizar el comportamiento de los sistemas físicos observando su comportamiento en respuesta a ciertos estímulos de entrada.

Definición 3.1 Sea S el sistema a modelar. S acepta un estímulo en la entrada y produce una señal de salida y(t). Se dice que S es un sistema LTI si se verifican las siguientes propiedades:

- 1. Linealidad: $S[\alpha x_1(t) + \beta x_2(t)] = \alpha y(x_1(t)) + \beta y(x_2(t)), \alpha, \beta \in \mathbb{R}$
- 2. Tiempo de invarianza: $S[x(t + t_0)] = y(t + t_0)$

Se puede analizar el comportamiento de un sistema LTI analizando su respuesta a la función Delta de Dirac $\delta(t)$. $\delta(t)$ es una función que vale cero en cualquier punto de su dominio, excepto en el punto cero, donde su valor es un infinito de un grado lo suficientemente alto como para hacer realidad la función:

$$\int_{-\infty}^{\infty} \delta(t)\phi(t)dt = \phi(0) \qquad (3.1)$$

Intuitivamente, aplicar $\delta(t)$ a una función $\phi(t)$ es equivalente a considerar el valor de $d\phi(t)$ en 0. Si en la entrada del sistema S establecemos $\delta(t)$ obtendremos la respuesta del sistema con un impulso unitario centrado en el cero.

Definición 3.2 Se define la respuesta impulsiva de un sistema a la salida del sistema cuando la función Delta de Dirac está presente en la entrada.

$$h(t) = S[\delta(t)] \qquad (3.2)$$

La respuesta impulsiva tiene mucha importancia ya que nos permite calcular la respuesta del sistema LTI en cualquier entrada. Una señal genérica $x(t)$ se puede expresar mediante la aplicación $\delta(t)$ trasladada en cualquier punto del dominio de $x(t)$.

$$x(t) = \int_{-\infty}^{\infty} x(t)\delta(t - \tau)d\tau = x(t) * \delta(t) \qquad (3.3)$$

La operación que se acaba de describir es la de la convolución entre dos señales (denotada por el símbolo $*$). Es evidente que $\delta(t)$ es un elemento neutro. Ahora es posible analizar la convolución para comprender el motivo por el que es la operación fundamental en el análisis de las señales. Si $x(t)$ es una señal genérica de entrada y $h(t)$ es la respuesta impulsiva del sistema S, entonces el resultado de la convolución

$$y(t) \stackrel{def}{=} x(t) * \delta(t) = (x * h)(t) = \int_{-\infty}^{\infty} x(t)\delta(t-\tau)d\tau \tag{3.4}$$

representa el comportamiento del sistema LTI modelado por su respuesta impulsiva $h(t)$ cuando $x(t)$ se coloca en su entrada. Este resultado es de fundamental importancia ya que muestra como la respuesta impulsiva caracteriza totalmente el sistema.

La convolución unidimensional discreta también se puede definir. Si $g(n)$ y $x(n)$ son definidos en $\mathbb{Z}$ entonces es posible calcular su convolución mediante:

$$(x * g)[n] \stackrel{def}{=} \sum_{m=-\infty}^{\infty} x[m]g[n-m] \tag{3.5}$$

La generalización al caso bidimensional es natural, ya que en el campo de la visión artificial la convolución se lleva a cabo principalmente entre señales discretas bidimensionales. La parte discreta del Delta de Dirac y del Delta de Kronecker, puede expresarse independientemente de la dimensión del espacio que se utiliza:

$$\delta_{i,j} \stackrel{def}{=} \begin{cases} 1 & \text{si i} = j \\ 0 & \text{en otro caso} \end{cases} \tag{3.6}$$

En el caso bidimensional, el tratamiento para los sistemas LTI es generalizado y hablamos de sistemas lineales de espacio invariante (LSI).

Definición 3.3 Se define filtro o núcleo o matriz de convolución a la respuesta de un sistema LSI discreto. El filtro tiene un tamaño de $2k \times 2k$ donde k es un valor arbitrario, que define cuántos valores de la respuesta impulsiva hay en la muestra.

Sea $h[n]$ un filtro de dimensiones $2k \times 2k$ e I una imagen a escala de grises (bidimensional), cada punto de coordenadas (i, j) con señal resultante de la convolución entre h e I viene dada por:

$$O(i,j) = \sum_{u=-k}^{k} \sum_{v=-k}^{k} h(u,v)I(i-u, j-v) \tag{3.7}$$

Intuitivamente, una convolución bidimensional consiste en filtrar una imagen de dimensiones $(2k+1) \times (2k+1)$ en la imagen I y para cada píxel centrado en dicha imagen, calcular la operación que se ha indicado en (3.7). Se pueden definir algunos parámetros para la convolución para cada dirección w y h tales como:

Paso $S_{w,h}$. Es la distancia expresada en píxeles que transcurre entre aplicaciones sucesivas (en la dirección dada) en el paso elemental de la convolución. Intuitivamente es la distancia entre el centro de la imagen que se filtra cuando esta pasa de la posición (i, j) a la posición $I_{i+S_w,j}$ (o $I_{i,j+S_h}$ dependiendo de la dirección en la que se mueva la imagen).

Relleno cero $P_{w,h}$. Número de ceros para añadir como borde al resultado de la convolución.

Asumiendo $S = S_h = S_w, P = P_h = P_w, I = I_h = I_w$, la imagen filtrada tendrá dimensiones:

$$O_w = O_h = \frac{I_s - k + 2P}{S} + 1 \qquad (3.8)$$

El resultado de (3.8) es un valor entero si se ha elegido el valor del paso correctamente para ser compatible con el tamaño de la imagen de entrada.

La parte de la visión artificial que se ocupa del procesamiento de imágenes ha sido a lo largo de los años, definir filtros manualmente con el objetivo de mejorar la calidad de las imágenes o para extraer sus características. Entre los ejemplos más conocidos encontramos el filtro Gaussiano para la reducción del ruido y el filtro Sobel para la extracción de las derivadas direccionales de la imagen. En la imagen 3.1 se muestra el resultado de la convolución de una imagen con núcleo de detección de bordes.

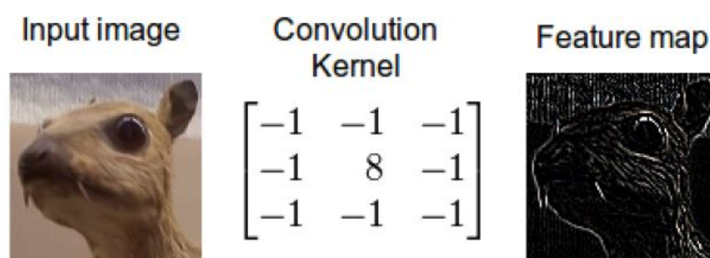


Figura 3.1: Extracción del borde de una imagen mediante convolución.

Extraer funciones de una imagen es un paso fundamental para poder equipar máquinas de percepción visual. De hecho, después de extraer ciertas características es posible analizar el contenido de la imagen y por lo tanto, hay que asegurarse de que la máquina pueda dar un significado a los objetos que aparecen dentro de la imagen (por lo que, tal vez los clasifiquen) y no sólo considerar la imagen como un conjunto $I_w \times I_h$ de valores ordenados.

Antes de profundizar en los filtros de convolución, es necesario extender la definición de convolución (3.7) a imágenes con más de una dimensión. A continuación no nos referiremos a imágenes, sino a volúmenes tridimensionales para que sea lo más general posible.

3.2.1.1. Convolución entre volúmenes

Un volumen es un paralepípedo rectangular completamente determinado por la tripleta (W, H, D) donde:

- $W \geq 1$ es la anchura.
- $H \geq 1$ es la altura.
- $D \geq 1$ es la profundidad.

Intuitivamente, una imagen con escala de grises es un volumen con $D = 1$, mientras que una imagen de 3 canales (RGB) es un volumen con $D = 3$.

La convolución para un volumen con $D = 1$ es la indicada en (3.7). La definición puede generalizarse a volúmenes con D arbitraria aplicando (3.7) para cada valor de D y sumando los resultados para obtener un único valor como resultado de la convolución. Incluso el filtro ahora es un volumen necesariamente con profundidad D, por lo que se obtiene un filtro bidimensional para cada nivel bidimensional del volumen de entrada.

Sea $I = (W_I, H_I, D_I) = \{I_1, .., I_D\}$ el volumen de entrada, sea $F = (2k, 2k, D_I) = \{F_1, .., F_{D_I}\}$ el volumen del filtro, que pertenece al conjunto de volúmenes de los filtros $\{(2k, 2k, D_I)\}^{F_d}$. Ahora el resultado de la convolución entre el volumen I y F pertenece a $\{(2k, 2k, D_I)\}^{F_d}$, donde el n-ésimo volumen del filtro viene dado por:

$$O_n(i,j) = \sum_{d=1}^{D_I} \sum_{u=-k}^{k} \sum_{v=-k}^{k} F_d(u,v) I_d(i-u, j-v) \tag{3.9}$$

El valor de los volúmenes de los filtros depende de los valores de entrada en el entorno bidimensional del punto en el que estamos calculando la convolución a lo largo de toda la profundidad del volumen de entrada. Tenemos que el resultado de la convolución para todos los volúmenes del filtro, es un volumen de profundidad F_d dado por la unión de los resultados convolucionales de (3.9):

$$O = \cup_{i=1}^{F_d} O_i \tag{3.10}$$

Como en el caso unidimensional, el mapa de características resultante habrá dado dimesiones dependiendo del relleno cero P y del paso S elegidos. Intuitivamente, las características que se pueden extraer de una convolución son solamente capaces de "ver", es decir, la convolución captura información espacial dada por las dimensiones del filtro de convolución combinándolo con la información presente en cada nivel, en la misma posición espacial, del volumen de entrada.

3.2.1.2. Aprendizaje de filtros de convolución

De (3.9) sabemos que la convolución de un volumen con profundidad D requiere D filtros para producir un volumen con $D = 1$. Entonces la convolución extrae sólo una característica del volumen de entrada. Esta característica puede o no ser suficiente para ser utilizada como único vector de características por un algoritmo de *Machine Learning*.

