Trabajo de Fin de Grado
Grado de Ingeniería Electrónica, Róbotica y Mecatrónica

Reconocimiento de imágenes con Redes Convolucionales en C

Autor: Álvaro Casas Martínez
Tutor: Alejandro José del Real Torres

Dep. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2017

Trabajo de Fin de Grado
Grado en Ingeniería Electrónica, Robótica y Mecatrónica

Reconocimiento de imágenes con Redes Convolucionales en C

Autor:
Álvaro Casas Martínez

Tutor:
Alejandro José del Real Torres
Profesor titular

Dep. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2017

Proyecto Fin de Carrera: Reconocimiento de imágenes con Redes Convolucionales en C

Autor: Álvaro Casas Martínez

Tutor: Alejandro del Real Torres

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2017

El Secretario del Tribunal

A mi familia

A mi tutor

A mis profesores

A mis compañeros

Agradecimientos

El presente trabajo de fin de grado fue realizado bajo la supervisión del doctor Alejandro José del Real Torres, a quien me gustaría expresar mi más profundo agradecimiento por hacer posible el desarrollo de este estudio. Además de agradecer su paciencia, tiempo y dedicación.

A mis padres Felicia y Diego, por apoyarme en los momentos más difíciles durante estos cuatro años de carrera, y por acompañarme en los buenos momentos a pesar de la distancia.

A mi hermana Mirian, por guiarme por el camino correcto y ayudarme a tomar decisiones importantes.

A mis compañeros y amigos, por haber vivido momentos muy felices con ellos y por el apoyo que he recibido de ellos.

Por último a mis profesores, por enseñarme sus conocimientos e invertir tiempo en que haya un futuro lleno de ingenieros competentes.

Álvaro Casas Martínez

Sevilla, 2017

Resumen

En el siguiente proyecto se explica cómo realizar una red neuronal artificial, en concreto, una red convolucional, para el reconocimiento de símbolos.

Se trata de una red supervisada, la cual se entrena a partir de un conjunto muy amplio de símbolos proporcionados por el dataset MNIST. Una vez la red haya sido entrenada, esta será capaz de clasificar un símbolo (ya sea del propio MNIST o de uno realizado por nosotros mismos) con una alta fiabilidad, del orden del 90%.

Abstract

The next text explain how to make an artificial neuronal network, in particular, a convolutional network, to recognize symbols.

It is a supervised network which is trainned from a very large dataset from MNIST dataset. When the network had been trained, it will be able to classify symbols (from MNIST or symbols whose have been made by us) with a good accuracy, about 90% of accuracy.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

Notación

a	Número real
$\mathbf{a}$	Vector
$\mathbf{A}$	Matriz
$\mathfrak{R}^n$	Conjuntos de números reales de dimensión n
e	Número de Euler
x_j	Elemento j-ésimo del vector $\boldsymbol{x}$
$x^{(i)}$	Columna i-ésima de la matriz $\boldsymbol{X}$
$\boldsymbol{A}^T$	Matriz $\boldsymbol{A}$ traspuesta
$\odot$	Producto de Hadamard
$f\,'(z)$	Derivada de la función $f(z)$
∇_θ	Gradiente con respecto a $\boldsymbol{\theta}$
σ (z)	Función sigmoidea
$P(y = 1\|\boldsymbol{x})$	Probabilidad de que y valga 1 sabiendo $\boldsymbol{x}$
$\frac{\partial \mathrm{J}(\theta)}{\partial \theta_i}$	Derivada parcial de la función $J(\boldsymbol{\theta})$ con respecto al i-ésimo elemento del vector $\boldsymbol{\theta}$

1 INTRODUCCIÓN

El esfuerzo es solo esfuerzo cuando comienza a doler.

- José Ortega y Gassett -

A lo largo de la historia las redes neuronales han sido un importante campo de investigación para su posible uso en varias ramas de la ciencia. El objetivo de su estudio era, tratar de crear modelos articiales de redes neuronales que solucionen problemas difíciles de resolver mediante técnicas algorítmicas convencionales.

La visión de una moderna red neuronal comienza en 1940 con los trabajos de Warren McCulloch y Walter Pitts quienes mostraron que las redes neuronales artificiales se podían estudiar con funciones matemáticas. Posteriormente, Donald Hebb propuso un mecanismo de aprendizaje en las neuronas biólogicas. Seguidamente aparece la red perceptrón y el aprendizaje asociado demostrando la capacidad para llevar a cabo el reconocimiento de patrones.

La continuación del avance de las redes neuronales comienza con el uso de la mecánica estadística para explicar el funcionamiento de la red recurrente y el desarrollo del algoritmo backpropagation.

Hoy en día se sigue investigando en redes neuroanles ya que aunque no aporten soluciones a todos los problemas son herramientas esenciales para determinados problemas.

1.1 Inspiración badada en la biología

Las redes neuronales artificiales están relacionados con sus homólogos biológicos. A continuación, se describen las características de la función cerebral que han inspirado el desarrollo de las redes neuronales artificiales.

El cerebro se compone de un gran número de neuronas interconectadas entre sí. Para el propósito de una red artificial, las neuronas tienen tres componentes principales: las dendritas, el cuerpo celular y el axón. Las dendritas son receptores que llevan señales eléctricas al cuerpo de la célula. El cuerpo de la célula sintetiza las señales entrantes. El axón es una fibra larga que transporta la señal desde el cuerpo celular a otras neuronas. El punto de contacto entre un axón y una dendrita de diferentes células se llama sinapsis. En la siguiente figura se muestra el diagrama esquemático simplificado de dos neuronas biológicas.

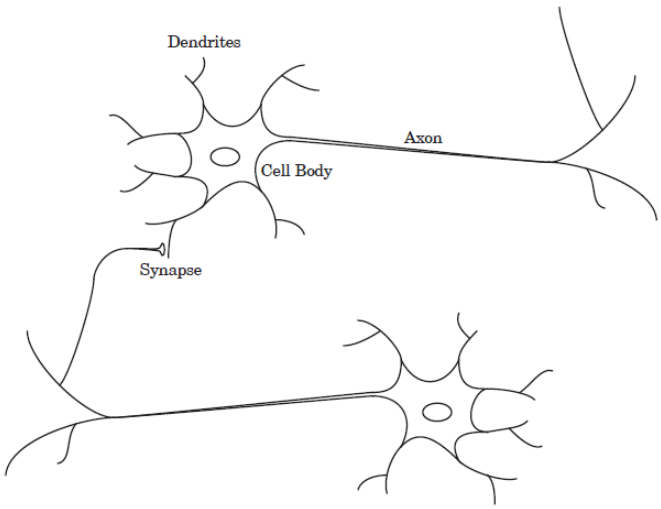


Figura 1-1: Diagrama esquemático de dos neuronas biológicas.

Las principales similitudes entre las redes biológicas y artificiales. En primer lugar, los bloques de construcción de ambas redes son simples dispositivos computacionales que son interconectados. En segundo lugar, las conexiones entre las neuronas determinan la función de la red.

Las redes neuronales artificiales comparten con las biológicas una estructura de computación en paralelo que aumenta la velocidad de resolución de un problema.

1.2 Aplicaciones

Las redes neuronales tienen un gran número de aplicaciones reales en la industria. De hecho ya han sido aplicadas en muchos dispositivos electrónicos comerciales, debido a que éstas, muestran mejores resultados en el reconocimiento de patrones. Algunos campos en los que pueden trabajar las redes neuronales son:

• Reconocimiento de Caracteres: Esta idea tomó mucho auge y fue tomada inicialmente en las palm que fueron muy populares hace algunos años, aún esta aplicación se emplea en algunos tablets y teléfonos celulares táctiles. De hecho, el reconocimiento de caracteres, permite reconocer texto escrito manualmente.

• Compresión de imágenes: Las redes neuronales pueden recibir y procesar grandes cantidades de información a la vez, siendo esto útil en la compresión de imágenes, solo basta hacer una observación con el crecimiento del internet y del diseño de imágenes y de animaciones, cada vez más pesadas.

• Mercado de Valores: El negocio del día a día de la bolsa bursátil es muy complicado, muchos factores pesan en si una acción propuesta subirá o bajará. Desde las RNA se puede examinar una gran cantidad de información de forma rápida y organizar todo de manera que se pueda hacer un adecuado estudio de proyección y predecir el valor de las acciones.

• Procesamiento de alimentos: En este caso se ha implementado el uso de una nariz electrónica que reemplaza a los seres humanos y llevar un adecuado control en la inspección, clasificación de los productos, por ejemplo en las pescaderías industriales, en el control del nivel de acidez de la mayonesa, el seguimiento de la maduración del queso, control de sabores, filtrado de señales etc.

• Medicina: una de las áreas que más ha ganado la atención es en la detección de afecciones cardiopulmonares, es decir compara muchos modelos distintos para identificar similitud en patrones y síntomas de la enfermedad. Estos sistemas ayudan a los médicos con el diagnóstico por el análisis de los síntomas reportados y las resonancias magnéticas y rayos x. También se han usado para dispositivos analizadores de habla para ayudar a

personas con sordera profunda, monitorización de cirugías, predicción de reacciones adversas a un medicamento, entendimiento de causas de ataques epilépticos.

• Milicia: Las redes neuronales juegan un papel importante en el campo de batalla, especialmente en aviones de combate y tanques que son equipados con cámaras digitales de alta resolución que funciona conectado a un computador que continuamente explora el exterior de posibles amenazas. De igual manera se pueden emplear para clasificar señales de radar, creación de armas inteligentes.

• Diagnóstico de máquinas: A nivel industrial cuando una de estas máquinas presenta fallos automáticamente las apaga cuando esto ocurre.

• Análisis de Firmas: Las redes neuronales pueden ser empleadas para la comparación de firmas generadas, (Por ejemplo en los bancos) Esta ha sido una de las primeras aplicaciones implementada a gran escala en USA y también han sido los primeros en usar un chip neuronal.

• Monitoreo de Aviones: controlan el estado de los motores de las aeronaves. Revisan los niveles de vibración de sonido y alertas tempranas de problemas en el motor.

• Biología: Mayor entendimiento del funcionamiento del cerebro, obtención de modelos de la retina.

• A nivel Administrativo: Identificaciones de candidatos para posiciones específicas, optimización de plazas y horarios en líneas de vuelo, minería de datos.

1.3 Actualidad de las redes neuronales

Las redes neuronales alcanzan cada vez mayor auge, teniendo multitud de aplicaciones en campos diversos y dando soluciones sencillas a problemas cuya resolución resulta complicada cuando se emplean máquinas algorítmicas. Aún así, el futuro de las redes neuronales no está todavía claro y será en los próximos años cuando se determine su evolución. Sin embargo y aunque suena a ciencia ficción, las redes neuronales podrán en el futuro permitir:

• Robots que pueden ver, sentir, oler y percibir el mundo que los rodea (De hecho ya existe algo llamado Computación afectiva).

• Predicción de valores e implementación en vehículos para la auto - conducción.

• Composición de música y documentos escritos (obras literarias).

• La comprensión de la información en el Genoma Humano.

