Capítulo IV

Redes Neuronales Artificiales.

Las redes neuronales artificiales, actualmente muy extendidas en todos los ámbitos de nuestra sociedad, tienen su origen en 1958 en un Laboratorio Aeronáutico de la Universidad de Cornell por Frank Rosenblatt, con la invención de su algoritmo *el perceptrón* [13], que simula la siguiente función:

$$f(x) = \begin{cases} 1 \; si \; \sum_{i=1}^{N} x_i w_i + b_i > 0 \\ 0 \qquad resto \; de \; casos \end{cases} \tag{4.1}$$

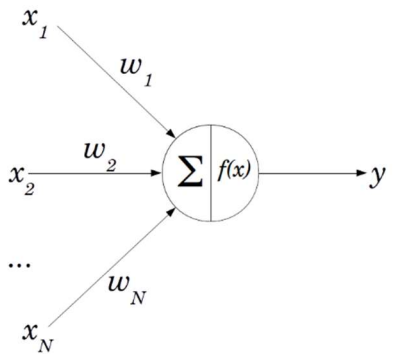

Figura 11: Concepto simple del algoritmo del perceptrón.

En la Figura 11 y en la definición (4.1), $x = (x_1, \dots, x_N)$ representa el conjunto de valores de entrada a la neurona, w_i corresponde a los pesos relacionados de cada input, b_i (bias) es el sesgo de cada valor de entrada y N el número de inputs de la neurona. Se trata de un clasificado binario que decide el grupo al que pertenece cada uno de los valores de entrada.

Este algoritmo, basado en la teoría del aprendizaje hebbiano de Donal O. Hebb [14], simula el comportamiento de las conexiones entre las neuronas cerebrales.

El perceptrón de una sola capa tiene el inconveniente de ser únicamente capaz de aprender patrones lineales. Sumado a la falta de capacidad computacional de la época, hizo que la teoría empezara a perder fuerza. Posteriormente, en 1985, fue con la

implementación del algoritmo de *backpropagation* [15] a las redes multicapa, lo que cambió el paradigma de las redes neuronales, haciendo éstas mucho más eficientes a la hora de predecir patrones. En las últimas décadas, el gran avance tecnológico, sobre todo en el campo de los semiconductores, ha convertido a las redes neuronales en un recurso fundamental en diferentes ámbitos de la sociedad.

En esta sección se va a hacer una breve explicación de los conceptos fundamentales de las redes neuronales profundas y sus parámetros, por lo que, si el lector está interesado en ampliar su conocimiento sobre la materia, en el libro *Deep Learning* [16] (Goodfellow et al.; 2016; MIT) puede encontrar información mucho más detallada.

4.1 Perceptrones multicapa, capas ocultas

Uno de los problemas principales del perceptrón, como explicamos anteriormente, es su linealidad, lo cual limita mucho las posibilidades de usarlo como herramienta matemática. Las capas ocultas han sido un método muy efectivo para solventar este problema, ya que esta sucesión de capas permite que la red simule comportamientos no lineales. Todas las neuronas de las capas están "completamente conectadas" (*fully-connected*) con las neuronas de sus capas vecinas como se muestra en la Figura 12.

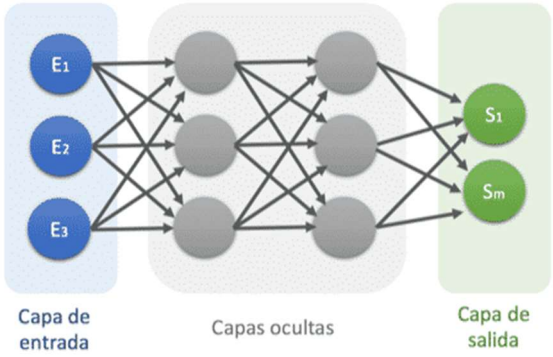

Figura 12: Concepto simple del perceptrón multicapa.

Como se puede ver en la Figura 13, este perceptrón multicapa se asemeja mucho a cómo las neuronas biológicas reciben la información mediante sus dendritas y la envían a través del axón, el cual representa las capas ocultas, hasta llegar a las terminales del axón. Estas terminales simulan los valores de salida del perceptrón multicapa.

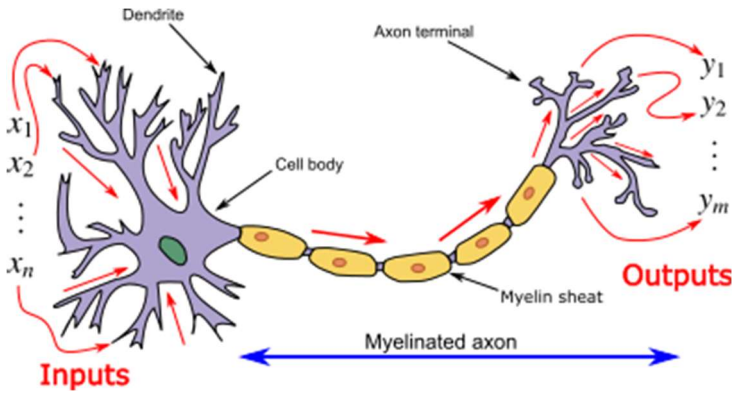

Figura 13: Símil entre neurona biológica y perceptrón multicapa.

A continuación, se describe matemáticamente la operativa necesaria en el caso de una red con una única capa oculta.

$$z_j = \sum_{j=1}^{m}\sum_{i=1}^{n} w_{j,i}x_i + b_j,$$
$$y_j = \sum_{j=1}^{m}\sum_{i=1}^{n} w_{j,i}z_i + b_j, \tag{4.2}$$

donde x_i representa los valores de entrada de la red, z_j el valor de las neuronas de la capa oculta, $w_{j,i}$ y b_j representan los "pesos" y el "sesgo" que se atribuye a cada neurona de la capa oculta, por ejemplo, $z_1 = (w_{1,1}x_1 + w_{1,2}x_2 + w_{1,3}x_3 + \cdots + w_{1,n}x_n) + b_1$. Por último, y_j corresponde a los valores resultado, situados en la capa final, que estamos buscando.