Las características extraídas de la operación de convolución tienen una disposición espacial, y por lo tanto, no son vectores de características, sino mapas de características. Es posible obtener un vector de características de un mapa de características linealizando el mapa $O \times O$ en un vector O^2.

Si estamos interesados en buscar características básicas y elementales, como las líneas horizontales o verticales, o un color en particular, entonces una sola característica puede ser suficiente. Sin embargo, si estamos interesados en la investigación de un patrón más complejo (una persona, una máquina ...) una sola característica no es suficiente para caracterizar la complejidad y la variabilidad (posiciones, dimensiones o colores) de la imagen.

Definir manualmente los filtros capaces de extraer características es un proceso largo y complejo que no siempre conduce a los resultados deseados. Por esta razón, se puede usar *Machine Learning* para aprender los valores de los filtros, para que la combinación de las características extraídas de los filtros pueda caracterizar completamente a la imagen que está en estudio.

Para hacer esto, es posible utilizar las redes de neuronas artificiales. Al colocar las neuronas en cuadrículas $2k \times 2k$, pueden usarse como filtros de convolución para la

extracción de las características y modificar sus valores (y por lo tanto, las características extraídas) durante los pasos de la propagación hacia atrás. El proceso de entrenamiento mostrará las imágenes a la red de los procesos a clasificar y el valor de la etiqueta real y. La pérdida también requiere el valor de la predicción, por este motivo es necesario que se agreguen capas de clasificación totalmente conectadas a la red para realizar la combinación de las características extraídas y obtener el valor de la clase y.

Es evidente que no es posible saber a priori cuántas características serán necesarias extraer para caracterizar completamente a las entradas. Esto lleva a la introducción de un hiperparámetro adicional que indicaremos con F_d (profundidad del filtro), que es la cantidad de filtros de convolución que se intentan aprender. Cada uno de estos filtros está compuesto de D filtros $2k \times 2k$. Es evidente por lo tanto, que una convolución que acepta un volumen de entrada $(W; H; D)$ se lleva a cabo con F_d filtros, cada uno de los cuales está compuesto por D filtros, produce un volumen a su vez con profundidad F_d. El conjunto de los F_d filtros se denomina capa de convolución. Durante el paso hacia delante, el volumen de la entrada tiene un filtro extraído de la capa de convolución (con igual profundidad a la profundidad del volumen de entrada). El resultado de la convolución con el filtro genera un mapa de características. El conjunto de mapas de características, generado por F_d filtros genera un volumen con profundidad F_c que a su vez puede ser utilizado para extraer otras características.

Como es fácil de intuir, el número de parámetros aumenta a mayor número de capas de convolución. Un método para reducir el número de parámetros sin reducir la efectividad del aprendizaje es el *pooling*, que se estudiará más adelante.

3.3. Estructura y propiedades de las CNN

Las redes de neuronas convolucionales son en términos generales, modelos de redes de neuronas artificiales multicapa. De acuerdo con la estructura de la corteza visual animal describida por Hubel y Weisel, el modelo puede ser interpretado visualmente como se muestra en la siguiente figura.

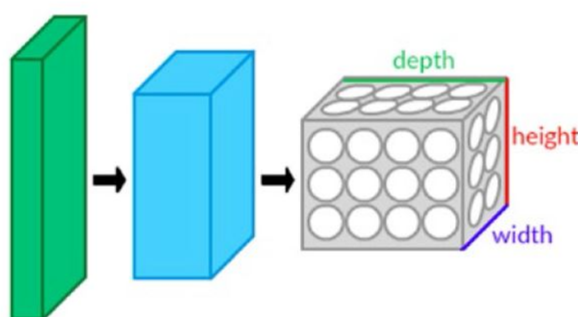


Figura 3.2: Visualización de una CNN.

Cada uno de los bloques representa una capa diferente de la red de neuronas convolucional, los cuales tienen altura, anchura y profundidad. De izquierda a derecha podemos observar, las capas de entrada, las capas ocultas (convolucional, agrupación y capas de abandono) y las capas completamente conectadas. Después de la última capa, el modelo genera una clasificación.

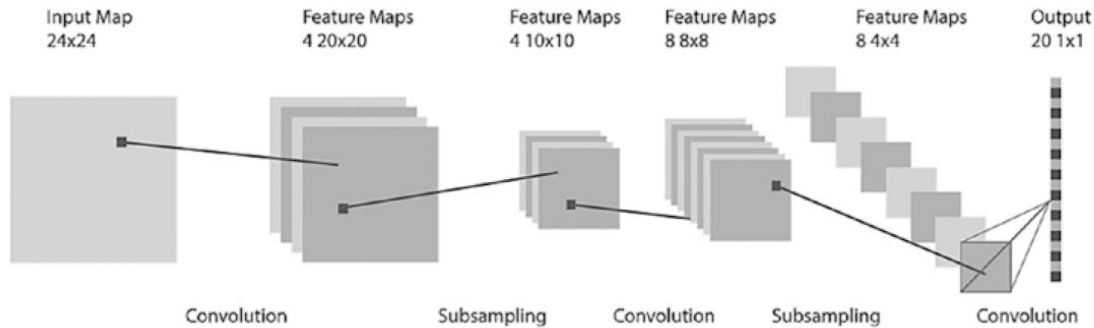


Figura 3.3: Arquitectura de una CNN.

Las capas completamente conectadas imponen la conectividad local entre las neuronas y las capas adyacentes. Como tal, las entradas de las capas ocultas son un subconjunto de neuronas de la capa que precede a dicha capa oculta. Esto asegura que las neuronas del subconjunto de aprendizaje produzcan la mejor respuesta posible. Además las unidades comparten el mismo peso y sesgo en el mapa de activación, de modo que todas la neuronas que estén en una capa dada están analizando la misma característica.

3.4. Componentes en la arquitectura de una CNN

3.4.1. Capa convolucional (*Convolutional layer*)

Es la capa principal de las redes de neuronas convolucionales, siendo indispensable el uso de una o de más capas de este tipo en dichas redes. Los parámetros en una capa convolucional en la práctica se refiere a un conjunto de filtros entrenable. Cada filtro ocupa un espacio pequeño a lo largo de las dimensiones de anchura y altura, pero se extiende por toda la profundidad del volumen de entrada al que se aplica. Durante la propagación directa o hacia delante, cada filtro a lo largo de la anchura y de la altura del volumen de entrada, produce un mapa de activación bidimensional (o mapa de características) para ese filtro. A medida que el filtro se mueve a lo largo del área de la entrada, se efectúa un producto escalar entre los valores del filtro y los de la región de entrada a la que se aplica.

De forma intuitiva, la red tendrá como objetivo el aprendizaje de filtros que se activan en presencia de alguna función específica en una región específica de la entrada.

Poner en cola todos estos mapas de características para todos los filtros, a lo largo de la dimensión de la profundidad forma el volumen de salida de una capa convolucional. Cada elemento de este volumen puede ser interpretado como la salida de una neurona que solamente se observa en una pequeña región de entrada y que comparte sus parámetros con otras neuronas en el mismo mapa de características, ya que todos estos valores provienen de la aplicación de un mismo filtro.

Por ejemplo, supongamos que se está tratando de clasificar una imagen, como el número 1 o como el número 0, y la imagen en realidad es un 1. Supongamos que la imagen tiene el fondo de color negro, pero el dígito tiene píxeles blancos. En la siguiente figura podemos observar lo que se está describiendo.

Figura 3.4: Imagen ejemplo de un 1.

El ordenador asociará los píxeles blancos al valor 1, y los píxeles negros al valor -1. Cuando introducimos una imagen a la capa convolucional, el modelo extrae las características únicas de dicha imagen, que en general, son los colores, las formas y los bordes los que definen la imagen. Una vez que tengamos las características, se realiza lo que se conoce como filtrado sobre esta imagen. La filtración es el proceso de tomar una característica de la imagen, que en este caso podemos suponer que sea un píxel de 3×3 cuadrado y combinarlo con un parche de dicha imagen, que también será un píxel de 3×3 cuadrado. En la siguiente figura podemos observar como se ve este proceso de filtrado.

Figura 3.5: Ejemplo de un filtrado.

Luego, multiplicamos el número del píxel de la característica por el número correspondiente al píxel del parche de la imagen. En nuestro ejemplo, deberíamos de conseguir una salida de 1 o -1 por cada operación. Cuando los píxeles coinciden exactamente, la salida debe ser 1, y cuando no lo hacen, debe ser -1. Por último, se toma la media de los productos de los píxeles. Si una imagen coincide exactamente, esta media debe de ser 1, y si no es así, debe ser menor que 1.

Supongamos que el parche de la imagen y la característica seleccionada no coinciden en absoluto. En este caso, cuando se toma la media, debe de salir -1. Colocamos este producto en el centro de la posición del parche de la imagen junto con una determinada característica, esto es lo que se conoce como un mapa de características.

Esto será el resultado de la capa de convolución y es lo que se utilizará en la siguiente capa. La capa convolucional sobre distintas operaciones producirá múltiples mapas de características. El proceso de hacer coincidir una característica con el parche de una imagen sobre cada posición posible se conoce como la convolución de una imagen. El mapa de activación/características para una red de neuronas convolucional se denota como:

$$h^k_{i,j} = tanh((w^k x)_{i,j} + b_k)$$

donde w_k es el peso, b_k es el sesgo, x es el valor del píxel específico y $tanh$ es para la no linealidad de los datos. Los subíndices i, j se refieren a la entrada de la matriz que

representa el mapa de activación/características. El peso w_k es lo que conecta en último lugar los píxeles en el mapa de características de la capa anterior.

La capa de convolución es en última instancia una pila de mapas de características que se han obtenido en la operación descrita anteriormente. Luego se pone el mapa de características en la capa de agrupación. Se calcula el tamaño espacial del volumen de salida como:

$$SpatialSize_{Output} = \frac{W - F + 2P}{S + 1}$$

donde W es el tamaño del volumen de entrada, F es el tamaño del cuerpo receptivo en la capa de convolución, P es el relleno cero y S es el paso.

3.4.2. Capa de agrupación (*Pooling layer*)

Entre sucesivas capas convolucionales, es usual colocar lo que se denomina como capa de agrupación en el medio de dichas capas. De forma concisa, la capa de agrupación toma los mapas de características producidos en la capa de convolución y los agrupa en una imagen. En esta capa, se produce la reducción de dimensionalidad reduciendo de esta forma, la complejidad que posee el modelo ayudando a evitar el sobreajuste del mismo. Es decir, lo que hacen las capas de agrupación es simplificar la información en la salida de la capa convolucional.