• Autodiagnóstico médico por medio de redes neuronales.

1.4 Conclusiones

El mundo de la informática tiene mucho que ganar con las redes neuronales. Su capacidad para aprender por ejemplo, hace que sean extremadamente flexibles y adaptables a un sin número de aplicaciones, convirtiéndolas en una herramienta potente. Cabe destacar que no se hace necesario elaborar un algoritmo para realizar una tarea específica, es decir, no es necesario conocer los mecanismos internos del problema. Las redes neuronales también son adecuadas para los sistemas en tiempo real, por las características de su arquitectura en paralelo, permitiendo así ejecutar cálculos a una gran velocidad y a sus tiempos de respuesta. Las redes neuronales contribuyen también a otras áreas de investigación tales como la neurología y la psicología que se utilizan regularmente para modelar partes de los organismos vivos y para investigar igualmente los mecanismos del cerebro. Por último, hay científicos que afirman que la conciencia podrá ser modelada en un futuro no lejano y que esta será una de las propiedades de las redes neuronales.

2 OBJETIVOS DE UNA RED NEURONAL

Antes de comenzar a analizar y explicar qué es una red neuronal, es importante saber cual es el impulso que ha llevado a usar esta herramienta en lugar de otras que podrían hacer este mismo trabajo.

Con una red neuronal se pretende clasificar una serie de datos. Por tanto, puede decirse que una red es un tipo de clasificador. Algunos clasificadores clásicos son la regresión lineal, regresión logística y regresión suave.

2.1 Regresión lineal

El objetivo de la regresión lineal es el de predecir un objetivo, "y", a partir de un vector $\boldsymbol{x} \in \Re^n$ que contiene una serie de características del ejemplo que quiere predecirse. Un ejemplo real sería el de conocer el precio de una casa a partir de las características que tiene (número de habitaciones, baños, superficie,…). En este caso el valor deseado "y" es el precio de la casa, y lo que conocemos a priori son las características de la casa, $\boldsymbol{x}$. Para poder hacer una buena predicción es necesario un número de ejemplos lo suficientemente grande para conseguir una buena aproximación, por lo tanto, el vector $\boldsymbol{x}$ está formado por varias características de diferentes ejemplos, de tal forma que x_j se refiere a la característica j-ésima. Se denota con una "i" a la i-ésima casa, por lo tanto, $y^{(i)}$ es el precio de la i-ésima casa y $\boldsymbol{x}^{(i)}$el vector de características de la i-ésima casa.

El objetivo es el de encontrar una función $y^{(i)} = h\left(\boldsymbol{x}^{(i)}\right)$ que pueda predecir el precio de la casa al presentarle un vector de características. Para ello, previamente se le ha presentado a la regresión lineal un conjunto de datos de los cuales se conocen los precios de cada casa junto a sus características. Cuanto mayor sea el número de este conjunto de datos más fiable será la predicción del precio al presentar un nuevo vector de características.

De forma general, para determinar la función $h(\boldsymbol{x})$ es necesario modelarla de manera lineal

$$h_\theta(\boldsymbol{x}) = \sum_j \theta_j x_j = \boldsymbol{\theta}^T \boldsymbol{x} \qquad (2\text{-}1)$$

donde $h_\theta(\boldsymbol{x})$ representa una larga familia de funciones parametrizadas por $\boldsymbol{\theta}$. Con este modelo de $h_\theta(\boldsymbol{x})$ lo que se busca es encontrar el valor de $\boldsymbol{\theta}$ para que $h_\theta(\boldsymbol{x}^{(i)})$ sea lo más cercano a $y^{(i)}$. Esta búsqueda se puede hacer mediante la función coste $J(\boldsymbol{\theta})$ de tal manera que nuestro objetivo es minimizar el valor de esta función. Como ejemplo la función coste puede tener la siguiente expresión:

$$J(\boldsymbol{\theta}) = \frac{1}{2}\sum_i \left(h_\theta\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right) = \frac{1}{2}\sum_i \left(\boldsymbol{\theta}^T \boldsymbol{x}^{(i)} - y^{(i)}\right)^2 \qquad (2\text{-}2)$$

Para encontrar el $\boldsymbol{\theta}$ que minimice a $J(\boldsymbol{\theta})$ hay varios algoritmos que realizan esta labor, entre ellos el gradiente descendente el cual necesita conocer el valor de $J(\boldsymbol{\theta})$ y el de $\nabla_\theta J(\boldsymbol{\theta})$, también conocido como el gradiente de la función coste que tiene la siguiente forma:

$$\nabla_\theta J(\boldsymbol{\theta}) = \begin{bmatrix} \frac{\delta J(\boldsymbol{\theta})}{\delta \theta_1} \\ \frac{\delta J(\boldsymbol{\theta})}{\delta \theta_2} \\ \vdots \\ \frac{\delta J(\boldsymbol{\theta})}{\delta \theta_n} \end{bmatrix} \quad (2\text{-}3)$$

Y en especial para nuestro ejemplo, la derivada de la función coste con respecto a un θ_j viene dado por:

$$\frac{\delta J(\boldsymbol{\theta})}{\delta \theta_j} = \sum_i x_i^{(i)} \left(h_\theta\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right) \quad (2\text{-}4)$$

2.2 Regresión logística

La regresión logística consiste en clasificar una entrada de forma discreta. Por ejemplo, si se tienen varias imágenes las cuales representan el número 0 o número 1, la regresión logística sería capaz de clasificar una imagen (si esa imagen tiene el numero 0 o 1) con una cierta probabilidad. Por lo tanto, hace una clasificación binaria del problema.

En la regresión logística se predice la probabilidad de que un ejemplo dado pertenezca a la familia "1" o a la familia "0". Esta probabilidad se calcula a través de estas funciones:

$$P(\boldsymbol{y} = 1|\boldsymbol{x}) = h_\theta(\boldsymbol{x}) = \frac{1}{1 + \exp(-\boldsymbol{\theta}^T \boldsymbol{x})} = \sigma(\boldsymbol{\theta}^T \boldsymbol{x})$$
$$P(\boldsymbol{y} = 0|\boldsymbol{x}) = 1 - P(\boldsymbol{y} = 1|\boldsymbol{x}) = 1 - h_\theta(\boldsymbol{x}) \quad (2\text{-}5)$$

Donde $\sigma(z) = \frac{1}{1+\exp(-z)}$, se conoce como la función sigmoidea y tiene forma de "S", esta función ajusta los valores de $\boldsymbol{\theta}^T \boldsymbol{x}$ a un rango de [0, 1]. El objetivo es calcular para un valor de $\boldsymbol{\theta}$, cuál es la probabilidad $P(\boldsymbol{y} = 1|\boldsymbol{x}) = h_\theta(\boldsymbol{x})$, que será grande cuando $\boldsymbol{x}$ pertenece a la familia "1" y pequeña cuando pertenezca a la "0".

Al igual que con la regresión lineal también se calcula la función coste como:

$$J(\boldsymbol{\theta}) = -\sum_i \left(y^{(i)} \log\left(h_\theta\left(\boldsymbol{x}^{(i)}\right)\right) + (1 - y^{(i)}) \log(1 - h_\theta(\boldsymbol{x}^{(i)}))\right) \quad (2\text{-}6)$$

Una vez encontrada $\boldsymbol{\theta}$ que minimice $J(\boldsymbol{\theta})$, se calculan las probabilidades de que el ejemplo pertenezca a una familia u otra y se asigna el ejemplo a la familia que mayor probabilidad tiene de pertenecer.

De la misma forma que en la regresión lineal, para minimizar la función coste también se calculará $\nabla_\theta J(\theta)$.

$$\nabla_\theta J(\boldsymbol{\theta}) = \sum_i x_i^{(i)} \left(h_\theta\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right) \quad (2\text{-}7)$$

Un ejemplo de la regresión lineal se representa en la siguiente imagen:

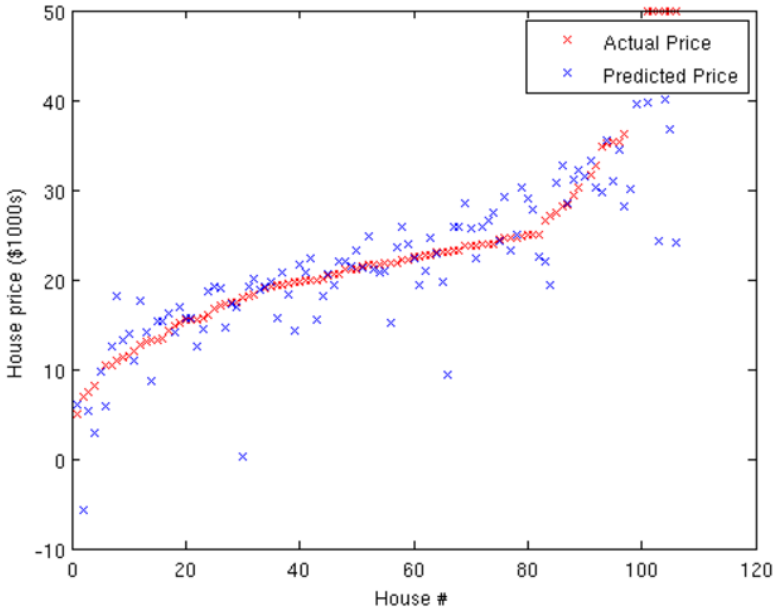


Figura 2-1: Gráfica en Matlab de precios reales y predichos.

donde las cruces rojas indican el precio real de una casa y las cruces azules indican el precio que predice la regresión lineal.

2.3 Regresión softmax

Se basa en la regresión logística pero extendida a clasificar ejemplos teniendo más de dos familias. De tal manera que $y^{(i)} \in \{1,\ldots, K\}$, donde K es el numero de familias.

Como en la regresión logista, para un ejemplo dado, se quiere estimar la probabilidad $P(\boldsymbol{y} = k|\boldsymbol{x})$ para cada valor de k $(k = 1, \ldots, K)$. Por lo tanto, se obtiene un vector K-dimensional (cuya suma de todos los elementos es 1) con las probabilidades estimadas. Concretamente, el modelo de $h_\theta(\boldsymbol{x})$ toma la forma de:

$$h_\theta(\boldsymbol{x}) = \begin{bmatrix} P(y = 1|\boldsymbol{x};\boldsymbol{\theta}) \\ P(y = 2|\boldsymbol{x};\boldsymbol{\theta}) \\ \vdots \\ P(y = K|\boldsymbol{x};\boldsymbol{\theta}) \end{bmatrix} = \frac{1}{\sum_{j=1}^{K} \exp(\theta^{(j)^T} x)} \begin{bmatrix} \exp(\boldsymbol{\theta}^{(1)^T}\boldsymbol{x}) \\ \exp(\boldsymbol{\theta}^{(2)^T}\boldsymbol{x}) \\ \vdots \\ \exp(\boldsymbol{\theta}^{(K)^T}\boldsymbol{x}) \end{bmatrix} \quad (2\text{-}8)$$

Aquí $\boldsymbol{\theta}^{(1)}, \boldsymbol{\theta}^{(2)}, \ldots, \boldsymbol{\theta}^{(K)} \in \Re^n$ y son los parámetros del modelo. Hay que tener en cuenta que el término $\frac{1}{\sum_{j=1}^{K} \exp(\boldsymbol{\theta}^{(j)^T}\boldsymbol{x})}$ normaliza la distribución para que la suma de todas las componentes sea 1.