4.2 Funciones de activación

Para aprovechar el potencial de la arquitectura multicapa, necesitamos de un componente fundamental, una *función de activación* ($f(\cdot)$), la cual aplicaremos a cada neurona. Estas funciones pueden decidir si una neurona debe activarse o no, y, en caso de activarse, el valor ponderado que ésta debe tener para nuestra red neuronal. Las funciones de activación se distinguen en lineales y no lineales.

ReLU (Rectified Linear Unit)

La activación *ReLU* es la más popular a la hora de construir redes neuronales, tanto por su simplicidad como por su buen desempeño en multitud de tareas [17]. Dado un elemento *x*, la función se define como el máximo entre éste y 0.

$$ReLU(x) = max(x, 0). \tag{4.3}$$

ELU (Exponencial Linear Unit)

La diferencia entre la activación *ReLU* y *ELU* es que, en esta última, obtenemos valores diferentes a cero cuando la entrada es negativa. La *ELU* es una función que tiende a minimizar el error de forma más rápida y a producir resultados más precisos [18]. En cambio, tiene el problema de ser computacionalmente más pesada que la *ReLU*.

$$ELU(x) = \begin{cases} x & si \;\; x > 0 \\ \alpha(e^x - 1) & si \;\; x < 0. \end{cases} \tag{4.4}$$

Sigmoide / Logística

La función sigmoide transforma los datos de entrada en un intervalo de valores entre 0 y 1.

$$sigmoid(x) = \frac{1}{1 + \exp(-x)}. \tag{4.5}$$

Se trata de una de las primeras activaciones utilizadas en redes neuronales, por ser una función diferenciable y por su similitud con el "potencial de acción" de las neuronas biológicas [19]. Pero, debido a que tiene una derivada cercana a cero en los extremos provocando problemas de desvanecimiento de gradiente, con el tiempo se ha visto reemplazada por la función *ReLU*. Ésta, además de evitar el problema anterior, es más fácil de entrenar por su simplicidad. Aun así, la función sigmoide es muy útil a la hora de construir redes neuronales, colocándose en la última capa para servir como función resultado del modelo.

TanH / Tangente-hiperbólica

Al igual que la función sigmoide, la función tangente-hiperbólica transforma los datos de entrada a un resultado acotado. En este caso, el intervalo es (-1,1).

$$tanh(x) = \frac{1 - \exp(-2x)}{1 + \exp(-2x)}. \tag{4.6}$$

La tangente-hiperbólica nos permite trabajar con valores de entrada negativos, aunque sigue teniendo el problema del desvanecimiento de gradiente cuando nos alejamos del cero, puesto que lejos del eje central la derivada tiende a cero.

Existen multitud de funciones de activación, cada una de ellas con sus respectivas ventajas e inconvenientes. En la Figura 14 se muestran algunas de las funciones de activación anteriormente mencionadas y otras que también están disponibles en Tensorfllow2.1 (véase [20] para mayor detalle).

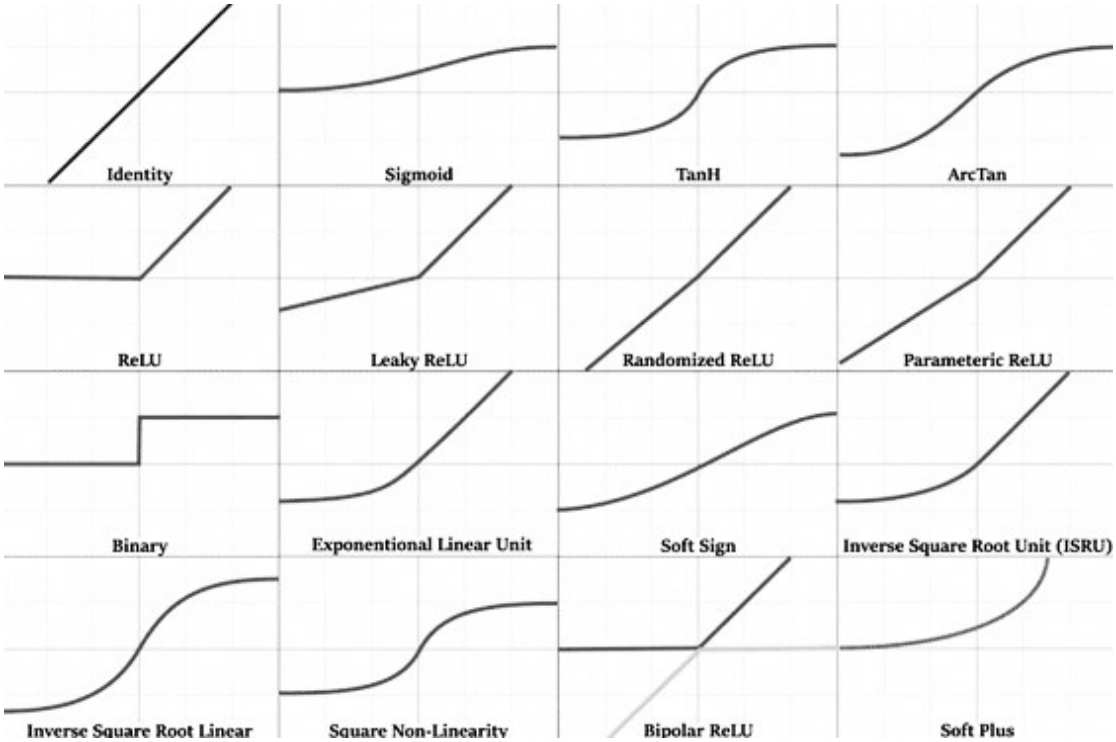

Figura 14: Algunas de las funciones de activación más usadas en el campo del Deep Learning.

El valor obtenido después de pasar por cualquier función de activación f, normalmente recibe el nombre de "activación" (a), por lo tanto, la formulación de las interconexiones neuronales de la red queda de la siguiente manera:

$$z_j^l = \sum_{j=1}^{m}\sum_{i=1}^{n} w_{j,i}{}^l a_i^{l-1} + b_j^l$$
$$a_j^l = f\left(z_j^l\right) \tag{4.7}$$

siendo l el superíndice que indica la capa correspondiente a cada parámetro.