El *pooling* es una operación que permite reducir la cantidad de parámetros de la red, por lo general, la operación describe la agrupación de manera operacional. Es posible imaginar la operación de agrupación con la de convolución entre volúmenes. Dado un volumen $I = (W_I; H_I; D)$ la operación de agrupación produce un volumen de salida $O = (W_O; H_O; D)$ deslizando un filtro $2k \times 2k$ que no se aprende, sino que solamente se utiliza para definir el cuerpo receptivo de la operación.

La operación de agrupación en un volumen genérico $I = (W, H, D) = \{I_1, .., I_D\}$ se puede representar como la convolución de I con un volumen de D filtros D_i idénticos. El filtro representa el muestreo de la respuesta impulsiva de la función f que se desea aplicar para reducir el número de parámetros. Sea FV el volumen del filtro, formado por D filtros idénticos a F. Ahora:

$$O_n(i, j) = \sum_{d=1}^{D_I} \sum_{u=-k}^{k} \sum_{v=-k}^{k} F(u, v) I_d(i - u, j - v) \tag{3.11}$$

Como en el caso de (3.7), se aplica para cada píxel del volumen de entrada deslizando el filtro con cierto paso S y rellenando el resultado con P ceros. El resultado de la operación de agrupación con $f = max$ se muestra en la figura 3.6.

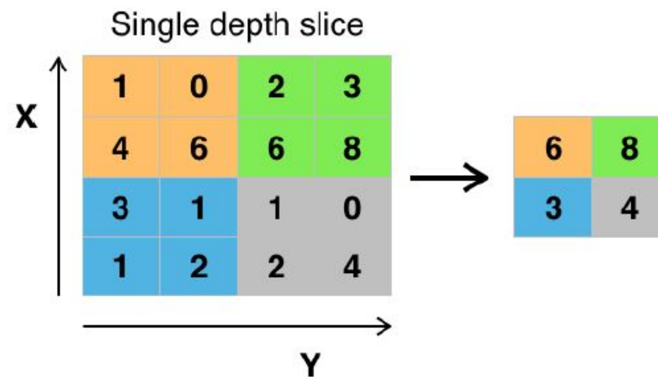


Figura 3.6: Max-pooling: filtro 2×2 y S=2.

Intuitivamente, la operación de agrupación consiste en extraer regiones $2k \times 2k$ y aplicar una función f al mapa $2k \times 2k$ seleccionando un solo valor, moviendo S píxeles y repitiendo la operación. Esto se hace para todos los niveles del volumen de entrada. La agrupación por tanto conserva la profundidad del volumen de entrada.

Los valores de los parámetros que se usan con más frecuencia son:

- Filtro de dimensiones 2×2.
- Paso igual al lado del filtro, $S = 2$.
- Función $f : \mathbb{R}^4 \rightarrow \mathbb{R}, (x_1, x_2, x_3, x_4) \rightarrow max(x_1, x_2, x_3, x_4)$.

Al hacerlo, el volumen resultante tendrá unas dimensiones $(W_0 = \frac{W}{2}, H_0 = \frac{H}{2}, D)$. Esto permite reducir en un 75 % el número de parámetros de entrada y así seguir entrenando las siguientes capas con muchos menos parámetros para realizar el aprendizaje. La agrupación con $f = max$ se denomina **max-pooling**. Este tipo de agrupación ha demostrado dar los mejores resultados hasta la fecha en la práctica. Sin embargo, no hay impedimento en usar cualquier otra función y observar el comportamiento del modelo en sus diversas fases.

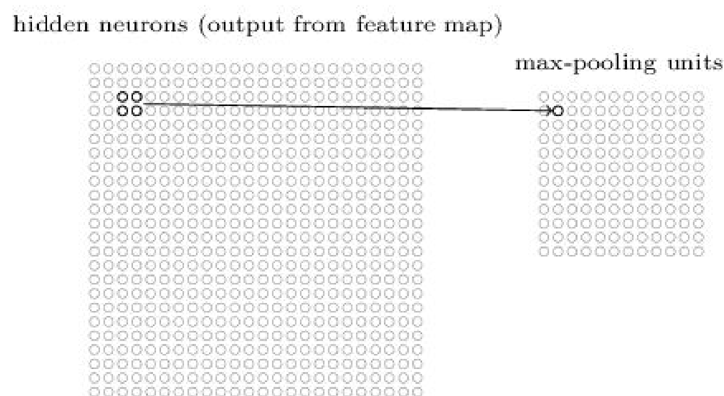


Figura 3.7: Max-pooling.

Nótese que en esta última imagen que se tenían 24×24 neuronas en la salida de la capa convolucional y después de la agrupación se tienen 12×12 neuronas.

Como se mencionó anteriormente, la capa convolucional involucra a más de un mapa de características. Se aplica *max-pooling* a cada mapa de características por separado. Enton-

ces, si hubiese tres mapas de características, la combinación de las capas convolucionales y *max-pooling* se vería así:

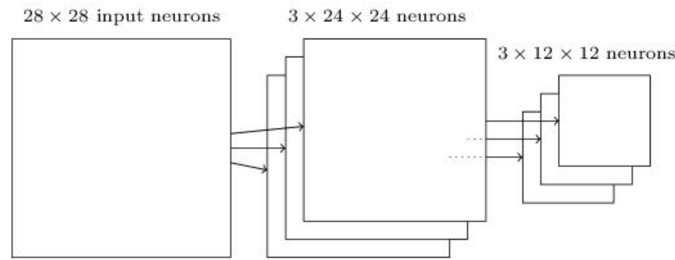


Figura 3.8: Max-pooling con 3 mapas de características.

Se puede pensar que *max-pooling* es una manera de que la red pregunte si una determinada característica se encuentra en algún lugar de una región de la imagen. Lo que se intuye es que una vez se ha encontrado una característica, su ubicación exacta no es tan importante como su ubicación aproximada en relación con otras características. Una gran ventaja es que hay muchas menos características agrupadas, por lo que esto ayuda a reducir la cantidad de parámetros necesarios en las capas posteriores.

Max-pooling no es la única técnica utilizada en la agrupación. Otra técnica es conocida como agrupación *L2* (*L2-pooling*). En este caso, en lugar de tomar la activación máxima de una región específica de neuronas 2×2, se toma la raíz cuadrada de la suma de los cuadrados de las activaciones 2×2. Si bien los detalles son diferentes, la intuición es similar a *max-pooling*, la agrupación *L2* es una forma de condensar la información de la capa convolucional.

La función que determina el tamaño espacial de la salida viene dada por:

$$SpatialSize_{Output} = W_2 \cdot H_2 \cdot L_2$$

donde

$$W_2 = \frac{w_1 - F}{S + 1}, H_2 = \frac{H_1 - F}{S + 1}, L_2 = L_1$$

3.4.3. Capa de unidades lineales rectificadas ReLu (*Rectified Linear Units Layer*)

ReLu es la abreviatura de *Rectified Linear Units*, es decir, unidades lineales rectificadas. Este tipo de capas es muy común en una red de neuronas convolucional y se puede utilizar múltiples veces en una misma red de neuronas artificiales, frecuentemente después de cada capa convolucional.

La función de esta capa es la de aumentar la propiedad de no linealidad de la función de activación, sin que se modifiquen los cuerpos receptivos de la capa convolucional.

Una función muy utilizada en las capas ReLu es $f(x) = max(0, x)$, aunque también es posible otras funciones como $f(x) = tanh(x)$, $f(x) = |tanh(x)|$ o la función sigmoide $f(x) = \frac{1}{1+e^{-x}}$. Además es posible utilizar en una capa una función cualquiera y en otra de ellas otra diferente.

En cuanto a la función $f(x) = tanh(x)$, permiten entrenar con una mayor rapidez y con un rendimiento similar a las capas ReLu.

En este tipo de capas no existe ningún parámetro a establecer, simplemente se utiliza una función fijada. Estas capas tampoco tienen parámetros entrenables, por lo que tienen una propagación hacia atrás simple: se propagan hacia atrás los errores calculados hasta ese momento que provienen de la capa posterior, pasándolos a la capa anteriormente considerada.

3.4.4. Capa completamente conectada (*Full connected layer*)

Este tipo de capa es exactamente igual a cualquier capa de una red de neuronas artificial clásica, pero que tiene su arquitectura totalmente conectada. Todas las neuronas de esta capa se encuentran conectadas a todas las neuronas de la capa anterior y más específicamente, se encuentran conectadas a todos los mapas de características de la capa anterior.

En este tipo de capa, a diferencia de lo que se ha visto hasta ahora en las redes de neuronas convolucionales, no utiliza la propiedad de conectividad local, ya que una capa completamente conectada se encuentra conectada a todo volumen de entrada, por lo que tiene un gran número de conexiones, mientras que las capas convolucionales se encuentran conectadas a una sola región local en la entrada y que muchas de las neuronas de la capa convolucional comparten parámetros.

El único parámetro que se puede configurar en este tipo de capa es el número de neuronas que la componen. En estas capas se conectan sus K neuronas con todos los volúmenes de entrada de cada una de sus K neuronas. Su salida será un único vector $1 \times 1 \times K$, que contiene las activaciones calculadas. Este hecho, el pasar de un volumen de entrada en 3 dimensiones hasta un solo vector de salida con una sola dimensión, sugiere que después de esta capa, no haya ninguna capa de convolución.

La función principal de las capas completamente conectadas es llevar a cabo una especie de agrupación de la información que se ha obtenido hasta ese momento, que servirá en cálculos posteriores para la clasificación final.

En general, en las redes de neuronas convolucionales, se suele utilizar más de una capa completamente conectada en serie y la última de ellas tendrá el parámetro K que es el número de clases que se encuentran presentes en el conjunto de datos. Los valores finales de K serán alimentados a la capa de salida, que a través de una cierta función probabilística realizará la clasificación.

Hay más parámetros que se pueden configurar, como los valores de los pesos y el sesgo, pero no es estrictamente necesario y se pueden utilizar valores predeterminados.