Para una mayor facilidad a la hora de expresar algunas expresiones se denota θ como una matriz n por K expresado por un vector fila cuyas componentes son vectores columnas:

$$\boldsymbol{\theta} = \begin{bmatrix} | & | & | & | \\ \boldsymbol{\theta}^{(1)} & \boldsymbol{\theta}^{(1)} & \ldots & \boldsymbol{\theta}^{(K)} \\ | & | & | & | \end{bmatrix} \quad (2\text{-}9)$$

Como en los anteriores clasificadores hay que calcular el coste de la función:

$$J(\boldsymbol{\theta}) = -\left[\sum_{i=1}^{m}\sum_{k=1}^{K} 1\{y^{(i)} = k\} \log \frac{\exp(\boldsymbol{\theta}^{(k)^T}\boldsymbol{x}^{(i)})}{\sum_{j=1}^{K} \exp(\boldsymbol{\theta}^{(j)^T}\boldsymbol{x}^{(i)})}\right] \quad (2\text{-}10)$$

Donde $1\{.\}$ es la función indicadora que vale 1 cuando la condición es verdadera y 0 cuando es falsa. Por ejemplo "$1\{2+2=4\}=1$" y "$1\{1+2=6\}=0$".

Una vez calculado el coste se computa el gradiente del coste y a partir de un algoritmo se resuelve el problema de convergencia.

$$\nabla_{\boldsymbol{\theta}^{(k)}} J(\boldsymbol{\theta}) = -\sum_{i=1}^{m} \left[\boldsymbol{x}^{(i)} \left(1\{y^{(i)} = k\} - P\left(y^{(i)} = k \middle| \boldsymbol{x}^{(i)}; \boldsymbol{\theta}\right) \right) \right] \quad (2\text{-}11)$$

3 REDES NEURONALES MULTICAPA

Se considera un problema de aprendizaje supervisado en el cual se tiene acceso a los ejemplos de entrenamiento $(\boldsymbol{x}^{(i)}, y^{(i)})$. Las redes neuronales son capaces de dar un buen ajuste a una función compleja y no lineal $h_{W,b}(\boldsymbol{x})$, con parámetros W, b que se pueden modificar dependiendo del problema.

Antes de empezar a describir las redes neuronales, se explica el modelo más simple de una red neuronal, la cual contiene una única neurona. La siguiente imagen muestra el modelo de la red neuronal:

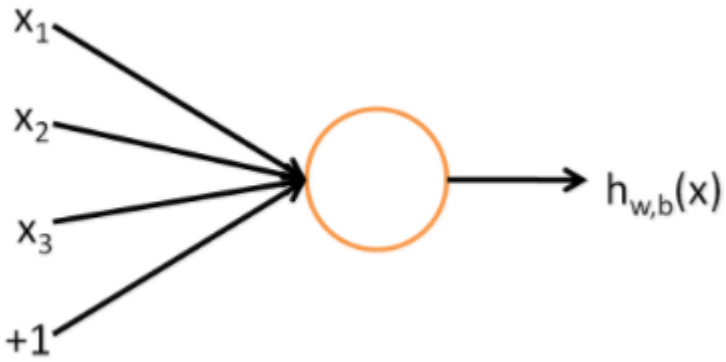


Figura 3-1: Esquema de una red neuronal simple.

Esta neurona se comporta como una unidad computacional que toma como entradas x_1, x_2, x_3 (y un +1 que es un término de interceptación) y como salidas $h_{W,b}(\boldsymbol{x}) = f(\boldsymbol{W}^T\boldsymbol{x}) = f(\sum_{i=1}^{3} W_i x_i + b)$, donde $f: \Re \to \Re$ es conocida como la función de activación. Se elegirá como función de activacion la función sigomoidea para explicar los siguientes apartados:

$$f(z) = \frac{1}{1 + \exp(-z)} \qquad (3\text{-}1)$$

Por lo tanto, este modelo de una única neurona se comportaría del mismo modo que la regresión logística.

3.1 Modelo de una red multicapa

Una red neuronal es una asociación de muchas neuronas simples, así pues, la salida de una neurona puede ser la entrada de otra. En la siguiente imagen se representa una red neuronal pequeña:

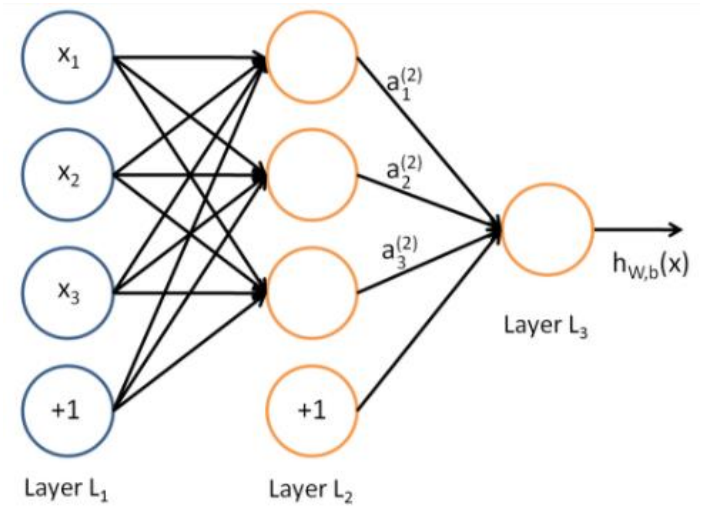


Figura 3-2: Esquema de una red neuronal multicapa.

En esta figura, se han representado las entradas con círculos al igual que las neuronas. Los círculos con el valor "+1" se conocen como bias y corresponden al término de intercepción. La capa que se encuentra a la izquierda (Layer L_1) se llama capa de entrada, y la de la derecha (Layer L_3) es la capa de salida (en este ejemplo la capa de salida solo tiene un nodo pero podrían ser más). La capa del medio se denomina capa oculta. Sin contar las bias, se puede decir que en este ejemplo la red neuronal tiene tres unidades de entrada, tres unidades ocultas y una unidad de salida.

Para denotar el número de capas de la red se usa n_l, en este ejemplo $n_l = 3$. También se etiqueta la capa l-ésima como L_l, en este ejemplo L_1 es la capa de entrada y L_{nl} es la capa de salida. La red neuronal tiene un conjunto de parámetros $(\boldsymbol{W}, b) = \left(\boldsymbol{W}^{(1)}, b^{(1)}, \boldsymbol{W}^{(2)}, b^{(2)}\right)$, donde $W_{ij}^{(1)}$ denota el parámetro (o peso) asociado a la conexión entre la unidad j en la capa l y la unidad i en la capa $l+1$. De la misma forma, $b_i^{(1)}$ es la bias asociada a la unidad i en la capa $l+1$. Por lo tanto, en el ejemplo, $\boldsymbol{W}^{(1)}$ pertenece a $\Re^{3x3}$, y $\boldsymbol{W}^{(2)}$ pertenece a $\Re^{1x3}$.

Se usa $a_i^{(l)}$ para denotar la activación (el valor de salida una vez evaluada la función de activación) de una unidad i en la capa l. Para $l = 1$, también se usa $a_i^{(1)} = x_i$ para denotar la entrada i-ésima. Dado un conjunto de parámetros $\boldsymbol{W}, b$ la red define una $h_{W,b}(\boldsymbol{x})$ que tiene como salida un número real. El cálculo que la red neuronal representa viene dado por las siguientes expresiones:

$$\begin{aligned} a_1^{(2)} &= f\left(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3 + b_1^{(1)}\right) \\ a_2^{(2)} &= f\left(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3 + b_2^{(1)}\right) \\ a_3^{(2)} &= f\left(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3 + b_3^{(1)}\right) \\ h_{W,b} = a_1^{(3)} &= f\left(W_{11}^{(2)}a_1^{(2)} + W_{12}^{(2)}a_2^{(2)} + W_{13}^{(2)}a_3^{(2)} + b_1^{(2)}\right) \end{aligned} \qquad (3\text{-}2)$$

También se denota $z_i^{(l)}$ a la suma total (incluyendo los pesos) de las entradas para cada unidad i en la capa l, incluyendo las bias. Por ejemplo, $z_i^{(2)} = \sum_{j=1}^{n} W_{ij}^{(1)}x_j + b_i^{(1)}$, así que $a_1^{(l)} = f\left(z_i^{(l)}\right)$.

Se puede llegar a una notación más compacta si se aplica la función $f(.)$ tal que así $f([z_1, z_2, z_3]) = [f(z_1), f(z_2), f(z_3)]$ Con esta notación las ecuaciones anteriores quedarían como:

$$\boldsymbol{z}^{(2)} = \boldsymbol{W}^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)} \qquad (3\text{-}3)$$

$$\boldsymbol{a}^{(2)} = f\left(\boldsymbol{z}^{(2)}\right)$$
$$\boldsymbol{z}^{(3)} = \boldsymbol{W}^{(2)}\boldsymbol{a}^{(2)} + \boldsymbol{b}^{(2)}$$
$$h_{W,b}(\boldsymbol{x}) = \boldsymbol{a}^{(3)} = f\left(\boldsymbol{z}^{(3)}\right)$$

A este paso se le conoce como forward propagation. De una manera más general para calcular la activación de la capa $l + 1$:

$$\boldsymbol{z}^{(l+1)} = \boldsymbol{W}^{(l)}\boldsymbol{a}^{(l)} + \boldsymbol{b}^{(l)}$$
$$\boldsymbol{a}^{(l+1)} = f\left(\boldsymbol{z}^{(l+1)}\right) \qquad (3\text{-}4)$$

A lo largo de la explicación se ha profundizado en el modelo de la red anterior pero hay otras posibles arquitecturas diferentes para una red neuronal, por ejemplo, redes con múltiples capas ocultas. La elección más común es una red con n_l capas donde la capa 1 es la de entrada, la capa n_l es la capa de salida, y cada capa l está conectada a la capa $l + 1$a través de las diferentes unidades que la componen. En esta configuración, para calcular la salida de la red se puede calcular de froma sucesiva todas las activaciones en la capa L_2, después en la capa L_3 y así sucesivamente hasta la capa L_{nl}, usando las ecuaciones anteriores que describen el forward propagation. Este es un ejemplo de una red neurnal feedforward, ya que el gráfico de conecividad no tiene ciclos ni bucles.