4.3 Función de coste

La función de coste, también conocida como función de error, es la herramienta que se usará para determinar la distancia entre el valor real y el predicho. La función de coste más usada en problemas de regresión es el error cuadrático medio:

$$c^{(i)}(w,b) = \frac{1}{2}\ \left(\tilde{y}^{(i)} - y^{(i)}\right)^2 \tag{4.8}$$

donde $\tilde{y}^{(i)}$ es el valor predicho e $y^{(i)}$ el valor real.

Para medir el error cometido para todo el conjunto de datos, debemos realizar el promedio de los errores del entrenamiento:

$$C(w,b) = \frac{1}{n}\sum_{i=1}^{n} c^{(i)}(w,b) = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{2}\left(\left(w^T x^{(i)} + b\right) - y^{(i)}\right)^2. \tag{4.9}$$

En el campo del Deep Learning, cuando se trata de métricas de regresión, también se trabaja frecuentemente con otras funciones de coste como el error absoluto medio (MAE), el error logarítmico cuadrático medio (MSLE) o el error porcentual absoluto medio (MAPE). En la página web [21], se pueden encontrar todas las funciones de coste que están disponibles en la librería de Tensorflow.

El objetivo final a la hora de entrenar una red neuronal es minimizar el error de nuestra predicción respecto a la solución real. Al tener muchas capas con muchas neuronas, encontrar una solución analítica al problema es prácticamente inviable. Esto conlleva que, en las redes neuronales profundas, se utilicen otros métodos para acercarnos a la solución, que se muestran a continuación.

4.4 Optimización, descenso del gradiente

El optimizador más usado en el campo del aprendizaje profundo es, sin lugar a duda, el descenso del gradiente. Éste consiste en un algoritmo de optimización iterativo de primer orden [22], que busca encontrar un mínimo local en una función diferenciable. La idea básica del algoritmo consiste en dar pasos en dirección opuesta al gradiente de la función en un punto, de manera iterativa, hasta encontrar un mínimo. Por lo que el punto siguiente a $x^{(i)}$, se define como:

$$x^{(i+1)} = x^{(i)} - \alpha \cdot \nabla f(x^{(i)}) \tag{4.10}$$

siendo $\alpha > 0$ el tamaño del paso, en la dirección opuesta al gradiente. En la Figura 15 se muestra gráficamente una sucesión de puntos obtenida al aplicar este método.

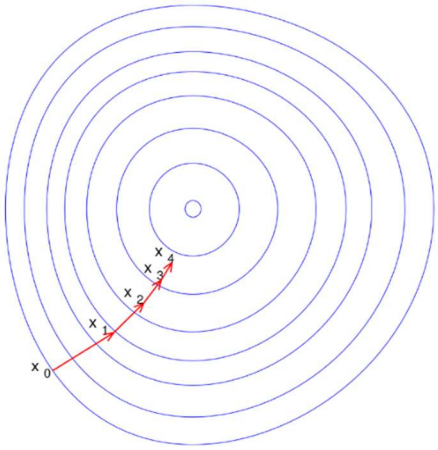

Figura 15: Imagen conceptual del descenso del gradiente. Fuente: pngegg.com

En el campo del Deep Learning, la función objetivo consiste en una suma de todos los errores individuales de cada neurona ponderados por sus respectivos pesos y como estos pesos se inicializan de manera aleatoria, el método se le denomina descenso de gradiente estocástico (SGD) [23]. La desventaja de éste se encuentra en los *outliers* de la muestra, que afectan de manera muy notable al método. Además, a la hora de resolver problemas, dependemos mucho del tamaño del paso, pudiendo estancarnos en un mínimo local no deseado.

Descenso de Gradiente con Momento (Inercia)

La diferencia de este método con el anterior [24], es que la velocidad del descenso se va acumulando a medida que avanzan los pasos, es decir, el momento le proporciona un descenso acelerado del gradiente que se define del siguiente modo:

$$p^{(i)} = \nabla f(x^{(i)}) + \eta p^{(i-1)} \tag{4.11}$$

$$x^{(i+1)} = x^{(i)} - \alpha p^{(i)}$$

siendo $p^{(i)}$ la dirección del descenso en cada iteración. El parámetro $\eta > 0$ actúa como un "coeficiente de rozamiento", para que el modelo tenga una velocidad límite y se pueda controlar la suavidad del descenso.

Aunque este método mejora el tiempo de convergencia del SGD y arregla el problema de los mínimos locales no deseados, también tiene el inconveniente de sobrepasar los resultados que se buscan (*overshooting*).

Descenso acelerado de Nesterov (NAG)

El descenso acelerado de Nesterov [25] consiste en un método de dos pasos que trata de mejorar el método anterior. En el primer paso, éste extrapola linealmente la trayectoria actual; y en el segundo paso, calculando el gradiente en el punto predicho, realiza una corrección de la trayectoria. De esta manera, se consigue una aproximación de segundo orden de la trayectoria con el coste computacional del método anterior, incrementando así la velocidad de convergencia. Sus ecuaciones son las siguientes:

$$\begin{aligned} \tilde{x}^{(i)} &= x^{(i)} - \alpha p^{(i-1)} \\ p^{(i)} &= \nabla f(\tilde{x}^{(i)}) + \eta p^{(i-1)} \\ x^{(i+1)} &= x^{(i)} - \alpha p^{(i)} \end{aligned} \tag{4.12}$$

Debido a la gran cantidad de datos de entrenamiento en este trabajo, se ha tenido que optar por el último método. Además, en las pruebas iniciales, se ha visto que también es el método más certero.

4.5 Learning Rate (Ratio de aprendizaje)

Como se ha mencionado anteriormente, el tamaño de paso, α, es uno de los parámetros fundamentales en el proceso del descenso de gradiente. En el campo del aprendizaje profundo se le conoce como ratio de aprendizaje, que define la velocidad a la que la red neuronal va a aprender.