3.4.5. Capa de pérdida (*Loss layer*)

En esta capa es donde se comparan las predicciones con los valores reales de las imágenes. Cuando se trata de clasificar y elegir entre K posibles niveles, se usaría un clasificador de pérdidas *softmax*. El uso de una función euclídea también es usual con el propósito de regresión contra las etiquetas de las imágenes. Sus funciones están dadas por:

- **Función de pérdida softmax**:

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

$$z_j = \sum_{k=0}^{d} W_{ik} x_k$$

donde:

1. z_j es un vector de probabilidades a posteriori.
2. j representa la $i-$ésima neurona de la capa de salida, es decir, de la capa de pérdida.
3. K representa el número total de neuronas de la capa de pérdida.
4. W_{ki} son los pesos.
5. x_i son los valores de entrada que recibe la capa de pérdida.
6. $\sigma(z)_j$ es la activación de las K neuronas de la capa de pérdida.

- **Función de pérdida euclídea**:

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

donde:

1. K representa el número total de neuronas de la capa de pérdida.
2. $\hat{y}_i$ representa las predicciones de las imágenes.
3. y_i representa los valores reales de las imágenes.

3.5. Hiperparámetros

En las secciones anteriores, se han introducido los conceptos básicos relevantes en las redes neuronales convolucionales. Cada uno de estos conceptos está definido dependiendo de uno o más grados de libertad: los hiperparámetros. La elección del valor de los hiperparámetros sigue siendo una operación vinculada a la intuición, por lo que no tiene bases matemáticas sólidas. La tabla 3.1 muestra todos los hiperparámetros relacionados con la operación de convolución entre volúmenes. Nótese que la elección del algoritmo de optimización en sí también es un hiperparámetro.

F_d	Número de filtros que forman la red convolucional
$2k \times 2k$	Área del filtro de convolución/pooling
S	Paso de la operación de convolución/pooling
P	Número de ceros de relleno para la salida de la convolución/pooling

Cuadro 3.1: Hiperparámetros de la convolución entre volúmenes.

Además de los hiperparámetros que se muestran en la tabla 3.1, cada arquitectura tiene muchos otros grados de libertad, muchos de los cuales que a continuación se enumeran

siguen siendo problemas abiertos y sus elecciones influyen en las presentaciones de las redes.

- **Dimensiones de la entrada**: las imágenes que se introducen en la red se deben de escalar antes de aplicar la primera convolución.
- **Número de capas de convolución**: al igual que es posible tener un número F_d de filtros arbitrarios en cada capa de convolución, es posible tener varias capas de convolución que combinan las F_d características extraídas de la capa anterior teniendo $F_d^{'}$ características.
- **Posición de la operación de agrupación (*pooling*)**: es posible elegir la cantidad de características mediante *pooling* después de cada capa, así como no introducir ninguna operación de *pooling*.
- **La función de *pooling*** : si se elige introducir una operación de *pooling*, la operación puede ser el máximo o cualquier otra función. En cada capa de agrupación se puede usar una función diferente.
- **Tamaño del filtro**: cada capa de convolución (y de agrupación) puede usar filtros con un cuerpo receptivo diferente en cada nivel de la red.
- **Capa totalmente conectada**: para combinar las características extraídas, la parte final de cada red consiste en capas completamente conectadas. Cuántas capas totalmente conectadas a usar y el número de neuronas que las forman es una elección totalmente arbitraria.
- **Función de pérdida**: la elección de la función de pérdida, es de lejos, lo que más afecta al comportamiento de la red en la prueba. Dependiendo de la función de pérdida elegida, las características que la red va aprendiendo van variando, al igual que las conexiones entre las neuronas de las capas completamente conectadas.
- **No linealidad**: se debe de elegir una función no lineal para la activación de las neuronas. Se puede elegir una función diferente para cada convolución o para cada capa completamente conectada.
- **Topología**: hasta ahora se ha asumido que las características extraídas de una capa de convolución necesariamente deben ser combinadas inmediatamante después de esta capa. Esto en realidad no es una elección requerida y es posible vincular las características extraídas de una capa con cualquier otra capa y/o mostrar las capas de convolución “centrales”. Se pueden combinar las características extraídas de las capas anteriores con las características que se pueden extraer de la imagen de entrada.

3.6. Arquitecturas de CNN

Ahora se puede tener una idea general de cómo se estructuran las redes de neuronas convolucionales: a partir de imágenes como datos de entrada en una capa de entrada, habrá una o más capas convolucionales, intercalando con capas ReLu, y cuando sea necesario capas de agrupación y en última instancia un conjunto de capas completamente conectadas antes de la capa de salida.

Se tienen distintas posibilidades a la hora de construir una red de neuronas convolucional. Destacan:

1. **LeNet**: desarrollada en los años 1990 por el reconocido investigador de aprendizaje profundo *Yann LeCun*. LeNet posee una arquitectura con una simplicidad relativa. El objetivo de este modelo fue clasificar códigos, leer códigos postales y la clasificación general de imágenes simples. Es considerada como la primera aplicación práctica exitosa de una red de neuronas convolucional.

En la siguiente figura se muestra un ejemplo de LeNet, donde las capas presentes se encuentran como sigue: entrada, capa convolucional, ReLu, capa de agrupación, capa convolucional y clasificador softmax.

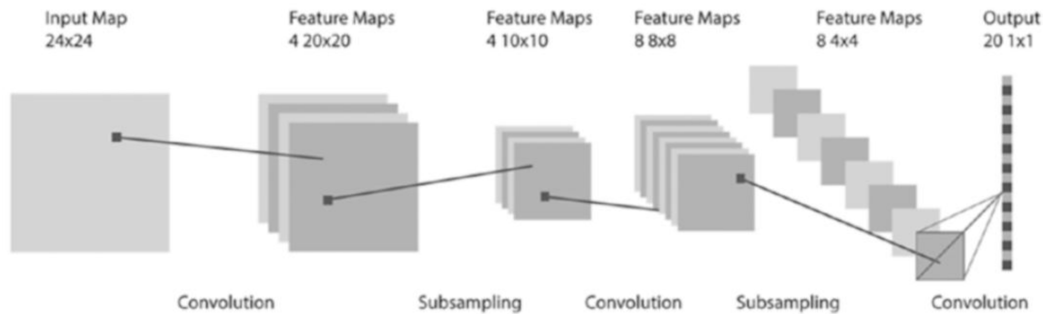


Figura 3.9: Arquitectura de LeNet.

En la figura 3.10 se observa brevemente la arquitectura LeNet5 (la quinta versión de la arquitectura LeNet inicial) para mostrar la topología de la primera arquitectura de convolución utilizada con éxito para la clasificación de imágenes de figuras escritas a mano.

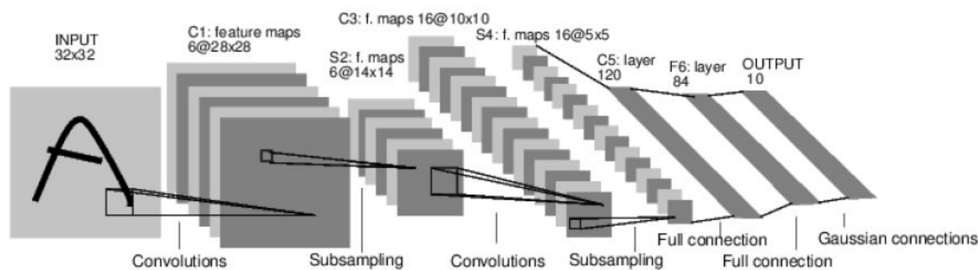


Figura 3.10: Arquitectura LeNet5.

Los hiperparámetros utilizados por el autor son:

- Dimensiones de la entrada: 32×32 en escala de grises.
- Número de capas de convolución: 2.
- Primera capa de convolución: aprende 6 filtros 5×5 produciendo 6 mapas de características 28×28.
- Primera capa de *max-pooling*: 2×2 con paso $S = 2$ para reducir a la mitad la resolución espacial de la entrada de 28×28 a 14×14.
- Segunda capa de convolución: aprende 16 filtros 5×5 produciendo 16 mapas de características 10×10.
- Segunda capa de *max-pooling*: 2×2 con paso $S = 2$, para reducir a la mitad la resolución espacial de 10×10 a 5×5.

- Número de capas totalmente conectadas: 3.
- Primera capa totalmente conectada: 120 neuronas, cuya entrada es la combinación de 16 mapas de características extraídas de los filtros (después de la disminución de la resolución realizada en el *pooling*) de la capa anterior.
- Segunda capa totalmente conectada: 84 neuronas conectadas totalmente con las 120 neuronas de la capa anterior.
- Tercera capa totalmente conectada: nivel de salida. 10 neuronas que representan 10 clases que la red puede clasificar.

El autor eligió arbitrariamente el valor de los hiperparámetros y la topología adoptada.

2. **GoogLeNet**: esta arquitectura de red neuronal convolucional ganó el premio *ImageNet Large-Scale Visual Recognition Challenge (ILSVRC)* en 2014 en honor a la LeNet de *Yann LeCun* anteriormente descrita. Fue desarrollada por *Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Ehran, Cinven Vanhoucke y Andrew Rabinovich*. El nombre de GoogLeNet proviene del hecho de que un número considerable de los desarrolladores de esta arquitectura eran trabajadores de Google Inc. Sus desarrolladores describen a GoogLeNet como una arquitectura que permite *"aumentar la profundidad y el ancho de la red manteniendo la restricción presupuestaria computacional"*.

Como se ilustra en la siguiente figura, la estructura propuesta es la siguiente: capa de entrada, capa convolucional, capa de agrupación máxima, capa convolucional (con función el máximo), comienzo (con dos capas), capa de agrupación máxima, comienzo, comienzo (con 5 capas), capa de agrupación máxima, comienzo (con 2 capas), capa de agrupación media, *dropout* , ReLu, clasificador softmax.