Las redes neuronales también pueden tener varias unidades de salida. En la siguiente imagen se representa una red con dos capas ocultas L_2 y L_3, y dos unidades de salida en la capa L_4:

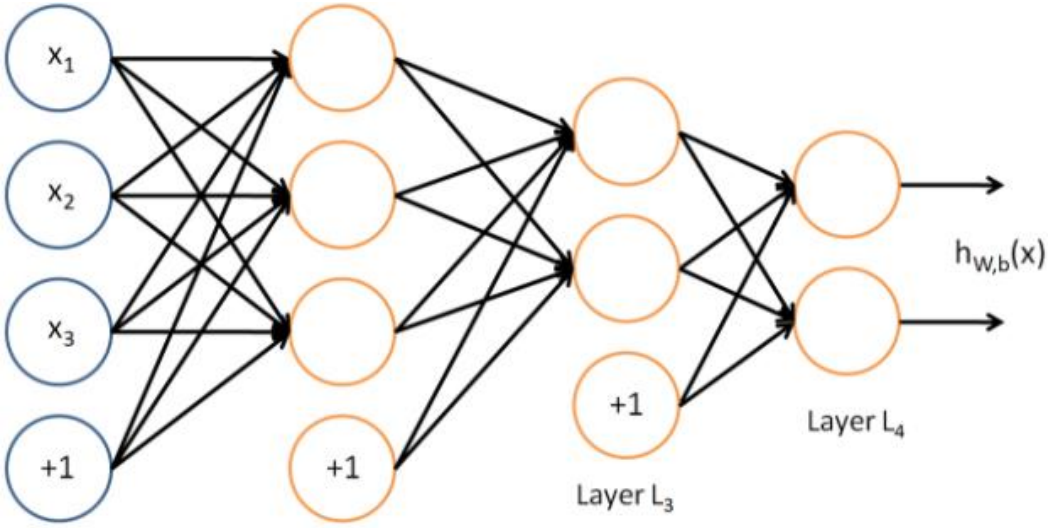


Figura 3-3: Esquema de una red neuronal multicapa con dos capas ocultas.

Para entrenar esta red se necesitan los ejemplos de entrenamiento $(\boldsymbol{x}^{(i)}, y^{(i)})$ donde $y^{(i)}$ pertenece a $\Re^2$. Este tipo de redes son útiles si hay más de una salida. Por ejemplo, una aplicación puede ser un diagnóstico médico, donde el vector $\boldsymbol{x}$ debería contener las caraterísticas de un paciente y el vector $\boldsymbol{y}$ el tipo de enfermedad que puede tener.

3.2 Entrenamiento de una red: Algoritmo de retropropagación

Backpropagation es uno de muchos métodos para entrenar una red neuronal. Es un esquema de entrenamiento supervisado, que significa, aprende de datos de entrenamiento (hay un supervisor para guiar su aprendizaje).

De una forma simple, backpropagation es como aprender de los errores. El supervisor corrige la red neuronal cuando comete errores.

El objetivo del aprendizaje es asignar los correctos pesos para que al presentar a la red una nueva entrada esta

sea capaz de clasificarla con un buen porcentaje de predicción.

3.2.1 Idea intuitiva

Para explicar este algoritmo se presenta el siguiente ejemplo de red neuronal (ver figura 3-4). La red tiene dos nodos en la capa de entrada (sin contar la bias). También tiene una capa oculta con dos nodos. La salida de la capa tiene dos nodos también, el nodo superior emite la probabilidad de "pass", mientras que el nodo inferior genera la probabilidad de "fail".

En las tareas de clasificación se usa generalmente una función Softmax como función de activación en la capa de salida para asegurar que las salidas son probabilidades y su suma es 1 de tal forma que:

$$Probabilidad_{pass} + Probabilidad_{fail} = 1 \qquad (3\text{-}5)$$

Paso 1: Forward propagation

Todos los pesos de la red se asignan de forma aleatoria. Consideramos el nodo de la capa oculta marcado como "V". Los pesos de las conexiones de las entradas a los nodos son w1,w2 y w3.

La red entonces toma como entrada el primer ejemplo de entrenamiento. Por ejemplo, para una entrada 35 y 67, la probabilidad de "pass" es 1.

- Entrada a la red = [35, 67]
- Salida deseada de la red (objetivo) = [1, 0]

Entonces la salida "V" puede calcularse como:

$$V = f(1 * w1 + 35 * w2 + 67 * w3)$$

Del mismo modo se calcula el otro nodo de la capa oculta. Las salidas de los dos nodos en la capa oculta actúan como entradas a los dos nodos en la capa de salida. Esto permite calcular las probabilidades de salida de los nodos en la capa de salida.

Se suponen que las probabilidades de salida de los dos nodos en la capa de salida son 0.4 y 0.6 respectivamente. Podemos ver que las probabilidades calculadas están muy lejos de las probabilidades deseadas (1 y 0 respectivamente), por lo que la red de la figura se dice que tiene una salida incorrecta.

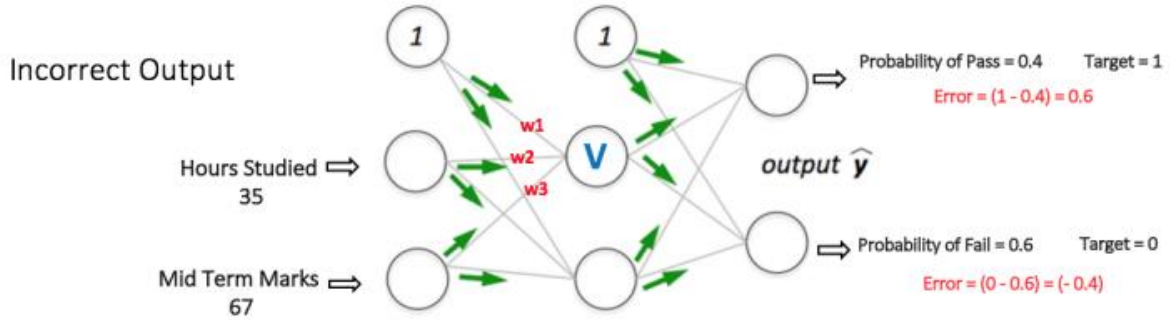


Figura 3-4: Esquema del cálculo de la salida de una red.

Paso 2: Backpropagation y ajuste de pesos

Se calcula el error total en los nodos de salida y se "propagan" estos errores de nuevo a través de la red usando backpropagation para calcular los gradientes. Luego se utiliza un método de optimizaión como Gradient Descent para ajustar todos los pesos en la red con el objetivo de reducir el error en la capa de salida. Esto se muestra en la siguiente figura:

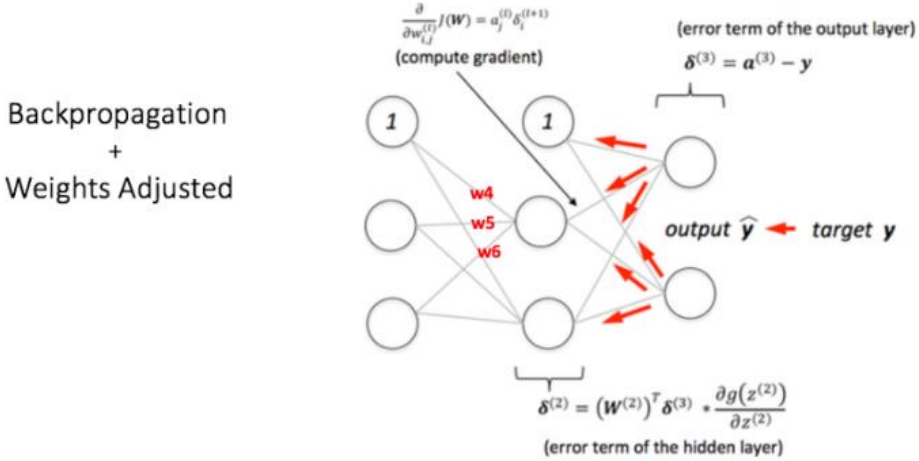


Figura 3-5: Esquema de la retropropagación en una red.

Si se vuelve a introducir el mismo ejemplo a la red, la red debería funcionar mejor que antes, ya que los pesos se han ajustado para minimizar el error de predicción. Como se muestra en la siguiente figura, los errores en los nodos de salida ahora se reducen a [0.2, -0.2] en comparación con [0.6, -0.4] anterior. Esto significa que la red ha aprendido a clasificar correctamente el primer ejemplo de entrenamiento.

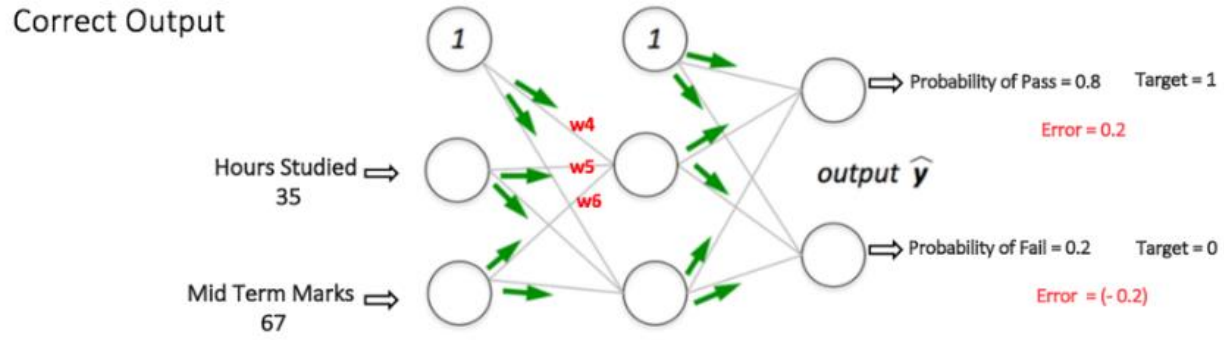


Figura 3-6: Esquema de aprendizaje de una red.

Se repite este proceso con todos los demás ejemplos de entrenamiento del conjunto de datos y la red aprenderá esos ejemplos.

3.2.2 Explicación matemática

Se supone que se tiene un conjunto de entrenamiento $\{(\boldsymbol{x}^{(1)}, y^{(1)}), \ldots, (\boldsymbol{x}^{(m)}, y^{(m)})\}$ de m ejemplos de entrenamiento. Se puede entrenar la red neuronal usando batch gradient descent. Profundizando más, para un ejemplo de entrenamiento $(\boldsymbol{x}, y)$, se define la función coste respecto al único ejemplo como:

$$J(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, y) = \frac{1}{2}\left|\left|h_{W,b}(\boldsymbol{x}) - y\right|\right|^2 \qquad (3\text{-}6)$$

Esto es una función coste que tiene forma de mínimos cuadrados. Dado un conjunto de entrenamiento con m ejemplos, se define la función coste total como:

$$J(\boldsymbol{W},\boldsymbol{b}) = \left[\frac{1}{m}\sum_{i=1}^{m} J\left(\boldsymbol{W},\boldsymbol{b};\boldsymbol{x}^{(i)},y^{(i)}\right)\right] + \frac{\lambda}{2}\sum_{l=1}^{n_{l-1}}\sum_{j=1}^{s_l}\sum_{i=1}^{s_{l+1}}\left(W_{ji}^{(l)}\right)^2$$
$$= \left[\frac{1}{m}\sum_{i=1}^{m}\frac{1}{2}\left\|h_{W,b}\left(\boldsymbol{x}^{(i)}\right) - y^{(i)}\right\|^2\right] + \frac{\lambda}{2}\sum_{l=1}^{n_{l-1}}\sum_{j=1}^{s_l}\sum_{i=1}^{s_{l+1}}\left(W_{ji}^{(l)}\right)^2 \quad (3\text{-}7)$$

El primer término en la definición de $J(\boldsymbol{W},\boldsymbol{b})$ es una media de la suma de términos referidos a los errores cuadráticos. El segundo término es un término de regularización (también llamado como término weight decay) que tiende a decrementar la magnitud de los pesos y ayuda a prevenir el overfitting.