Uno de los problemas que se encuentra a la hora de configurar este parámetro, es que, si se quiere ser preciso, la red necesita de una ratio de aprendizaje pequeña, haciendo que el aprendizaje sea muy lento y tenga la posibilidad de acabar en un mínimo local no deseado (*underfiting*). En cambio, si se quiere tener un aprendizaje rápido, se necesita de una ratio de aprendizaje grande, lo que puede producir una perdida sustancial de precisión a causa del *overfiting.*

Por ello, en este TFG, a la hora de buscar cómo definir correctamente este parámetro, se ha optado por aplicar una ratio que descienda a lo largo del proceso, de manera que se consigan las ventajas de ambos modelos. Para ello, se ha decidido aplicar una *ratio de aprendizaje exponencial decreciente*, que viene dada por la siguiente fórmula:

$$Lr = Lr_{inicial} * e^{-k*epoch} \tag{4.13}$$

siendo:

- Lr: ratio de aprendizaje
- $Lr_{inicial}$: ratio de aprendizaje inicial
- k: parámetro de velocidad de descenso
- $epoch$: iteración

Hay que tener en cuenta, que cuando se utilizan este tipo de ratios de aprendizaje, pueden surgir dos tipos de problemas:

1. Si k es muy alta, la ratio de aprendizaje descenderá muy rápido y es posible que ocurra *underfiting*, es decir, que el descenso acabe antes de llegar a la zona del mínimo deseado.
2. Si k es muy baja, es posible que estemos mucho tiempo trabajando a ratios de aprendizajes demasiado altos y no consigamos la precisión que se busca.

Por lo tanto, este tipo de *Learning Rate* tiene una dificultad añadida, donde el usuario debe probar y estimar de manera oportuna, cuál es la ratio de aprendizaje que más le conviene en cada situación, y a qué velocidad quiere que ésta descienda en el problema.

En este TFG se ha probado con muchas combinaciones de diferentes $Lr_{inicial}$ y k para cada problema, hasta encontrar el punto óptimo donde los modelos convergen al mínimo global de manera precisa y en el mínimo tiempo posible.

4.6 Backpropagation

Anteriormente, uno de los problemas fundamentales en el campo del aprendizaje profundo consistía en la dificultad de minimizar la función cuando la red neuronal se hacía muy grande. Esto cambió completamente desde la publicación del artículo del descenso de gradiente mediante *backpropagation* [15]. En este apartado, se expone el método matemático utilizado en dicho método.

Para calcular el error de cada neurona en una capa se utiliza la siguiente formulación:

$$\begin{aligned} \delta_j^L &= \frac{\partial C}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial z_j^L} \\ \frac{\partial C}{\partial b_j^L} &= \frac{\partial C}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial z_j^L} \cdot \frac{\partial z_j^L}{\partial b_j^L} = \delta_j^L \cdot 1 \\ \frac{\partial C}{\partial w_j^L} &= \frac{\partial C}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial z_j^L} \cdot \frac{\partial z_j^L}{\partial w_j^L} = \delta_j^L \cdot a_j^{L-1}, \end{aligned} \tag{4.14}$$

donde C corresponde al coste final de la red, z_j^L es el valor asociado de cada neurona, a_j^L corresponde a la activación, w_j^L son los pesos correspondientes a cada neurona, b_j^L es el sesgo asociado y δ_j^L es el error que se atribuye a cada neurona en el proceso de *backpropagation*. El superíndice *L* corresponde a la capa en la que se encuentra cada término.

Como se puede apreciar en la formulación anterior, utilizando la regla de la cadena, se puede conocer cómo varía el coste respecto a los diferentes parámetros de cada neurona, y de esta manera, podemos variar los pesos para minimizar los errores δ_j^L correspondientes. Cuando necesitamos realizar esto para muchas capas, en el artículo [15], propone lo siguiente:

$$\delta_j^{L-1} = \frac{\partial C}{\partial z^{L-1}}. \tag{4.15}$$

La anterior ecuación facilita todo el proceso, porque se puede calcular el error de una neurona como la suma ponderada de todos los errores de las neuronas de la capa

posterior, ponderándolas con los pesos y funciones de activación. De esta manera, si se realiza este cálculo con todas las neuronas de esa capa, se puede saber el error de todas las neuronas de la capa anterior. Repitiendo este proceso hasta la capa inicial, podemos cuantificar cómo varía el error de todas las neuronas de la red variando sus pesos a consecuencia (*backpropagation*).

4.7 Batch y epoch (lote y época)

Tras calcular el error asociado a cada neurona de nuestra red, se podría realizar una actualización de los pesos de éstas, aunque en la práctica esto no es lo que se suele hacer. Un problema típico a la hora de entrenar redes neuronales es la variedad de los datos de entrada, esto puede ocasionar que, si se actualiza los pesos de las neuronas con cada muestra del conjunto de datos, el entrenamiento resulte muy errático. Una estrategia para evitar este problema es estimar, sin cambiar los pesos, el error de la red con varias muestras de datos. De esta manera, el error promedio que se consigue hace que el entrenamiento sea menos errático y disminuye sustancialmente el tiempo necesario para entrenar el conjunto de datos. A este conjunto de muestras se le denomina lote (batch).

Entrenar a la red con todos los datos una sola vez puede no ser suficiente. El número de veces que la red entrena a todo el conjunto de datos se denomina época (epoch).

Hay que tener en cuenta que, cuando se entrena con un tamaño de lote muy grande, el error que se consigue para calibrar la red es demasiado general y se pierden características importantes de los datos. Además, esto dispara el número de épocas necesarias para llegar al mínimo. Por ello, se suele utilizar un tamaño de lote comedido, evitando así el entrenamiento errático (sin lote) a la par que un entrenamiento generalista.

4.8 Batch Normalization

Una de las herramientas más populares utilizadas en los últimos proyectos de aprendizaje profundo es la normalización por lotes de entrenamiento [26]. En términos generales, esta técnica tiene como objetivo suavizar las distribuciones de activación, controlando la media y la desviación estándar de los outputs de cada capa en cada lote. Para una activación y_j de la capa y se define la normalización por lotes (BN de sus siglas en inglés Batch Norm) mediante la expresión:

$$BN\left(y_j^{(b)}\right) = \gamma \cdot \left(\frac{y_j^{(b)} - \mu(y_j)}{\sigma(y_j)}\right) + \beta, \tag{4.16}$$

donde $y_j^{(b)}$ corresponde al valor del output de y_j del número de lote b. $\gamma > 0$ y β son los parámetros encargados de controlar la media y la varianza del valor resultante.