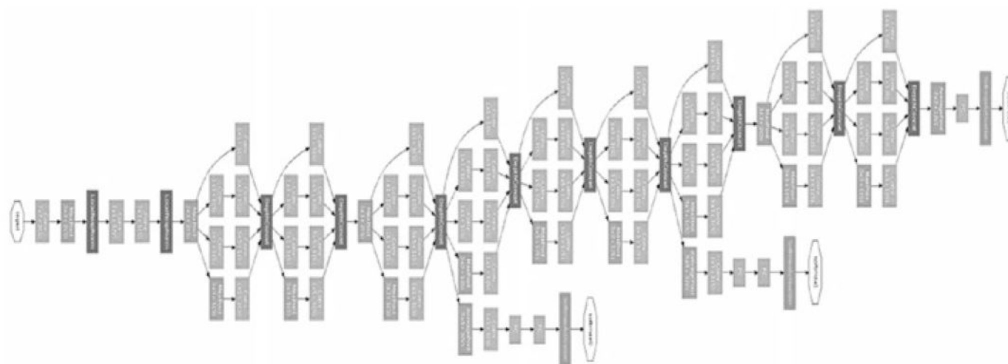


Figura 3.11: Arquitectura GoogLeNet

El enfoque de la arquitectura inicial es que a través de la orientación en las capas como se ha descrito anteriormente, la red de neuronas convolucional permite aumentar el número de unidades en cada etapa sin hacerlo hasta el punto en el que el modelo se vuelva demasiado complejo. En general, el modelo busca procesar información visual a varias escalas y luego agregar cálculos a la siguiente etapa, así que los niveles más altos de abstracción son analizados simultáneamente.

3. **AlexNet**: desarrollada por *Alex Krizhevsky, Ilya Sutskever y Geoffrey Hinton*, ganó el *ILSVRC* en 2012. Similar a la arquitectura LeNet, AlexNet utiliza "neuronas no saturadas" e implementa eficientemente la GPU (unidad de procesamiento gráfico)

para las capas de convolución. Las neuronas de las capas completamente conectadas, se encuentran conectadas a todas las neuronas de la capa anterior, las capas de respuesta/normalización siguen a la primera y a la segunda capa convolucional, y el núcleo de la segunda, cuarta y quinta capa son conectadas únicamente al núcleo de la capa anterior, que estaría en la misma GPU. La arquitectura se puede ver en la siguiente figura: cinco capas convolucionales, 3 capas completamente conectadas, la salida es un clasificador *softmax* de 1000 vías.

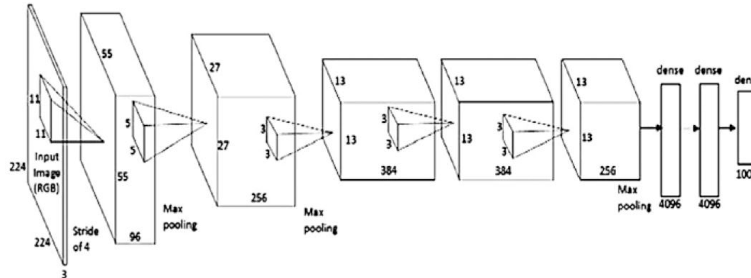


Figura 3.12: Arquitectura AlexNet

4. **VGGNet**: desarrollada por *Karen Simonyan* y *Andrew Zisserman* de la Universidad de Oxford. Llevó a segundo lugar a AlexNet en el *ILSVRC* en el año 2014. El cuerpo receptivo es 3×3, con filtros de 1×1, el paso es 2 y el tamaño máximo de agrupación es 2×2. La arquitectura es tal que la entrada se alimenta a través de varias capas convolucionales a tres capas completamente conectadas (la primera y la segunda capa tienen 4096 canales y el final es una capa *softmax* que realiza una clasificación de 1000 vías).

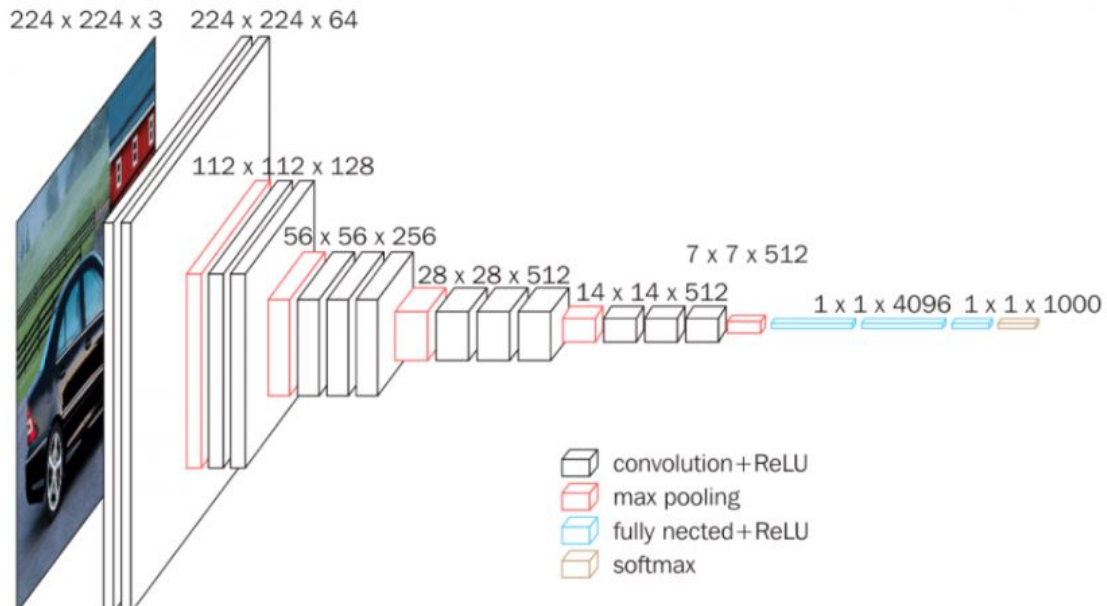


Figura 3.13: Arquitectura VGGNet

5. **ResNet**: ganador del primer premio del *ISLVRC* en el año 2015. ResNet cuenta con 152 capas, una cantidad muy superior a las capas de las redes anteriormente mencionadas. Fue desarrollada por *Kaimin He, Xiangyu Zhang, Shaoqing Ren y Jian Sun*, todos ellos investigadores de Microsoft. El objetivo de este tipo de red es formar una red que aprende funciones residuales con referencias en las entradas de

la capa, en lugar de una red que aprende funciones sin referencia. El resultado final es una red que es considerablemente más fácil de enseñar, más fácil de optimizar y con esto se gana una mayor precisión a partir de una mayor profundidad, en lugar de una red que pierde precisión a medida que se gana en profundidad.

3.7. Convoluciones 1×1

Hasta ahora, se suponía que el final de la red debe ser necesariamente una o más capas totalmente conectadas. Lógicamente es así ya que la tarea de la red es clasificar, sin embargo, es posible reemplazar las capas totalmente conectadas por capas de convolución con volúmenes $1 \times 1 \times F_d$ (para aprender) cuyo resultado es el mismo que se puede obtener de una capa totalmente conectada, pero que no limita a la red a tener unas dimensiones de entrada fijas.

Volvamos a la arquitectura de LeNet5: la primera capa totalmente conectada combina las 16 características extraídas de filtros 5×5 con 120 neuronas. Esta combinación está hecha extrayendo 120 funciones del volumen $5 \times 5 \times 16$. La única forma de poder extraer estas características es realizar una convolución de entrada $(5; 5; 16)$ con una capa de de convolución de dimensión $5 \times 5 \times 120$ para que el resultado de la convolución entre volúmenes sea un volumen de dimensiones $(1; 1; 120)$.

Considerando el volumen resultante como un tensor de rango 3, es posible eliminar las dimensiones unitarias y considerar las características extraídas como un vector unidimensional de 120 características. Luego se sigue con la creación de la capa totalmente conectada, utilizando el producto matricial de este vector para una matriz de pesos.

Si en cambio consideramos un volumen en su totalidad, sin eliminar las dimensiones unitarias, entonces se puede obtener una conexión completa entre los valores de entrada y las neuronas de una capa posterior, mediante el uso de convoluciones entre volúmenes. De hecho si consideramos un volumen de entrada $I = (1; 1; N)$ es posible definir una capa de convolución de dimensiones $1 \times 1 \times M$ (con M número arbitrario de filtros para aprender, que están formados por neuronas individuales) y efectuar la convolución del volumen de entrada para esta capa de convolución.

De la definición de convolución entre volúmenes (3.9) se sabe que cada filtro se enreda con cada nivel del volumen de entrada y el resultado de la suma de las convoluciones. Teniendo filtros de dimensión unitaria, la convolución coincide con el valor del filtro (un escalar) por cada uno de los valores de entrada (escalar): al hacerlo, se obtiene una capa totalmente conectada utilizando únicamente convoluciones.

Hay dos razones por las cuales no se deberían de utilizar capas totalmente conectadas al final de una red de neuronas convolucional y utilizar en su lugar convoluciones $1 \times 1 \times M$:

- Implementaciones eficientes: todos los marcos de aprendizaje automáticos aplicados a la visión artificial, ofrecen operaciones de convolución entre volúmenes optimizándolo lo máximo posible, dada la frecuencia con la que se utilizan. Estas mismas optimizaciones no siempre se realizan para operaciones menos frecuentes, como productos matriciales.
- Dimensiones de la entrada: utilizando convoluciones 1×1, incluso si la red es entrenada con imágenes de tamaño fijo, en la fase del test se pueden proporcionar imágenes con cualquier dimensión. La razón es que la operación de convolución se

deslizará a lo largo de cada dimensión del volumen moviéndose un píxel a la vez en cada dirección, lo cual no es posible con una capa totalmente conectada, ya que la salida es siempre unidireccional. El resultado por lo tanto, en el caso de una imagen de entrada de dimensiones mayor que lo esperado por la red no será un volumen $(1; 1; M)$ sino un volumen $(O_w; O_h; M)$

Está comprobado empíricamente que cuando múltiples características complejas se combinan, mayor es la capacidad de aprendizaje de la red. Las características vienen dadas por la combinación de otras más simples. Para combinarlas, es necesario poner en cascada capas de convolución que combinen las características extraídas. El aumento de las capas de convolución y de las capas totalmente conectadas ha dado lugar a la rama del *Machine Learning* conocida como *Deep Learning*, cuyo propósito es crear modelos capaces de trabajar con altos grados de abstracción, es decir, con muchas capas. De hecho, cada capa puede verse como un grado de abstracción que durante la fase de entrenamiento, el algoritmo debe de aprender.