El parámetro weight decay λ controla la importancia de los dos términos. Se nota que $J(\boldsymbol{W},\boldsymbol{b};\boldsymbol{x},y)$ es el coste error cuadrático con respecto a un único ejemplo y $J(\boldsymbol{W},\boldsymbol{b})$ es la función coste total (con todos los ejemplos) el cual incluye un término weight decay.

Esta función coste se usa con frecuencia tanto para problemas de clasificación como de regresión. Para clasificación, se deja $y = 0$ o 1 representa dos clases (se recalca que la función de activación sigmoidea tiene una salida comprendida en [0, 1]). Para problemas de regresión, primero se escalan las salidas para asegurarse de que están comprendidas en un rango [0, 1].

El objetivo es minimizar $J(\boldsymbol{W},\boldsymbol{b})$ como una función de **W** y **b**. Para entrenar la red neuronal, se inicializa cada parámetro $W_{ij}^{(l)}$ y cada $b_i^{(l)}$ con unos valores muy próximos a cero y después aplicar un algoritmo optimizado como batch gradient descent. Hay que hacer incapié en que hay que inicializar los parámetros con números aleatorios. Si todos los parámetros comienzan con los mismo valores entonces las unidades de las capas ocultas acabarán aprendiendo la misma función de entrada ($W_{ij}^{(1)}$ será la misma para todos los valores de *i*, así pues $a_1^{(2)} = a_2^{(2)} = a_3^{(2)} = \cdots$ para cualquier entrada de *x*). La inicialización aleatoria sirve para romper esta simetría.

Para entender mejor como se actualizan los parámetros **W** y **b** se muestra una iteración del gradient descent:

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha\frac{\delta}{\delta W_{ij}^{(l)}}J(\boldsymbol{W},\boldsymbol{b})$$
$$b_i^{(l)} = b_i^{(l)} - \alpha\frac{\delta}{\delta b_i^{(l)}}J(\boldsymbol{W},\boldsymbol{b}) \quad (3\text{-}8)$$

Donde α es el ratio de aprendizaje. El paso importante es el de calcular las derivadas anteriores. A continuación se describe el algoritmo backpropagation, el cual, calcula de forma eficiente estas derivadas parciales.

Primero se describe como backpropagation se usa para calcular $\frac{\delta}{\delta W_{ij}^{(l)}}J(W,b;x,y)$ y $\frac{\delta}{\delta b_i^{(l)}}J(W,b;x,y)$, la derivada parcial del coste $J(\boldsymbol{W},\boldsymbol{b};\boldsymbol{x},y)$ definida con respecto a un ejemplo $(\boldsymbol{x}, y)$. Una vez se calculen éstos, se observa que la derivada de la función coste total $J(W,b)$ se calcula como:

$$\frac{\delta}{\delta W_{ij}^{(l)}}J(\boldsymbol{W},\boldsymbol{b}) = \left[\frac{1}{m}\sum_{i=1}^{m}\frac{\delta}{\delta W_{ij}^{(l)}}J\left(\boldsymbol{W},\boldsymbol{b};\boldsymbol{x}^{(i)},y^{(i)}\right)\right] + \lambda W_{ij}^{(l)}$$
$$\frac{\delta}{\delta b_i^{(l)}}J(\boldsymbol{W},\boldsymbol{b}) = \left[\frac{1}{m}\sum_{i=1}^{m}\frac{\delta}{\delta b_i^{(l)}}J\left(\boldsymbol{W},\boldsymbol{b};\boldsymbol{x}^{(i)},y^{(i)}\right)\right] \quad (3\text{-}9)$$

La intuición detrás del algoritmo backpropagation es como sigue. Dado un ejemplo de entrenamiento (**x**, y), primero se ejecuta un paso hacia adelante para calcular todas las activaciones a través de la red, incluyendo el valor de salida hipótetico $h_{W,b}(\boldsymbol{x})$. Entonces, para cada nodo *i* en la capa *l*, nos gustaría calcular un término error $\delta_i^{(l)}$ que mida cuanto ese nodo fue responsable de cualquier error en la salida. Para un nodo de salida, se puede medir esta diferencia directamente entre las activaciones de la red y el valor verdadero del objetivo, y usar

esto para definir $\delta_i^{(n_l)}$ (donde la capa n_l es la capa de salida) pero esto no se puede hacer para las unidades ocultas. Para las neuronas de las capas ocultas se calculará $\delta_i^{(l)}$ según una media ponderada de los términos de error de los nodos que toman $a_i^{(l)}$ como una entrada. A continuación se presenta los pasos del algoritmo backpropagation:

1. Realizar un paso feedforward calculando las activaciones para las capas L_2, L_3 así sucesivamente hasta la capa de salida L_{n_l}.
2. Para cada unidad de salida *i* en la capa n_l (la capa de salida), fijar

$$\delta_i^{(n_l)} = \frac{\delta}{\delta z_i^{(n_l)}} \frac{1}{2} \left\| y - h_{W,b}(\boldsymbol{x}) \right\|^2 = -\left(y_i - a_i^{(n_l)}\right) . f'\left(z_i^{(n_l)}\right) \quad (3\text{-}10)$$

3. Para $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$

   Para cada nodo *i* en la capa *l*, fijar

$$\delta_i^{(l)} = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) f'\left(z_i^{(l)}\right) \quad (3\text{-}11)$$

4. Calcular las derivadas parciales deseadas, que se dan como:

$$\begin{aligned} \frac{\delta}{\delta W_{ij}^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}; x, y) &= a_j^{(l)} \delta_i^{(l+1)} \\ \frac{\delta}{\delta b_i^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}; x, y) &= \delta_i^{(l+1)} \end{aligned} \quad (3\text{-}12)$$

Se puede reescribir e algoritmo usando notación vectorial. El operador "⊙" denota el producto Hadamard. Y teniendo en cuenta que $f'([z_1, z_2, z_3]) = [f'(z_1), f'(z_2), f'(z_3)]$. El algoritmo se escribe como:

1. Realizar un paso feedforward calculando las activaciones para las capas L_2, L_3 así sucesivamente hasta la capa de salida L_{n_l}.
2. Para cada unidad de salida *i* en la capa n_l (la capa de salida), fijar

$$\boldsymbol{\delta}^{(n_l)} = -\left(\boldsymbol{y} - \boldsymbol{a}^{(n_l)}\right) \odot f'\left(\boldsymbol{z}^{(l)}\right) \quad (3\text{-}13)$$

3. Para $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$, fijar

$$\boldsymbol{\delta}^{(l)} = \left(\left(\boldsymbol{W}^{(l)}\right)^T \boldsymbol{\delta}^{(l+1)}\right) \odot f'\left(\boldsymbol{z}^{(l)}\right) \quad (3\text{-}14)$$

4. Calcular las derivadas parciales deseadas, que se dan como:

$$\begin{aligned} \nabla_{W^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, y) &= \boldsymbol{\delta}^{(l+1)} \left(\boldsymbol{a}^{(l)}\right)^T \\ \nabla_{b^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, y) &= \boldsymbol{\delta}^{(l+1)} \end{aligned} \quad (3\text{-}15)$$

En los pasos 2 y 3 anteriores, se necesita calcua $f'\left(z_i^{(l)}\right)$ para cada valor de *i*. Asumiendo que $f(z)$es la función de activación sigmoidea, tendríamos $a_i^{(l)}$ almacenada tras el paso forward a través la red. Por lo tanto, usando la expresión con la que trabajamos anteriormente para $f'(z)$, podemos calcular esta como:

$$f'\left(z_i^{(l)}\right) = a_i^{(l)}\left(1 - a_i^{(l)}\right) \quad (3\text{-}16)$$

Finalmente, se describe el algoritmo gradient descent completo. En el pseudocódigo siguiente, $\Delta \boldsymbol{W}^{(l)}$ es una matriz (de la misma dimensión que $\boldsymbol{W}^{(l)}$), y $\Delta \boldsymbol{b}^{(l)}$ es un vector (de la misma dimensión que $\boldsymbol{b}$). Nosotros implementamos una iteración del bach gradient descent como:

1. Fijar $\Delta \boldsymbol{W}^{(l)} := 0, \Delta \boldsymbol{b}^{(l)} := 0$ para todo *l*.
2. Para *i* = 1 a *m*,
    a. Usar backpropagation para calcular $\nabla_{\boldsymbol{W}^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, y)$ y $\nabla_{\boldsymbol{b}^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, y)$.
    b. Fijar $\Delta \boldsymbol{W}^{(l)} := \Delta \boldsymbol{W}^{(l)} + \nabla_{\boldsymbol{W}^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, y)$.
    c. Fijar $\Delta \boldsymbol{b}^{(l)} := \Delta \boldsymbol{b}^{(l)} + \nabla_{\boldsymbol{b}^{(l)}} J(\boldsymbol{W}, \boldsymbol{b}; \boldsymbol{x}, y)$.
3. Actualizar los parámetros:

$$\begin{aligned} \boldsymbol{W}^{(l)} &= \boldsymbol{W}^{(l)} - \alpha\left[\left(\frac{1}{m}\right)\Delta \boldsymbol{W}^{(l)} + \lambda \boldsymbol{W}^{(l)}\right] \\ \boldsymbol{b}^{(l)} &= \boldsymbol{b}^{(l)} - \alpha\left[\frac{1}{m}\Delta \boldsymbol{b}^{(l)}\right] \end{aligned} \quad (3\text{-}17)$$

Para entrenar la red se repiten los pasos del gradient descent para reducir la duncion coste $J(\boldsymbol{W}, \boldsymbol{b})$.

3.3 Ejemplo del funcionamiento de una red neuronal multicapa

El siguiente ejemplo describe el proceso de enseñanza de la red neuronal multicapa empleando backpropagation. Para ilustrar este proceso se utiliza la red neuronal de tres capas con dos entradas y una salida, que se muestra en la siguiente imagen:

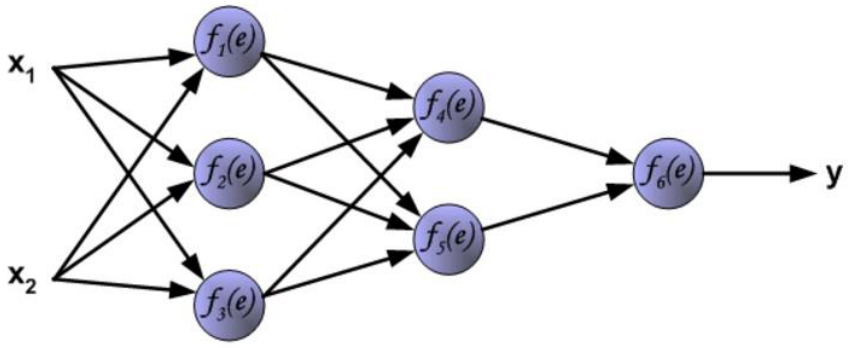


Figura 3-7: Ejemplo de una red.