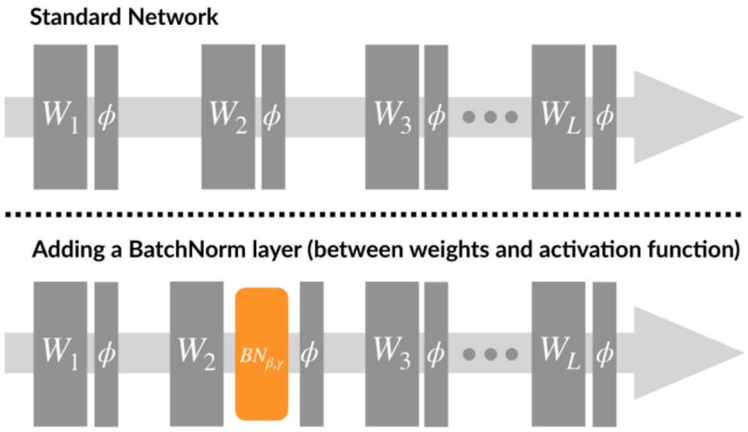

Figura 16: Estructura clásica de una arquitectura de red neuronal con Batch Normalization.

En la Figura 16, se puede apreciar que la normalización por lotes se aplica después de la primera fórmula de la ecuación (4.2) y antes de la ecuación (4.7). Es decir, después de aplicar los pesos y antes de calcular la activación.

En el artículo original, [26], los autores indican que la mejora de esta técnica viene dada por la "reducción del cambio interno de covariables", puesto que, según su hipótesis, la inicialización de parámetros y los cambios en la distribución de las entradas de cada capa afectan a la tasa de aprendizaje de la red neuronal. Sin embargo, algunos académicos argumentan que es debido a la suavización de la *función objetivo,* mejorando así el rendimiento. También, hay quienes justifican que es debido a la *explosión de gradiente* que genera el método, o que es debido a que la normalización por lotes logra un *desacoplamiento de la dirección de entrenamiento,* entre otros [27].

La normalización por lotes de entrenamiento es un método que funciona muy bien para el entrenamiento de redes neuronales, aunque no está claro el porqué de su efectividad, como ocurre con gran parte del campo del aprendizaje profundo.

4.9 Tipos de redes neuronales profundas.

En el campo del aprendizaje profundo existen numerosas arquitecturas de redes neuronales, desde la simple red fully-connected (FC), hasta otras mucho más complejas como las que se mencionan a continuación:

- Red Generativa Adversaria (GAN): consiste en una arquitectura donde una red genera datos, mientras otra discrimina y evalúa los datos creados.
- Red Neuronal Recurrente (RNN): es una red que tiene en cuenta la secuencia de los datos de entrada, por lo que es muy utilizada para el procesamiento de lenguaje natural (NLP)
- **Red Neuronal Convolucional (CNN):** utilizada comúnmente con imágenes, videos e incluso reconocimiento de voces. Consiste en la aplicación de filtros a los datos iniciales, que se encargan de extraer información adicional de los datos de entrada. Este tipo de red será el que utilizaremos en este TFG, por lo que hemos dedicado el próximo capítulo 5 a una descripción más detallada de las mismas.

- Híbridas: cuando el problema a resolver es muy complejo, se utilizan redes formadas por mezclas de otras, como ocurre en el caso de la conducción autónoma.

En la siguiente página, la Figura 17 muestra cómo se estructuran algunas de las principales arquitecturas que se utilizan en el campo del aprendizaje profundo.