3.8. Regularización

Cuando los perceptrones multicapa tienen más de una capa, se sabe que tienen capacidad de aproximarse a un determinado objetivo, lo que conduciría a un sobreajuste. Para prevenir dicho sobreajuste, se recomienda regularizar los datos de entrada. Sin embargo, este proceso se realiza de forma diferente en el caso de las redes de neuronas convolucionales, se puede usar:

- **Dropout**, es una técnica en la que una serie de neuronas seleccionadas al azar se ignoran durante el entrenamiento. Esto significa que su contribución a la activación de las neuronas siguientes se elimina temporalmente en la propagación hacia delante y las actualizaciones de la neurona no se aplican a la neurona en la propagación hacia atrás. El efecto es que la red se vuelve menos sensible a los pesos específicos de las neuronas dando como resultado una red que es capaz de dar una mejor generalización y que tenga un menor sobreajuste a los datos de entrenamiento.
- **Agrupación estocástica**, donde la activación es seleccionada de forma completamente aleatoria. La agrupación estocástica no requiere hiperparámetros y puede usarse como heurística, es decir, con otras técnicas de regularización.
- **DropConnect**, que es una generalización de *dropout*, donde cada conexión es capaz de eliminarse con probabilidad $1 - p$. Cada unidad en esta capa ingresa datos de unidades aleatorias en la capa anterior, que cambian cada iteración. Esto ayuda a garantizar que los pesos no se sobreajusten.
- **Decadencia de peso**, que funciona de forma similar a las regularizaciones L1 (donde el coste agregado es proporcional al valor absoluto de los coeficientes de los pesos) y L2 (donde el coste agregado es proporcional al cuadrado del valor de los coeficientes de los pesos) en las cuales penalizamos grandes vectores de peso.

De estos métodos, *dropout* es muy utilizado en las redes de neuronas convolucionales, porque se ha demostrado que es una técnica efectiva. Más allá de prevenir el sobreajuste, se ha observado que *dropout* mejora la eficiencia computacional de redes de neuronas artificiales con grandes cantidades de parámetros, debido a que esta forma de regularización hace que una red de neuronas artificiales se vuelva más pequeña durante una iteración

dada. Después de todas estas iteraciones, la actuación de las redes más pequeñas pueden ser promediadas a una predicción general que habría realizado una red completa. En segundo lugar, se observa que la capa *dropout* introduce un rendimiento aleatorio en la red que permite un ruido dentro de los datos a promediar, de modo que la ocultación de las señales en los mismos es disminuida.

No es extraña la utilización de la regularización L1, pero hay que tener en cuenta el hecho de que los vectores de peso pueden reducirse a 0, en algunas ocasiones lo suficiente como para dejar una matriz de pesos escasamente poblada. El efecto negativo de este tipo de regularización es que las entradas a ciertas capas que contienen información importante pueden pasar desapercibidas debido a una conexión “muerta” entre capas. Por el contrario, cuando se desea una selección de funciones muy explícita, la regularización L1 puede producir un rendimiento significativamente mejor.

La regularización L2 es vista tradicionalmente como el método estándar utilizado en las redes de neuronas convolucionales, porque tiende a penalizar los pesos demasiado grandes y favorece a aquellos que son pequeños en su proporción con respecto a la totalidad de la matriz. A diferencia de la regularización L1, se obtiene una matriz de pesos más grande, lo que hará que la red alimente más datos de una capa determinada a la siguiente. Como tal, la selección de características será menos dura que cuando se utiliza la regularización L1.

Otro tipo de regularización consiste en una modificación de los esquemas de regularización imponiendo límites en el tamaño de un peso determinado. Esto permitiría que las actualizaciones de los parámetros tengan un límite estricto, limitándose el número de posibles soluciones que puede proporcionar una red determinada. Esto ayudaría a entrenar la red de neuronas artificial de una forma más rápida y en la solución óptima evitar que los parámetros se actualicen demasiado en una dirección incorrecta.

Capítulo 4

Aplicaciones de CNNs en R

R es un lenguaje de programación que tiene un enfoque al análisis estadístico. Es uno de los lenguajes más utilizados en los campos del aprendizaje automático, que es el campo en el que se van a realizar las aplicaciones de este trabajo.

Además, R es capaz de cargar diferentes librerías con distintas funcionalidades, como la que aquí se va a utilizar, la librería Keras.

4.1. Librería Keras

Una de las librerías que mejor implementa las redes de neuronas artificiales es la librería **Keras** del paquete estadístico R.

Según la misma página web https://keras.rstudio.com/ de la librería, Keras es una API (un conjunto de definiciones y protocolos que se utiliza para desarrollar e integrar el software de las aplicaciones) de redes de neuronas artificiales de alto nivel desarrollada con un enfoque en permitir una rápida experimentación, "poder pasar de la idea al resultado de una forma rápida es clave para realizar una buena investigación". Keras posee las siguientes características:

- Permite que el mismo código se ejecute en la CPU o en la GPU.
- Es una API con un sencillo uso que facilita la creación rápida de prototipos de modelos de aprendizaje profundo.
- Tiene un soporte integrado para las redes convolucionales, las redes recurrentes y cualquier combinación de ambas.
- Admite arquitecturas de red arbitrarias.

4.1.1. Funciones

Dentro de esta librería encontramos muchas funciones para poder implementar en casos prácticos las redes de neuronas artificiales, y más específicamente las redes de neuronas convolucionales. En esta sección, vamos a recoger las funciones que se van a utilizar así como para lo que sirven.

4.1.1.1. layer_conv_1d()

Esta función es utilizada para crear una capa convolucional unidimensional. Sus principales argumentos son:

- **object**: Modelo u objeto de capa.
- **filters**: Valor entero, representa la dimensión del espacio de salida, es decir, el número de filtros de salida de la convolución.
- **kernel_size**: Es un entero o una lista simple de enteros, especificando la longitud de la ventana de convolución unidimensional.

Por defecto se utiliza una activación lineal y el paso es igual a 1. Si queremos cambiar alguno de estos valores habría que modificar los argumentos *activation* y *strides* respectivamente.

4.1.1.2. layer_conv_2d()

Esta función es utilizada para crear una capa convolucional bidimensional. Sus argumentos principales son:

- **object**: Modelo u objeto de capa.
- **filters**: Valor entero, es la dimensionalidad del espacio de salida, es decir, el número de filtros de salida en la convolución.
- **kernel_size**: Número entero o una lista de 2 números enteros, especificando el ancho y el alto de la ventana de convolución $2D$. Puede ser un único entero para especificar el mismo valor para todas las dimensiones espaciales.
- **strides**: Número entero o una lista de 2 números enteros, especificando los pasos de la convolución a lo largo de la anchura y de la altura. Puede ser un único número entero para especificar el mismo valor para todas las dimensiones espaciales.
- **padding**: Sus valores pueden ser *"valid"* o *"same"*. Se tiene que *"same"* es ligeramente inconsistente cuando se tiene un valor de paso igual a 1.
- **input_shape**: Valor entero de las dimensiones de la entrada sin incluir el eje de las muestras. Este argumento es necesario cuando se utiliza esta capa como la primera capa de un modelo.

4.1.1.3. layer_embedding()

Convierte enteros positivos en vectores densos de tamaño fijo. Esta capa únicamente puede ser utilizada al principio de la red. Sus principales argumentos son:

- **object**: Modelo u objeto de capa.
- **input_dim**: Valor entero mayor que cero. Es el índice entero máximo más 1.
- **output_dim**: Valor entero mayor o igual que cero. Es la dimension de la capa.

4.1.1.4. layer_activation()

Con esta función se indica la función de activación que se va a aplicar a la salida. Como principales argumentos tenemos:

- **object**: Modelo u objeto de capa.
- **activation**: Nombre de la función de activación a usar. Si no se especifica nada, se aplica la activación "lineal" $a(x) = x$.
- **input_shape**: Forma de entrada necesaria cuando se utiliza esta capa como la primera capa de un modelo.

4.1.1.5. layer_dropout()

Función para establecer aleatoriamente una tasa de unidades de entrada a cero en cada actualización durante el tiempo de entrenamiento, para evitar el sobreajuste. Principales argumentos:

- **object**: Modelo u objeto de capa.
- **rate**: Su valor está entre 0 y 1. Es la fracción de las unidades de entrada a soltar.

4.1.1.6. layer_golbal_max_pooling_1d()

Operación global de *max-pooling* para datos temporales. Su principal argumento es:

- **object**: Modelo u objeto de capa.

Los demás valores se suelen dejar por defecto.

4.1.1.7. layer_max_pooling_2d()

Operación de *max-pooling* para datos espaciales. Argumentos:

- **object**: Modelo u objeto de capa.
- **pool_size**: Entero o lista de 2 enteros, factores por los que se debe de reducir la escala (vertical, horizontal). Si se toma el valor $(2, 2)$ reducirá a la mitad la entrada en ambas dimensiones espaciales. Si únicamente se especifica un entero, se utilizará la misma longitud de ventana para ambas dimensiones.

4.1.1.8. layer_dense()

Esta función implementa la operación: salida = activación(dot(input,kernel) + sesgo) donde la activación es la función de activación especificada en el elemento anterior como argumento de activación, kernel es una matriz de ponderaciones creada por la capa, y el sesgo es un vector con los sesgos creado por la capa (únicamente aplicable si use_bias tiene el valor TRUE). Principales argumentos:

- **object**: Modelo u objeto de capa.
- **units**: Valor entero positivo. Es la dimensión del espacio de salida.
- **use_bias**: Si la capa utiliza o no un vector de sesgo. Por defecto contiene el valor TRUE.

4.1.1.9. layer_flatten()

Sirve para acoplar una entrada dada, sin afectar al tamaño del lote.

4.2. Ejemplo práctico I

En este ejemplo se tratará de aplicar las redes de neuronas convolucionales para realizar una clasificación. Se utilizará el conjunto de datos `dataset_imbd`, que contiene 25.000 críticas de películas desde IMBD etiquetadas según despierten un sentimiento positivo o negativo. Las críticas han sido preprocesadas y cada crítica está codificada como una secuencia de índices de palabras (enteros). Por conveniencia, las palabras se clasifican según la frecuencia general en el conjunto de datos, por ejemplo, el entero "3" codifica la tercera palabra más frecuente en los datos.