Cada neurona se compone de dos unidades. La primera unidad agrega productos de coeficientes de pesos y señales de entrada. La segunda unidad realiza la función no lineal, llamada función de activación neuronal. La señal *e* es la señal de salida del sumador, y $y = f(e)$ es la señal de salida del elemento no lineal. La señal *y* es también señal de salida de la neurona.

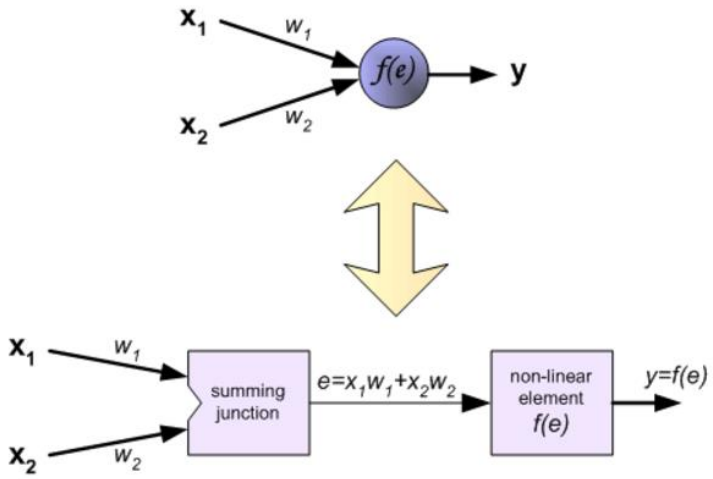
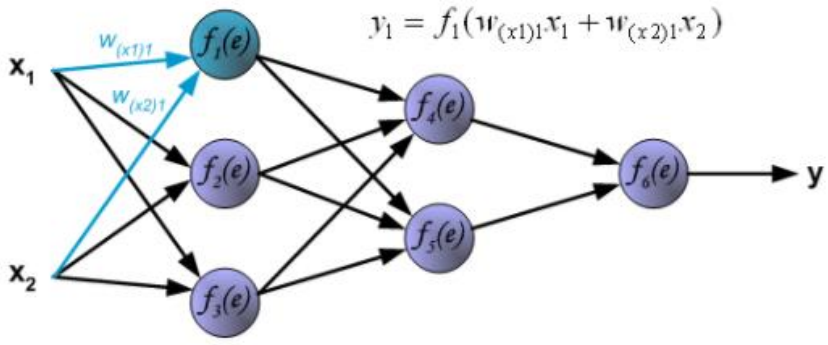
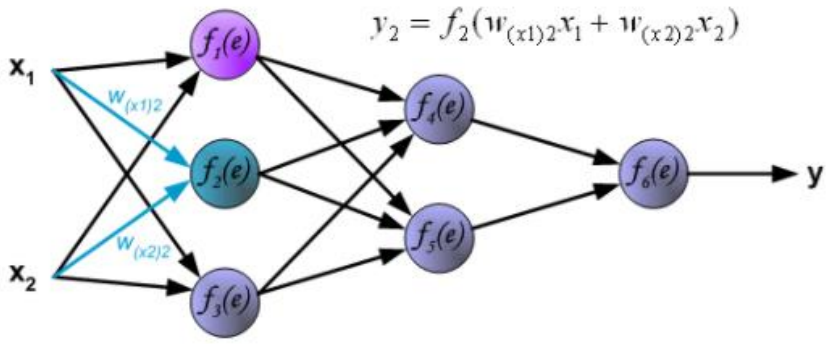


Figura 3-8: Funcionamiento de una neurona.

Para enseñar a la red neuronal necesitamos un conjunto de datos de entrenamiento. El conjunto de datos de entrenamiento consta de señales de entrada (x_1 y x_2) asignadas con el objetivo correspondiente (salida deseada) z. El entrenamiento de la red es un proceso iterativo. En cada iteración los coeficientes de pesos de los con forzar ambas señales de entrada del conjunto de entrenamiento. Después de esta etapa se puede determinar valores de señales de salida para cada neurona en cada capa de la red. Las imágenes de abajo ilustran cómo se está propagando la señal a través de la red. Los símbolos $w_{(xm)n}$ representan pesos de conexiones entre la entrada de red x_m y la neurona n en la capa de entrada. Los símbolos y_n representan la señal de salida de la neurona n.





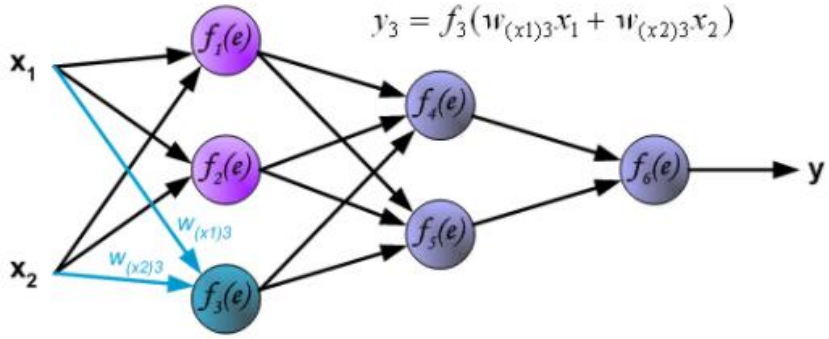


Figura 3-9: Ejemplo de propagación en una red.

En la propagación de señales a través de la capa oculta. Los símbolos w_{mn} representan pesos de conexiones entre la salida de la neurona m y la entrada de la neurona n en la siguiente capa.

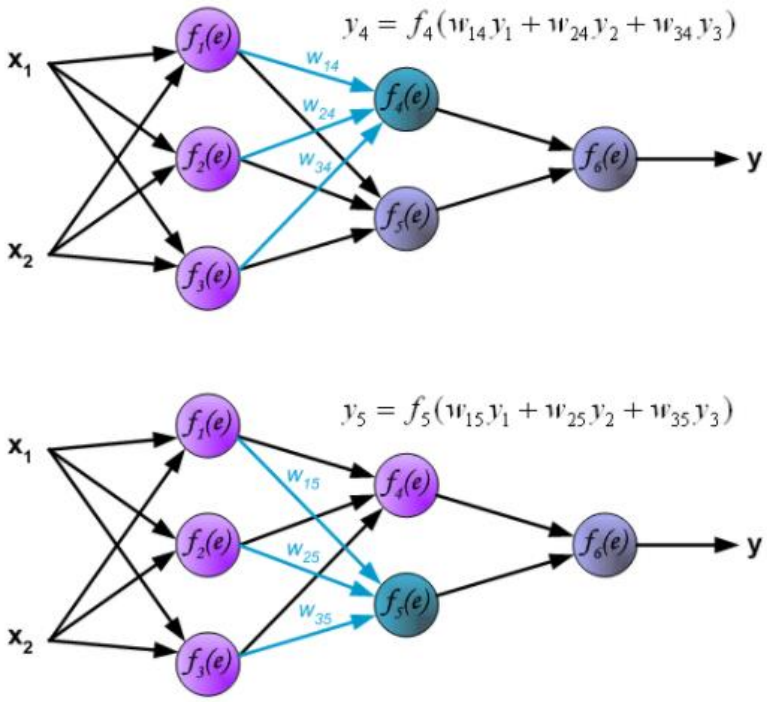


Figura 3-10: Ejemplo de propagación en una red en las capas ocultas.

En la propagación de señales a través de la capa de salida.

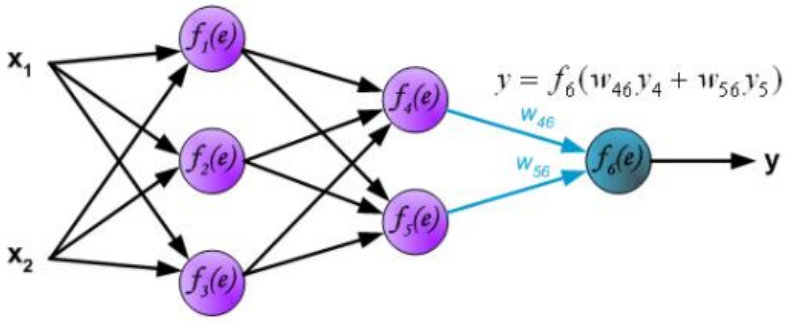


Figura 3-11: Ejemplo de propagación a la salida.

En el siguiente paso del algoritmo, la señal de salida de la red y se compara con el valor de salida deseado (el objetivo), que se encuentra en el conjunto de datos de entrenamiento. La diferencia se denomina señal de error δ de la neurona de la capa de salida.

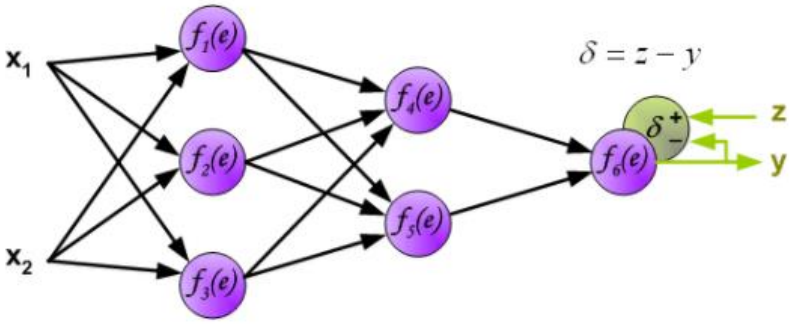


Figura 3-12: Cálculo del error a la salida de la red.

Es imposible calcular la señal de error para las neuronas ocultas directamente, porque los valores de salida de estas neuronas son desconocidos. Durante muchos años el método efectivo para entrenar redes multicapa ha sido desconocido. Sólo a mediados de los ochenta se ha elaborado el algoritmo de retropropagación. La idea es propagar la señal de error δ (calculada en un solo paso de enseñanza) de vuelta a todas las neuronas.

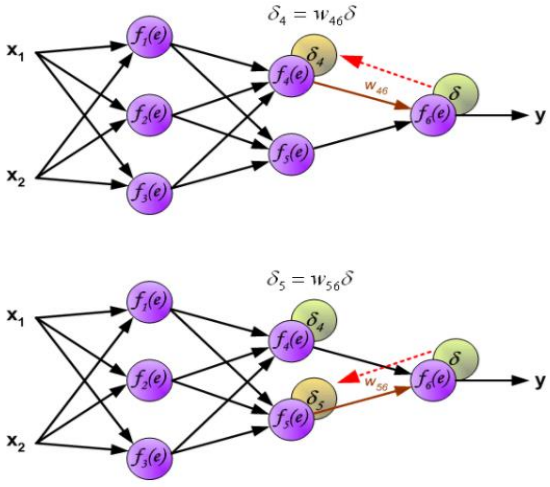


Figura 3-13: Ejemplo de retropropagación (I).