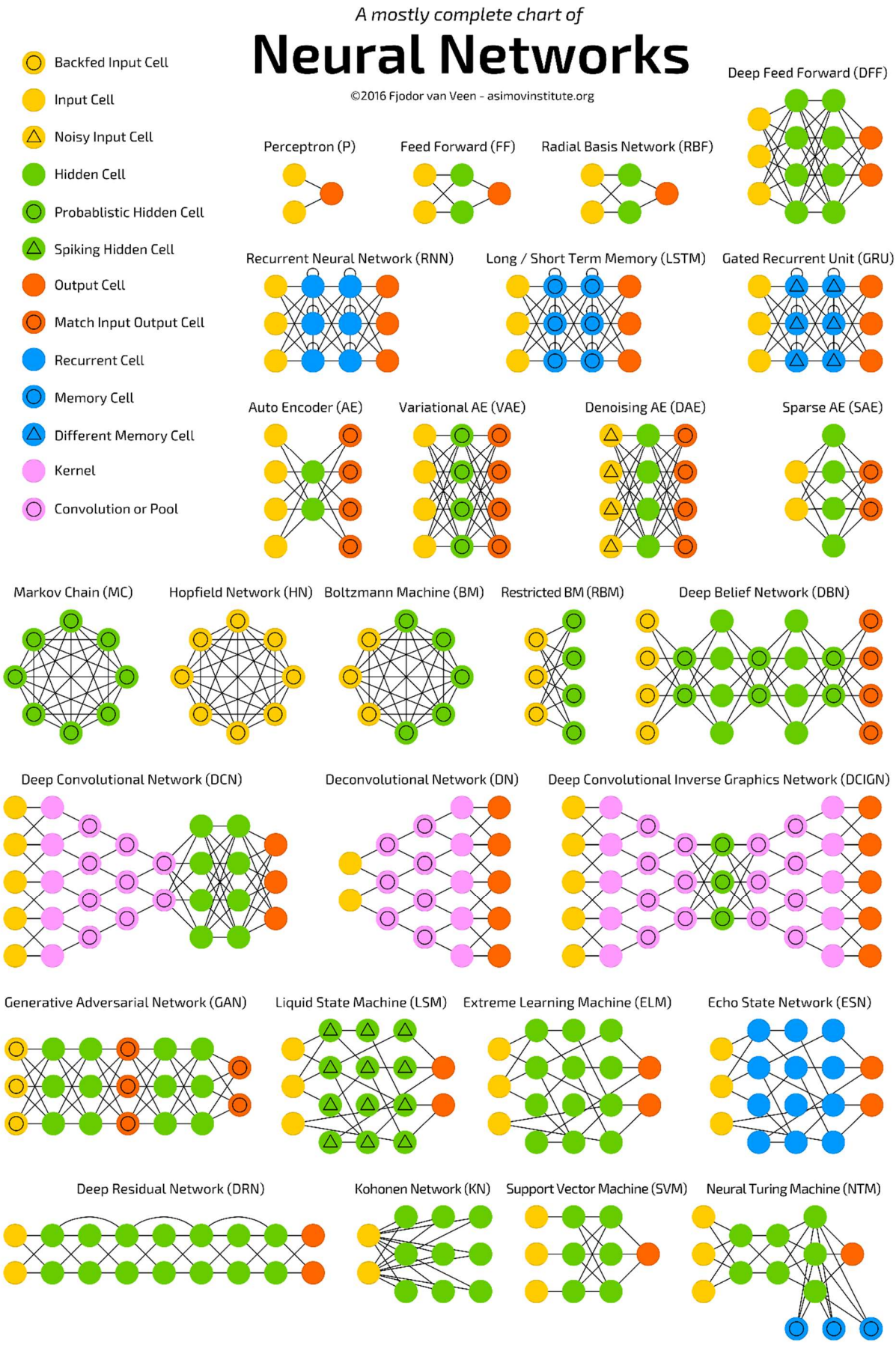

Figura 17: Arquitecturas de redes neuronales más usadas. Fuente: [28]

Capítulo V

Redes Neuronales Convolucionales (CNNs)

Como ya se ha mencionado, el objetivo del trabajo es encontrar una manera de predecir mejor la profundidad de los defectos presentes en el interior de una placa a partir de mediciones termográficas capturadas en una de sus superficies. Para ello, utilizaremos redes neuronales convolucionales. Este tipo de redes son muy eficaces a la hora de extraer información mediante patrones en los datos y en el presente trabajo se intentará sacar provecho de esta característica para mejorar la precisión de nuestras predicciones.

Las CNNs son redes neuronales jerarquizadas cuyas capas convolucionales se alternan con las capas de muestreo, recordando así a las células simples y complejas de la corteza visual primaria [29]. Por ello, el entrenamiento de las redes convolucionales difiere sustancialmente al de las redes convencionales. Algunos de los primeros trabajos que utilizan las CNNs son: Schmidhuber et al., 1996 [30]; Olshausen y Field, 1997 [31]; Hoyer y Hyvarinen, 2000 [32].

Una de las características que tienen las CNNs es la implementación de filtros localizados, que en nuestro caso servirán para detectar de manera más sensible la información de cada termografía y derivada topológica.

5.1 Capa convolucional

Según el artículo [33], dada una matriz $A_{m\times n}$ y una matriz $C_{k\times l}$ con $k < m$ y $l < n$ se define la convolución de las matrices A y C como una nueva matriz $D = A * C$ definida a partir de la expresión:

$$d_{ij} = \sum_{r=1}^{k}\sum_{s=1}^{l} a_{(i-1)+r,(j-1)+s} \cdot c_{r,s}\,, \tag{5.1}$$

donde d_{ij} está definida para $i = 1, \dots, m-k+1$ y $j = 1, \dots, n-l+1$. La matriz C se le denomina *kernel* o filtro. Para ilustrar este proceso de convolución se ha considerado el ejemplo de la Figura 18:

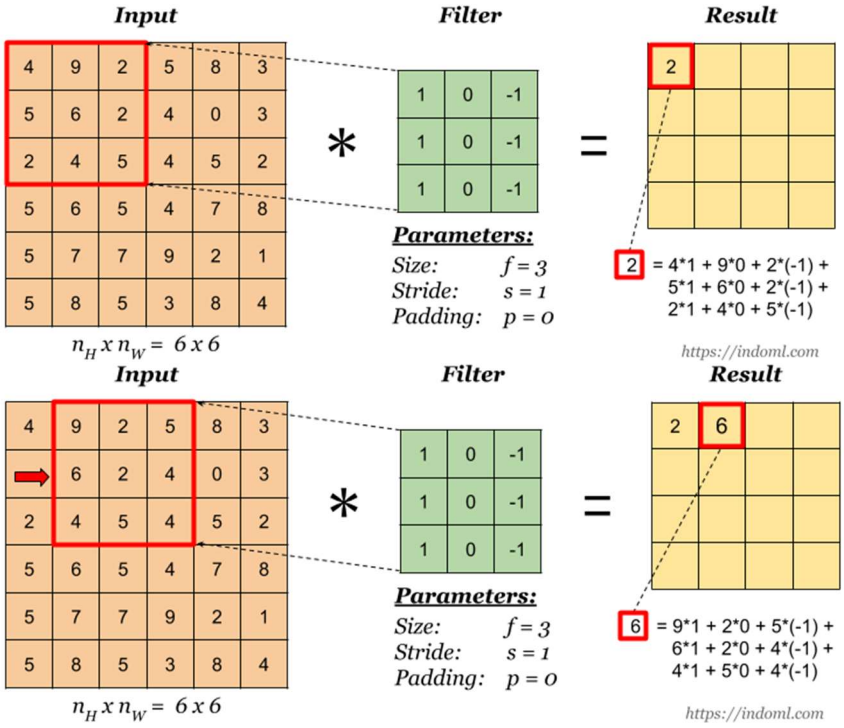

Figura 18: Operación de un filtro a la matriz de entrada. Fuente: indoml

Como se puede ver en la Figura 18, la convolución consiste en la multiplicación directa de la matriz inicial por una matriz filtro en cada zona, creando así una matriz final más pequeña que resalte ciertas características de la matriz inicial.