Definimos parámetros.

En primer lugar, procedemos a cargar la librería Keras y definir los siguientes parámetros:

```
library(keras)

max_features <- 5000 # Número máximo de características
maxlen <- 400 # Longitud máxima
batch_size <- 32 # Tamaño del batch
embedding_dims <- 50 # Tamaño de embedding
filters <- 250 # Número de filtros
kernel_size <- 3 # Tamaño del núcelo
hidden_dims <- 250 #Dimensión capas ocultas
epochs <- 2 # Número de epochs
```

`batch_size` es dividir el número de ejemplos que se cargan en memoria para que no tarde demasiado tiempo en actualizar los pesos.

Si `epochs` toma el valor 2, es para que itere 2 veces sobre el conjunto de entrenamiento.

Preparación de los datos.

```
# Keras carga todos los datos en una lista con la siguiente estructura:
# List of 2
# $ train:List of 2
# ..$ x:List of 25000
# .. .. [list output truncated]
# .. ..- attr(*, "dim")= int 25000
# ..$ y: num [1:25000(1d)] 1 0 0 1 0 0 1 0 1 0 ...
# $ test :List of 2
# ..$ x:List of 25000
# .. .. [list output truncated]
```

```
# .. ..- attr(*, "dim")= int 25000
# ..$ y: num [1:25000(1d)] 1 1 1 1 1 0 0 0 1 1 ...
#
# Los datos x incluyen secuencias enteras, cada entero es una palabra.
# Los datos y incluyen un conjunto de etiquetas enteras (0 o 1)
# El argumento num_words indica que solamente las palabras max_fetures más
# frecuentes se convertirán a números enteros.

imdb <- dataset_imdb(num_words = max_features)

# Rellenar la secuencia para que tengan la misma longitud
# Esto convierte al conjunto de datos en una matriz, cada línea es una
# crítica y cada columna una palabra en la secuencia
# Rellena las secuencias con 0 a la izquierda
x_train <- imdb$train$x %>%
  pad_sequences(maxlen = maxlen)
x_test <- imdb$test$x %>%
  pad_sequences(maxlen = maxlen)
```

Definición del modelo.

```
#Inicialización del modelo.
model <- keras_model_sequential()

model %>%
  # Se comienza con una capa de incrustación eficiente que busca los
  # índices de vocabulario en las dimensiones embedded_dims
  layer_embedding(max_features, embedding_dims, input_length = maxlen) %>%
  layer_dropout(0.2) %>%

  # Adición de una convolución unidimensional, la cual aprenderá filtros.
  # Filtros de grupo de palabras de tamaño filter_length
  layer_conv_1d(
    filters, kernel_size,
    padding = "valid", activation = "relu", strides = 1
  ) %>%
  # Aplicación de max-pooling
  layer_global_max_pooling_1d() %>%

  # Adición de una capa oculta
  layer_dense(hidden_dims) %>%

  # Aplicación de 20% en la capa dropout.
  layer_dropout(0.2) %>%
  layer_activation("relu") %>%

  # Proyectar en una sola unidad de la capa de salida y aplicar
```

```
  # la función sigmoide

  layer_dense(1) %>%
  layer_activation("sigmoid")

# Compilación del modelo
model %>% compile(
  loss = "binary_crossentropy",
  optimizer = "adam",
  metrics = "accuracy"
)
```

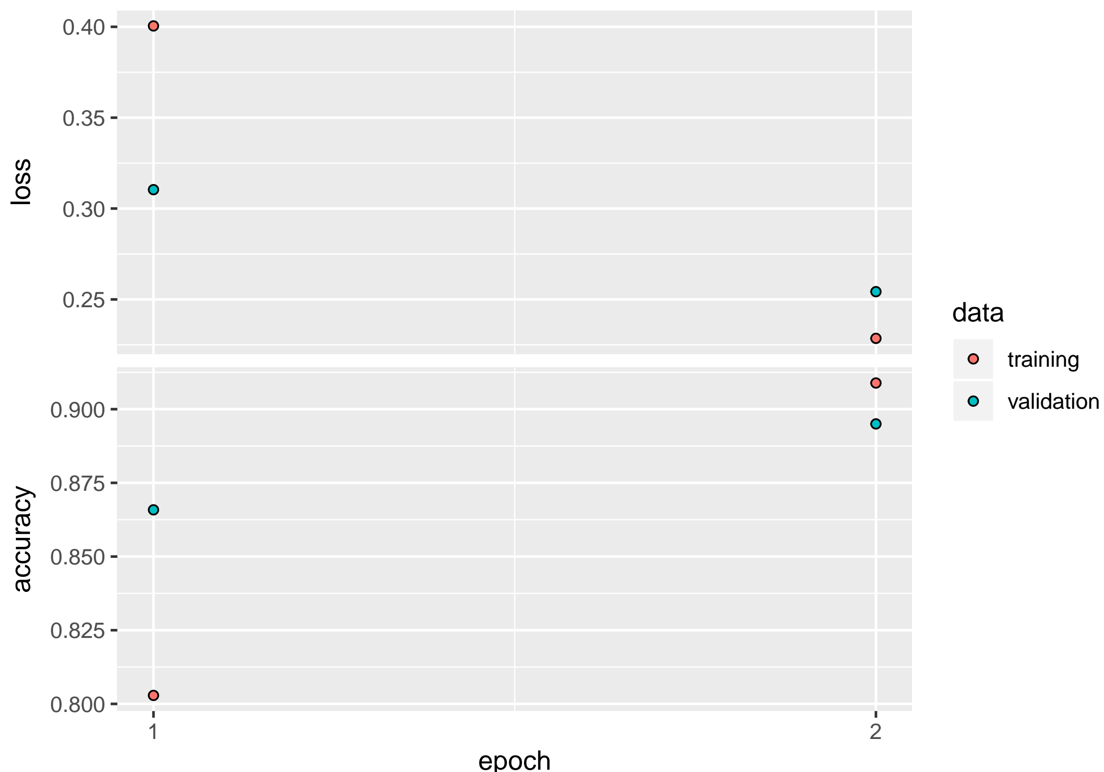
Entrenamiento.

Utilizamos la función `fit()` para entrenar el modelo durante 2 etapas usando conjuntos de 32 datos, definidos anteriormente como `epochs` y `batch_size` respectivamente.

```
history <- model %>%
  fit(
    x_train, imdb$train$y,
    batch_size = batch_size,
    epochs = epochs,
    validation_data = list(x_test, imdb$test$y)
  )
```

El objeto `history` que devuelve `fit()` incluye las medidas de pérdida y de precisión, que además, se pueden representar gráficamente:

```
plot(history)
```



4.3. Ejemplo práctico II

El conjunto de datos MNIST contiene pequeñas imágenes en escala de grises de 28×28 píxeles de dígitos escritos a mano que han sido clasificados por humanos.

Dicho conjunto de datos se encuentra disponible dentro de la propia librería Keras para su uso.

Importación de los datos

```
mnist <- dataset_mnist()
```

Preparación de los datos

El conjunto de datos ya viene dividido en conjuntos de entrenamiento y test, cada uno de ellos con un conjunto de variables características (las imágenes) y una variable objetivo.

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

Las dimensiones del conjunto de características (las imágenes) de entrenamiento se dan a continuación.

```
dim(x_train)
```

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

Es decir, se tienen 60000 imágenes en escala de grises, cada una de ellas con un tamaño de 28×28 píxeles. Vamos a asignar estos valores a continuación.

```
img_filas <- 28
img_columnas <- 28
```

Estas imágenes no tienen la forma correcta como tensores, ya que falta el número de canales. Esto se puede corregir tanto para los conjuntos de entrenamiento como para los conjuntos test mediante el uso de la función `array_reshape()`. También se va a crear la variable `input_shape` que contenga las correctas dimensiones de las imágenes.

```
x_train <- array_reshape(x_train,
                         c(nrow(x_train),
                           img_filas,
                           img_columnas, 1))
x_test <- array_reshape(x_test,
                        c(nrow(x_test),
                          img_filas,
                          img_columnas, 1))
input_shape <- c(img_filas,
                 img_columnas, 1)
dim(x_train)
```

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

Como en todas las redes de neuronas artificiales, los datos deben de normalizarse. Dado que los valores de los píxeles representan la luminosidad en una escala de 0 (negro) hasta 255 (blanco), todos se pueden normalizar dividiendo cada uno de ellos por el valor máximo, es decir, dividir entre 255.

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

El espacio muestral de la variable objetivo contiene 10 elementos, es decir, hay 10 clases en la variable objetivo. Estas pueden ser codificadas usando la función `to_categorical()`.

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

Por ejemplo, la imagen 355 del conjunto de entrenamiento se corresponde al número 4. Nótese que se empieza a contar desde el cero.

```
y_train[355,]
```

```
##  [1] 0 0 0 0 1 0 0 0 0 0
```

El modelo

Se construye la red convolucional cuya primera capa es una capa convolucional de 16 filtros cada uno de ellos de tamaño 3×3 utilizando la función de activación `ReLu` y las dimensiones definidas anteriormente como `input_shape`.

A esta capa de convolución, le sigue una capa de agrupación máxima, cuyo tamaño de cuadrícula es 2×2 que reducirá a la mitad la dimensión de la primera capa. Luego se tiene una capa de abandono en la cual internamente irá dejando el 25 % de las neuronas sin utilizar en el proceso de entrenamiento, lo que nos permite tener una red más robusta para el sobreajuste.

Con `layer_flatten()` lo que se hace es obtener una capa plana con neuronas todas al mismo nivel, en una dimensión en lugar de en tres dimensiones distintas ya que las capas totalmente conectadas a diferencia de las capas convolucionales necesitan tener una capa plana donde no hay dimensiones. De esta forma, se crea una capa totalmente conectada de 10 neuronas. En este caso de nuevo utilizamos dropout=0.5 para evitar de nuevo la posibilidad de sobreajuste y por último, una capa totalmente conectada con el número de clases indicadas, que serían 10, y con activación `softmax` para obtener una normalización en la última capa.