Los coeficientes de ponderación w_{mn} utilizados para propagar los errores de regreso, son iguales a los utilizados durante el cálculo del valor de salida. Sólo se cambia la dirección del flujo de datos (las señales se propagan desde la salida a las entradas una tras otra). Esta técnica se utiliza para todas las capas de red. Si los errores propagados proceden de pocas neuronas, se añaden. La ilustración está abajo:

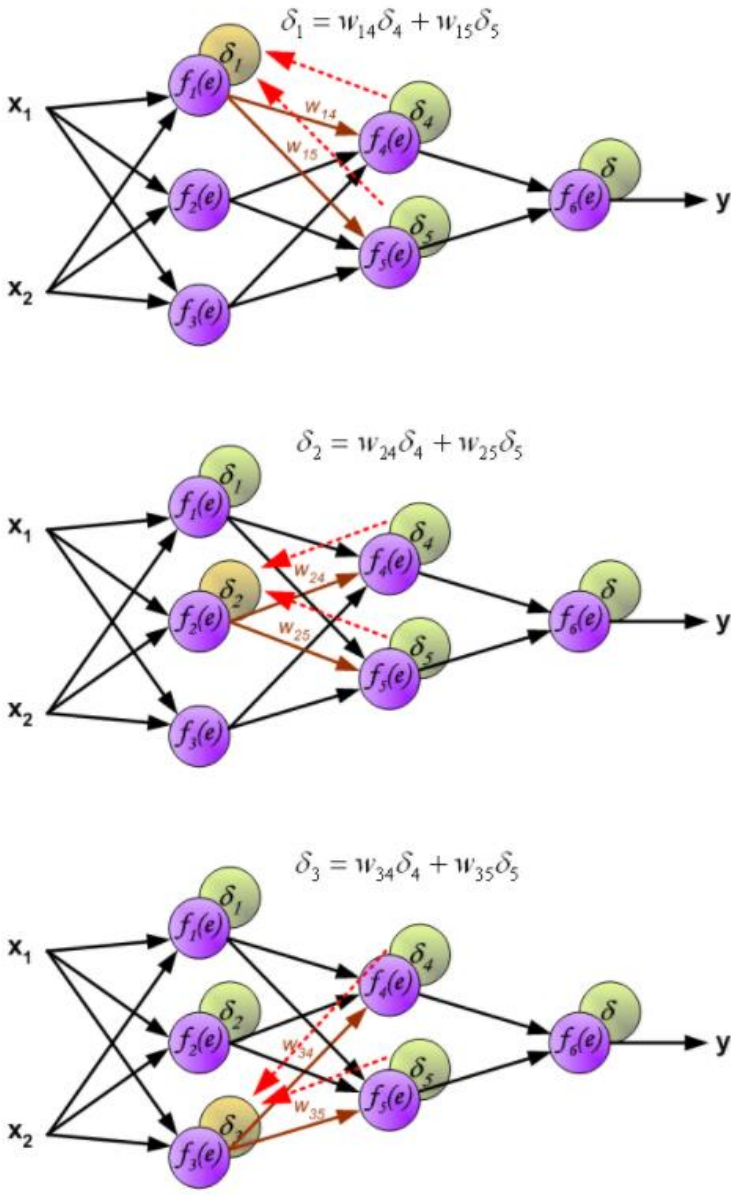
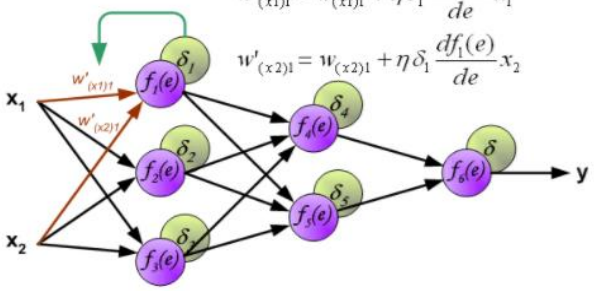


Figura 3-14: Ejemplo de retropropagación (II).

Cuando se calcula la señal de error para cada neurona, los coeficientes de ponderación de cada nodo de entrada de neurona pueden ser modificados. En las fórmulas a continuación $\frac{df(e)}{de}$ representa la derivada de la función de activación neuronal (los pesos que se modifican).

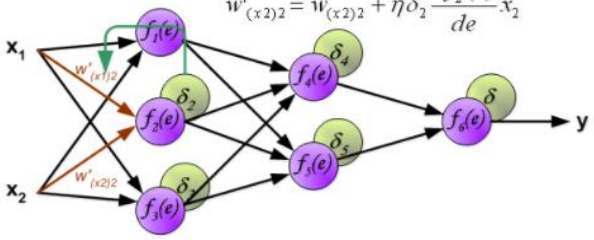
$$w'_{(x1)1} = w_{(x1)1} + \eta\delta_1 \frac{df_1(e)}{de} x_1$$

$$w'_{(x2)1} = w_{(x2)1} + \eta\delta_1 \frac{df_1(e)}{de} x_2$$



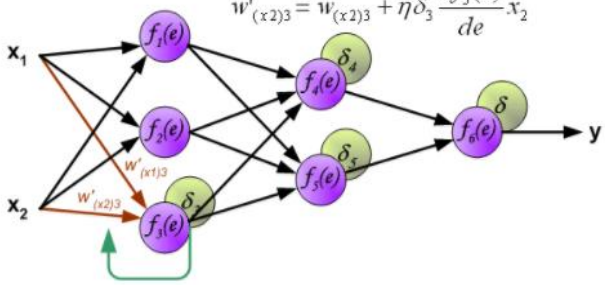
$$w'_{(x1)2} = w_{(x1)2} + \eta\delta_2 \frac{df_2(e)}{de} x_1$$

$$w'_{(x2)2} = w_{(x2)2} + \eta\delta_2 \frac{df_2(e)}{de} x_2$$



$$w'_{(x1)3} = w_{(x1)3} + \eta\delta_3 \frac{df_3(e)}{de} x_1$$

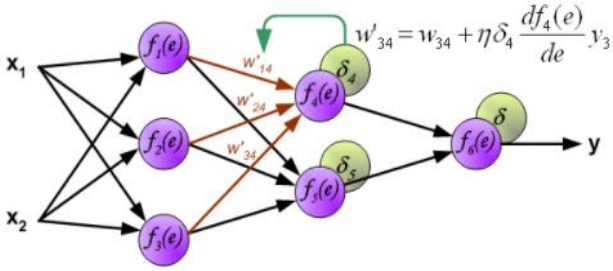
$$w'_{(x2)3} = w_{(x2)3} + \eta\delta_3 \frac{df_3(e)}{de} x_2$$



$$w'_{14} = w_{14} + \eta\delta_4 \frac{df_4(e)}{de} y_1$$

$$w'_{24} = w_{24} + \eta\delta_4 \frac{df_4(e)}{de} y_2$$

$$w'_{34} = w_{34} + \eta\delta_4 \frac{df_4(e)}{de} y_3$$



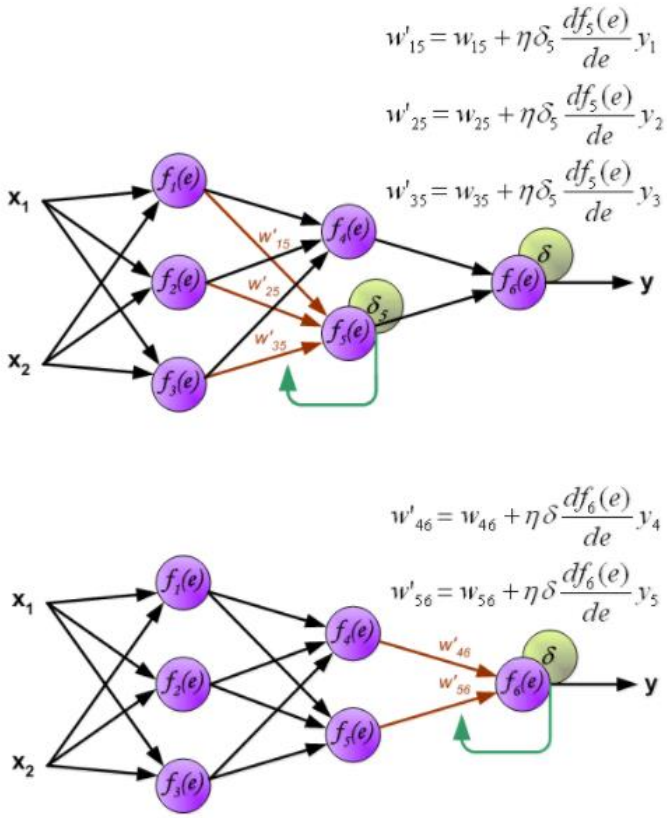


Figura 3-15: Ejemplo de retropropagación (III).

El coeficiente η afecta a la velocidad de enseñanza de la red. Hay algunas técnicas para seleccionar este parámetro. El primer método es iniciar el proceso de enseñanza con gran valor del parámetro. Mientras se establecen los coeficientes de ponderación, el parámetro se reduce gradualmente. El segundo método, más complicado, empieza a enseñar con un pequeño valor de parámetro. Durante el proceso de enseñanza el parámetro se incrementa cuando la enseñanza es avanzada y luego disminuye de nuevo en la etapa final. Iniciar el proceso de enseñanza con bajo valor de parámetro permite determinar signos de coeficientes de ponderación.

4 PROBLEMA DE OVERFITTING

En el aprendizaje automático, una de las tareas más comunes es ajustar un modelo a un conjunto de datos de entrenamiento, de modo que se puedan hacer predicciones fiables sobre datos que en general no han sido entrenados.

El overfitting ocurre cuando un modelo es excesivamente complejo, como cuando se tienen demasiados parámetros en relación al número de observaciones. Un modelo que ha sido sobreajustado, tiene un bajo nivel de predicción ya que reacciona exagreadamente a pequeñas fluctuaciones en los datos de entrenamiento.

En la siguiente imagen se representa de forma gráfica el problema de overfitting:

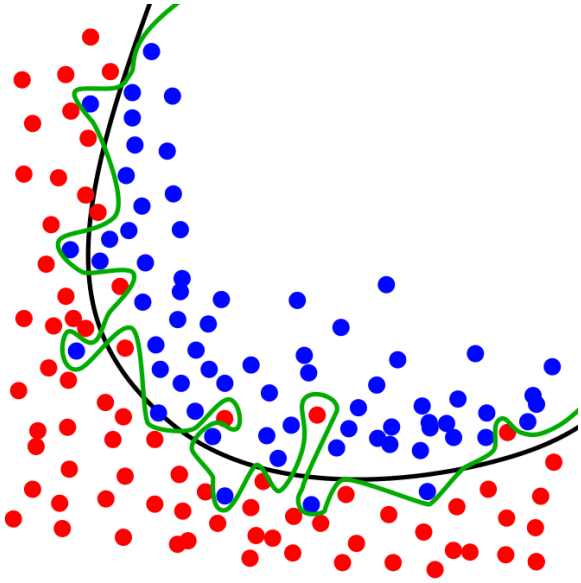


Figura 4-1: Gráfica en la que se representa el problema de overfitting.