Nótese que según la igualdad (5.1) y la Figura 18, el tamaño de la matriz de salida es menor que el de entrada. Cuando el salto (*stride*) de la matriz filtro es la unidad, la correlación de las dimensiones viene dada por:

$$(A_x - C_x + 1) \times (A_y - C_y + 1), \tag{5.2}$$

donde x es la dimensión horizontal e y la dimensión vertical. Es decir, con la notación utilizada en la definición (5.1) tenemos $A_x = m$, $A_y = n$, $C_x = k$ y $C_y = l$

Otra de las opciones a la hora de aplicar un filtro es dar dos o más saltos, en vez de uno. De esta manera, la matriz resultante ocupa menos espacio, pero existe la posibilidad de perder información en el proceso. En la Figura 19, se puede observar cómo este proceso de convolución se ha realizado con un salto (*stride*) de la matriz filtro de dos unidades.

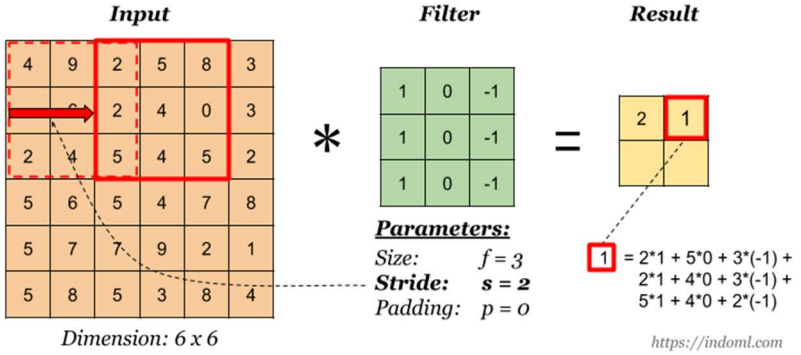

Figura 19: Aplicación de un filtro con doble paso. Fuente: indoml

La convolución al replicarse con diferentes filtros consigue tantas matrices finales como número de filtros se apliquen. Estos filtros pueden ser matrices fijas con funcionalidades determinadas como ilustra la Figura 20, pero en nuestro caso, el proceso para seleccionar los filtros óptimos pasa por la utilización de la red neuronal con el fin de encontrar los que minimicen el error de la red.

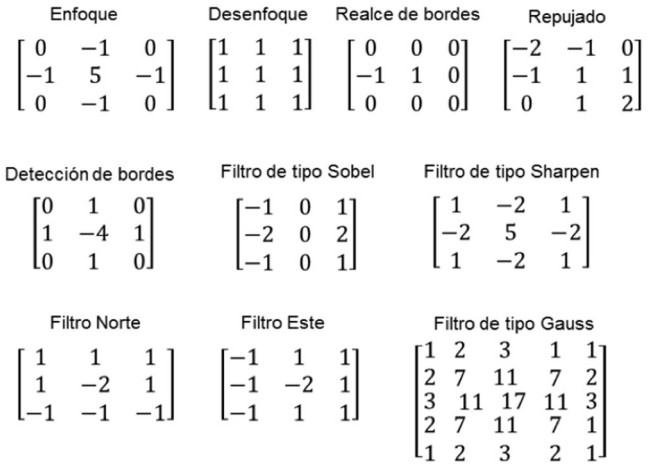

Figura 20: Tipos de filtros más usados. Fuente: [33]

5.2 Padding (rellenado)

Uno de los problemas clásicos que se encuentra a la hora de aplicar filtros es la pérdida de información en el perímetro de la matriz, que empeora con la sucesiva acumulación de filtros que se aplica. Una solución a este problema es el *padding*, que consiste en agregar valores adicionales de relleno alrededor del límite de nuestra matriz. En el ejemplo siguiente, mostrado en la Figura 21, se muestra cómo al aplicar el *padding* a la matriz, ésta aumenta su tamaño de 6x6 a 8x8, de manera que la matriz final conserva el tamaño de la matriz inicial. Por lo tanto, la correlación de las dimensiones tras el *padding* y la convolución viene dada por:

$$(A_x - C_x + p_x + 1) \times (A_y - C_y + p_y + 1), \tag{5.3}$$

donde p_x y p_y corresponden al número de filas y columnas adicionales que añade el *padding*.

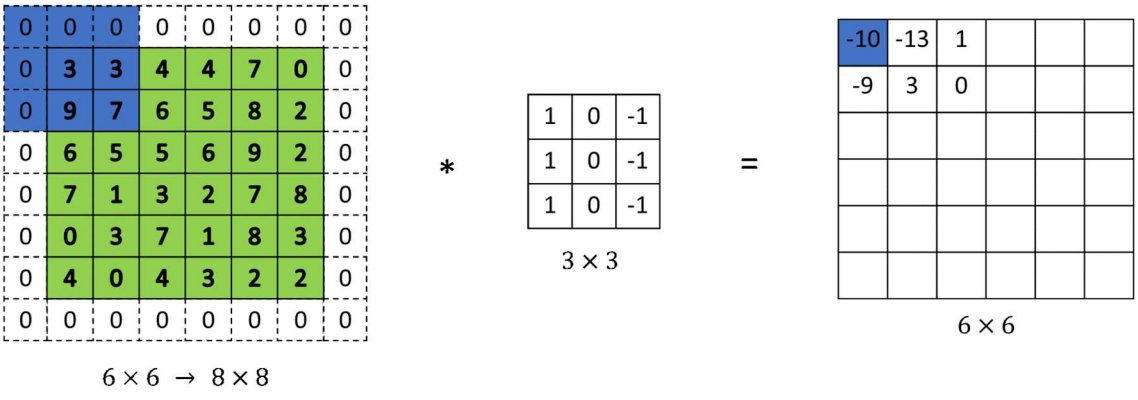

Figura 21: Aplicación de *padding* a una matriz y posterior convolución con un filtro 3x3. Fuente: http://datahacker.rs

5.3 Convolución de matrices de diferentes dimensiones.

Los datos de entrada en la red neuronal no tienen por qué ser bidimensionales. Por ejemplo, en el caso de las fotos a color, nos encontramos una matriz de 3 dimensiones, siendo esta última dimensión los tres colores RGB. En este caso, las dimensiones del filtro deben ser $C_x \times C_y \times 3$, como ilustra la Figura 22. En otros casos, tenemos un vector unidimensional, como es el caso de los datos de series temporales. En conclusión, la dimensión de los filtros se debe adaptar siempre a las dimensiones de los datos de entrada.