Creando el modelo

```
model <- keras_model_sequential() %>%
  layer_conv_2d(filters = 16,
                kernel_size = c(3,3),
                activation = 'relu',
                input_shape = input_shape) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_dropout(rate = 0.25) %>%
  layer_flatten() %>%
  layer_dense(units = 10,
              activation = 'relu') %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = num_classes,
              activation = 'softmax')
```

Un resumen del modelo muestra que hay 27320 parámetros en dicho modelo, todos ellos entrenables.

La primera capa se reduce a 26×26, ya no es 28×28 pues los filtros son 3×3 perdiéndose una fila y una columna. Además tenemos 160 parámetros, ya que al tener 16 filtros 3×3, se tendría $16 \cdot (9 + 1) = 160$. Ese 1 hace referencia a los sesgos.

En la capa de agrupación máxima no hay ningún peso que aprender, al igual que en la capa de abandono y en `flatten`, por eso en dichas capas se tienen 0 parámetros. En `flatten` se produce el cambio de dimensión con respecto a la capa anterior, obteniéndose $13 \cdot 13 \cdot 16 = 2704$.

Y por último las capas totalmente conectadas, en la primera de las cuales hay $2704 \cdot 10 + 10 = 27050$, donde el 10 que aparece sumado son los sesgos. De nuevo hay otra capa de abandono con 0 parámetros y por último una capa totalmente conectada con $10 \cdot 10 + 10 = 110$ parámetros.

```
model %>% summary()
```

```
## Model: "sequential_1"
## ___________________________________________________________________________
## Layer (type)                     Output Shape                  Param #
## ===========================================================================
## conv2d (Conv2D)                  (None, 26, 26, 16)            160
## ___________________________________________________________________________
## max_pooling2d (MaxPooling2D)     (None, 13, 13, 16)            0
## ___________________________________________________________________________
## dropout_2 (Dropout)              (None, 13, 13, 16)            0
## ___________________________________________________________________________
## flatten (Flatten)                (None, 2704)                  0
## ___________________________________________________________________________
## dense_2 (Dense)                  (None, 10)                    27050
## ___________________________________________________________________________
## dropout_3 (Dropout)              (None, 10)                    0
## ___________________________________________________________________________
## dense_3 (Dense)                  (None, 10)                    110
## ===========================================================================
## Total params: 27,320
## Trainable params: 27,320
## Non-trainable params: 0
## ___________________________________________________________________________
```

Compilando

La entropía cruzada categórica sirve como función de pérdida. `Adadelta` optimiza el descenso del gradiente y la precisión sirve como métrica.

```
model %>% compile(
  loss = loss_categorical_crossentropy,
  optimizer = optimizer_adadelta(),
  metrics = c('accuracy')
)
```

Entrenamiento

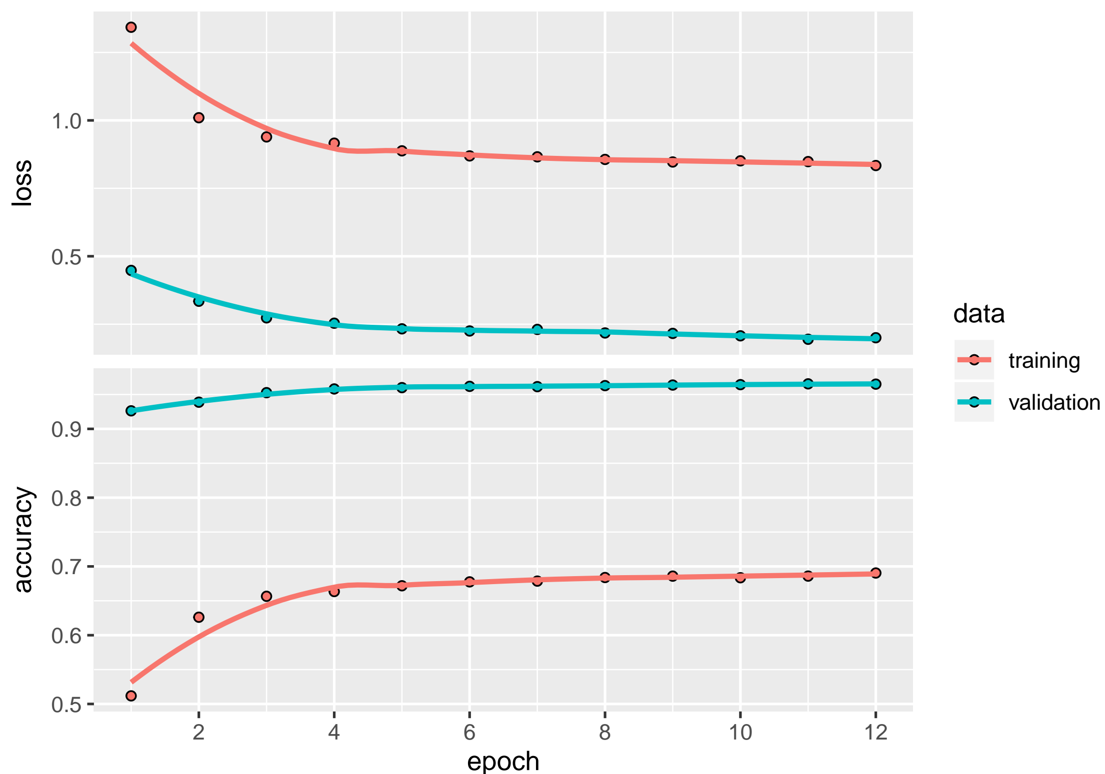
Utilizamos la función `fit()` para entrenar el modelo durante 12 etapas usando conjuntos de 128 imágenes, que permitirá a los tensores ajustarse a la memoria de unidad de procesamiento de gráficos del ordenador con el que se esté trabajando. El modelo se ejecutará con una división de validación que se establece en 0.2.

```
batch_size <- 128
epochs <- 12

# Entrenar el modelo
history <- model %>% fit(
  x_train, y_train,
  batch_size = batch_size,
  epochs = epochs,
  validation_split = 0.2
)
```

El objeto `history` que devuelve `fit()` incluye las medidas de pérdida y de precisión, que además, se pueden representar gráficamente:

```
plot(history)
```



A medida que el modelo entrena, se muestran las medidas de pérdida y precisión. Este modelo comienza con una precisión de un 50 % en el conjunto de entrenamiento, y al final de las 12 etapas termina con algo más de un 65 % de precisión. Si miramos la precisión en el conjunto de validación, es mayor al 90 % en todas las etapas acabando con una precisión mayor al 95 %.

Es decir, en este gráfico tenemos mejores resultados con el conjunto de validación que sobre el conjunto de entrenamiento, lo cual descarta el sobreajuste.

Evaluación de la precisión

El modelo puede ser evaluado usando el conjunto test.

```
score <- model %>% evaluate(x_test,
                            y_test)

cat('Test loss: ', score$loss, "\n")
```

```
## Test loss:  0.1932831
```

```
cat('Test accuracy: ', score$acc, "\n")
```

```
## Test accuracy:  0.9656
```

Se tiene una precisión del 96.5600014 % en el conjunto test, la cual es bastante alta.

Predicciones.

Cuando el modelo ha sido entrenado, puede ser utilizado para predecir sobre un nuevo conjunto de datos.

Se generan predicciones sobre nuevos datos:

```
predicciones=model %>% predict_classes(x_test)
```

De esta forma, el modelo ha predecido la clase para cada una de las imágenes en el conjunto de prueba. Las primeras 10 predicciones serían:

```
predicciones[1:10]
```

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

Las predicciones toman valores desde 0 hasta 9, y devuelven aquel valor que el modelo considera más probable. Esto es, la primera predicción devuelve 7, por lo que el modelo considera que esa primera imagen hace referencia al número 7, el cual considera el más probable entre las 10 posibles clases de salida. La octava predicción devuelve el valor 9, es decir, el modelo considera que la octava imagen del conjunto se corresponde con un 9.

Bibliografía

[1] Allaire, J., Xie, Y., McPherson, J., Luraschi, J., Ushey, K., Atkins, A., Wickham, H., Cheng, J., Chang, W. and Iannone, R. 2019. *Rmarkdown: Dynamic documents for r.*

[2] Anderson, D. and McNeill, G. 1992. Artificial neural networks technology. *Kaman Sciences Corporation.* 258, 6 (1992), 1–83.

[3] Dahl, D.B., Scott, D., Roosen, C., Magnusson, A. and Swinton, J. 2019. *Xtable: Export tables to latex or html.*

[4] DI STEFANO, L., TONIONI, A. and GALEONE, P. Rete neurale convoluzionale per classificazione di immagini e localizzazione di oggetti.

[5] II, T. Beysolow 2017. *Introduction to Deep Learning Using R.* Apress.

[6] J. J.Allaire, F.C. with *Deep Learning with R.* Manning.

[7] Luque-Calvo, P.L. 2017. *Escribir un Trabajo Fin de Estudios con R Markdown.* Disponible en http://destio.us.es/calvo.

[8] Maiani, F. 2016. Applicazioni e limiti della classificazione di immagini con reti neurali convoluzionali in dispositivi mobili. (2016).

[9] Marsland, S. 2014. *Machine Learning: An Algorithmic Perspective.* Chapman; Hall/CRC.

[10] Matich, D.J. 2001. Redes Neuronales: Conceptos básicos y aplicaciones. *Universidad Tecnológica Nacional, México.* (2001).

[11] Nacelle, A. Redes Neuronales Artificiales. *Universidad de la República, Uruguay.*

[12] Nielsen, M.A. 2015. *Neural Networks and Deep Learning.* Dtermination Press.

[13] R Core Team 2016. *R: A Language and Environment for Statistical computing.* R Foundation for Statistical Computing.

[14] RStudio Team 2015. *RStudio: Integrated Development Environment for R.* RStudio, Inc.

[15] Wickham, H. 2019. *Stringr: Simple, consistent wrappers for common string operations.*

[16] Wickham, H., Chang, W., Henry, L., Pedersen, T.L., Takahashi, K., Wilke, C., Woo, K. and Yutani, H. 2019. *Ggplot2: Create elegant data visualisations using the grammar of graphics.*

[17] Wickham, H., François, R., Henry, L. and Müller, K. 2020. *Dplyr: A grammar of data manipulation.*

[18] Xie, Y. 2019. *Knitr: A general-purpose package for dynamic report generation in r.*