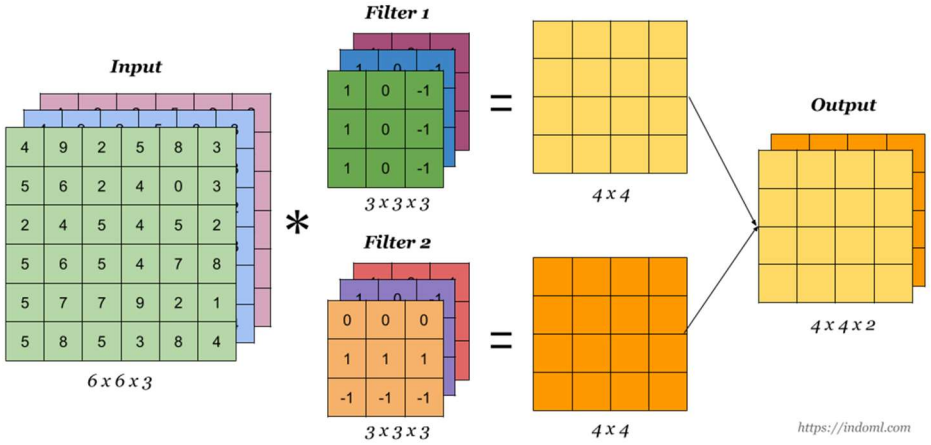

Figura 22: Convolución de matriz 3D con dos filtros 3D. Fuente: https://indoml.com

5.4 Capa Pooling

El concepto de agrupación (*pooling*), publicado en 2010 en el artículo [34], es uno de los pilares fundamentales en los sistemas actuales de redes neuronales a la hora de crear sistemas eficientes. Uno de los problemas fundamentales de crear muchos filtros, es el aumento de la cantidad de datos con los que se va a entrenar a la red, y, por ello, tras la convolución con filtros, generalmente se aplica la operación *pooling.* Este proceso consiste en la aplicación de otro filtro para disminuir el tamaño de la matriz. Los más usados son dos: *max-pooling* y *average pooling*. En el *max-pooling* se elige el valor más alto de la sección que aplica el filtro para el valor de la matriz final, mientras que el a*verage pooling* realiza la media de los valores seleccionados.

Para ilustrar el funcionamiento de este tipo de procesos, se muestra en la Figura 23 un ejemplo donde puede verse el resultado de aplicar ambos filtros a una matriz de tamaño 4x4. El método del *max-pooling* recoge los 4 valores más altos de cada sección 2x2 y el a*verage pooling* una media de todos los valores de cada sección. De esta manera, cada dimensión de la matriz final termina siendo la mitad de la original.

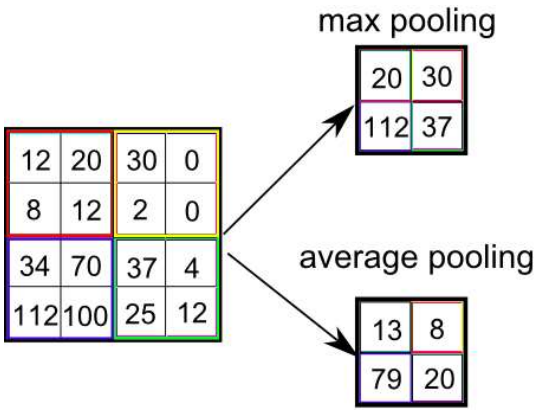

Figura 23: Aplicación de *max-pooling* y *average pooling* a una matriz de tamaño 4x4.

5.5 Ejemplos de redes CNN complejas

Una de las arquitecturas complejas de redes neuronales convolucionales más utilizadas es la llamada *LeNet* [35] (*LeNet-5*). Ésta consta de 2 fases claramente diferenciadas, como se puede apreciar en la Figura 24: una doble convolución con sus respectivos *poolings* y posteriormente, varias capas convergentes de neuronas *fully-connected.* La finalidad de esta red convolucional es sectorizar la información de la imagen inicial divergiéndola en múltiples imágenes para después entrenar la red neuronal con estos datos más específicos.

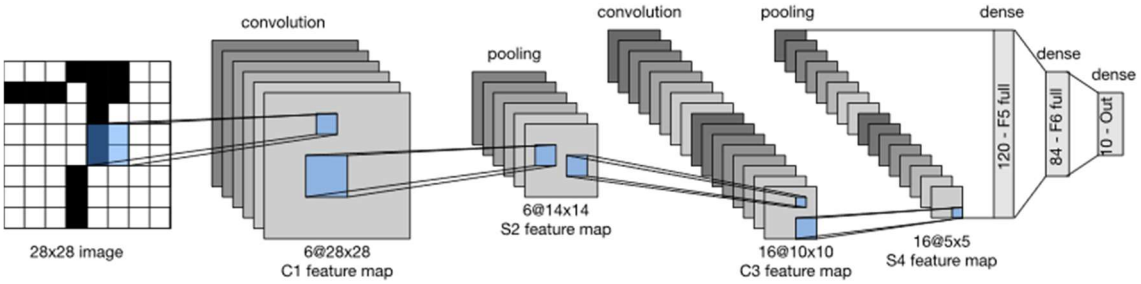

Figura 24: Aplicación de una red *LeNet-5* a una imagen 28x28. Fuente: [36]

Este trabajo se inspirará en la red *LeNet-5* para construir su propia red neuronal convolucional para lidiar con el problema planteado anteriormente.

Otras redes CNN complejas pueden ser:

- AlexNet [37]
- Red de bloques (VGG) [38]
- Red de redes (NiN) [39]
- Red con concatenaciones paralelas (GoogLeNet) [40]

En resumen, en este TFG se utilizarán las redes convolucionales, junto a las diferentes capas asociadas a éstas, para resaltar la información en forma de patrones de las muestras termográficas y derivadas topológicas, para mejorar así el entrenamiento de nuestras redes.