La imagen es un gráfico donde se visualizan dos clases de objetos (los puntos de color rojo y los puntos de color azul). Lo que se pretende es dividir las dos clases con una curva para posteriromente clasificar un nuevo punto de entrada. En la imagen hay una línea verde que representa un modelo sobreajustado y la línea negra representa un modelo regularizado. Mientras que a línea verde sigue mejor los datos de entrenamiento, es demasiado dependiente de él y es probable que una tasa de error más alta en nuevos datos no vistos, en comparación con la línea negra.

Por lo tanto overfitting es un problema que hay que evitar y se puede hacer con varios métos, algunos ejemplos son técnicas como validación cruzada, regularización, detención temprana, comparación de modelos o abandono. La base de algunas de las técnicas es penalizar explícitamente modelos excesivamente complejos, o probar la capacidad del modelo para generalizar evaluando su funcionamiento en un conjunto de datos no utilizados para el entrenamiento.

5 RED CONVOLUCIONAL

Las redes convolucionales aparecen con el objetivo de disminuir la carga computacional a la hora de resolver un problema de clasificación. Los anteriores clasificadores eran buenos para trabajar con matrices con pocas características (matriz "x" de tamaño reducido). El problema surge cuando estas matrices llegan a ser muy grandes, por ejemplo, un conjunto de ejemplos de entrenamiento de imágenes con dimensiones grandes. La idea de las redes convolucionales es extraer las características más significativas de una imagen con el objetivo de disminuir las matrices de características.

Las redes convolucionales son un tipo de redes neuronales que han demostrado tener un buen rendimiento en áreas como el reconocimiento y clasificación de imágenes. Las CNNs han tenido éxito en la identificación de caras, objetos y señales de tráfico, aparte, han aumentado la visión en robots y han mejorado el desarrollo de vehículos autónomos.

Unas de las primeras redes convolucionales fue la LeNet desarrollado por Yann LeCun. Esta red se usó principalmente para el reconocimiento de caracteres, como la lectura de códigos postolas, dígitos, etc.

A continuación, se explica cómo la arquitectura LeNet aprende a reconocer las imágenes.

La LeNet tiene la siguiente forma:

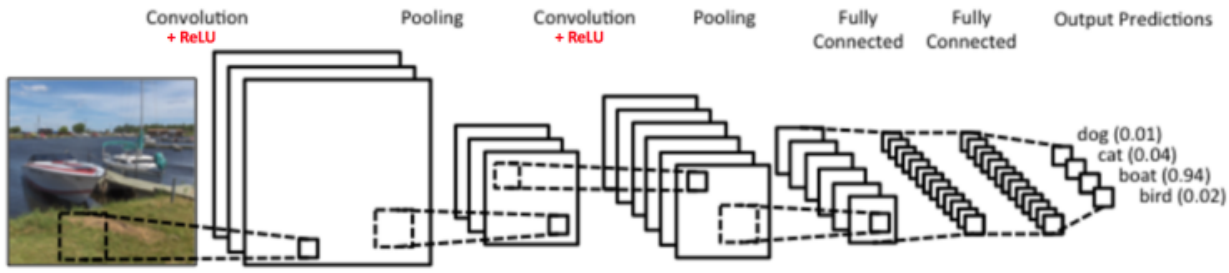


Figura 5-1: Esquema de una red convolucional.

Esta red convolucional clasifica una imagen de entrada en cuatro categorías: perro, gato, barco o pájaro. La red al recibir como imagen de entrada un barco, la red observa la imagen y la clasifica atendiendo a la probabilidad más alta de que la imagen sea una de las cuatro categorías. La suma de todas las probabilidades en la capa de salida deber ser 1.

En la imagen 5-1 se pueden observar diferentes partes de la red: las convoluciones, el pooling y la clasificación (fully connected layer). Cada una de estas partes se explicará posteriormente para una comprensión de las redes convolucionales.

Hay que tener en cuenta que cada imagen, puede ser representada como una matriz con de valores por cada pixel. Estos pixeles serán las entradas de la red.

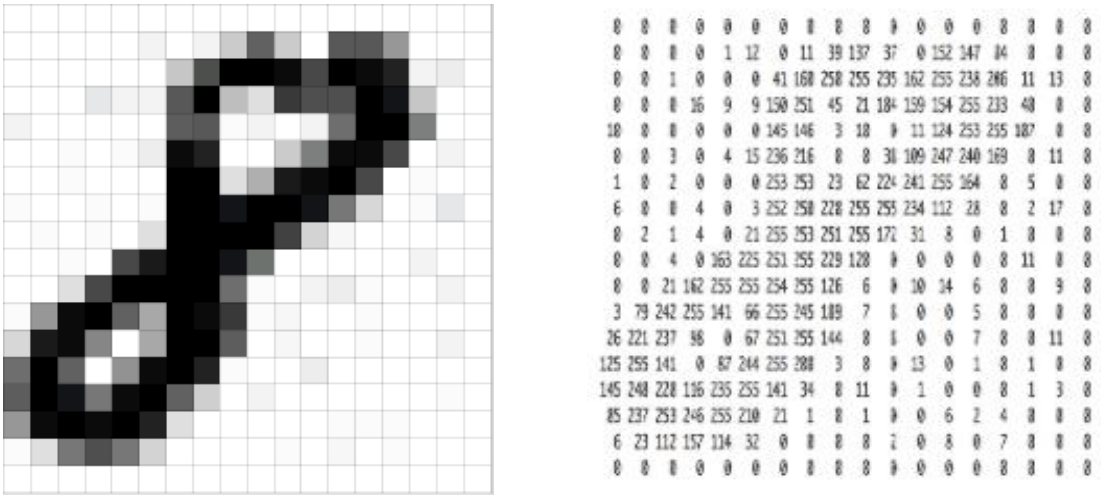


Figura 5-2: Representación de una imagen como una matriz.

5.1 Convolución

El normbre de las redes convolucionales viene de la operación convolución. El principal propósito de una Convolución en las redes convolucionales es el de extraer las características de una imagen de entrada.

Como se dijo anteriormente, cada imagen puede ser considerada como una matriz de números. Se considera una imagen 5x5 cuyos elementos de la matriz solo pueden ser 0 o 1.

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

También se considera otra matriz de 3x3 como la siguiente:

1	0	1
0	1	0
1	0	1

Entonces, la convolucion entre la imagen de 5x5 y la matriz 3x3 puede ser calculada como:

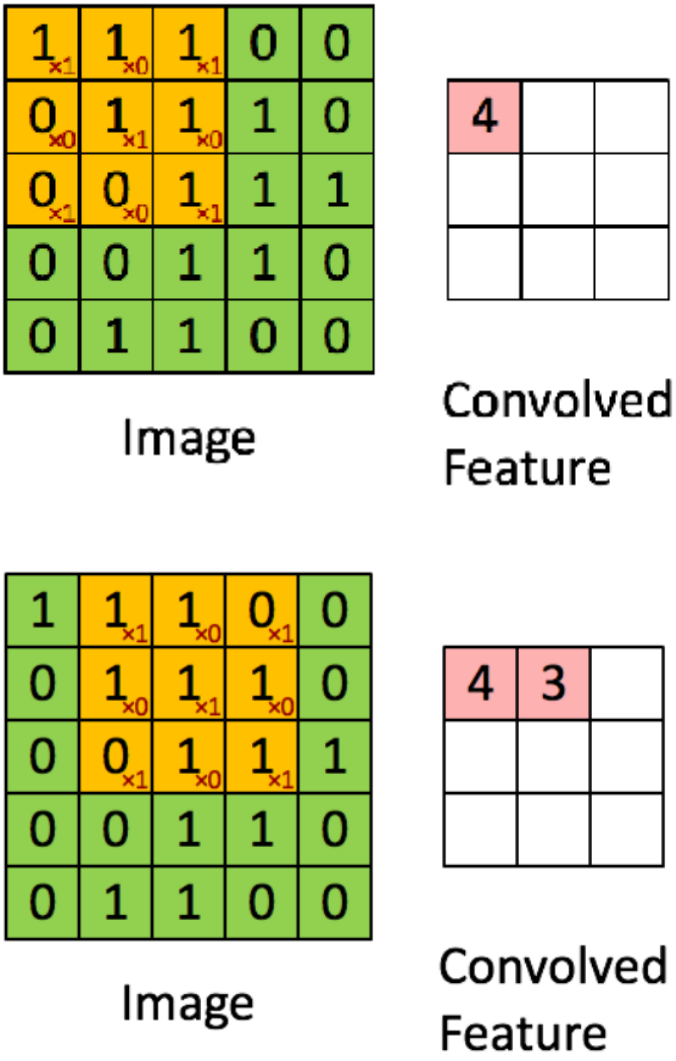


Figura 5-3: Convolución.

La idea de la convolución es desplazar la matriz naranja sobre la imagen original (verde) con un salto de un pixel y para cada posición se calcula la multiplicación entre las dos matrices y se suman las salidas de la multiplicación para obtener un entero (de solor rosa). Hay que tener en cuenta que la matriz 3x3 solo "ve" una parte de la imagen de entrada en cada paso.

En terminología de las redes convoucionales, la matriz 3x3 se conoce como filtro o kernel y la matriz formada al realizar la convolución (rosa) se conoce como "convolved feature" o "feature map". Es importante tener en cuenta que los fitros actúan como detectores de características de la imagen de entrada.

En forma de pseudocódigo, la convolución sería:

```
for each image row in input image:
   for each pixel in image row:

      set accumulator to zero

      for each kernel row in kernel:
```

```
        for each element in kernel row:

            if element position  corresponding* to pixel position then
               multiply element value  corresponding* to pixel value
               add result to accumulator
            endif

    set output image pixel to accumulator
```

Es evidente que se producirán diferentes valores en "feature maps" con diferentes filtros para una misma imagen de entrada. Como ejemplo se considera la siguiente imagen de entrada:

En la siguiente tabal, se observa los efectos de la convolucion de la imagen anterior con diferentes filtros.Como se muestra, podemos realizar operaciones como detección de bordes, nitidez y desenfoque simplemente

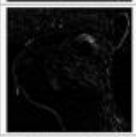
Operation	Filter	Convolved Image
Identity	$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}$	
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

Tabla 5-1: Comparación al aplicar diferentes filtros a una imagen.

cambiando los valores numéricos de nuestra matriz de filtro antes de la operación de convolución, esto significa que diferentes filtros pueden detectar diferentes características de una imagen, por ejemplo bordes, curvas, etc.

En la práctica, una CNN aprende los valores de estos filtros por sí sola durante el proceso de entrenamiento (aunque aún hay que especificar parámetros como el número de filtros, el tamaño del filtro, la aquitectura de la red, etc, antes del proceso de entreanamiento). Cuanto más número de filtros haya, más caraterístias de la imagen se extraen y mejor será la red para reconocer patrones en imágenes que no han sido utilizadas para el entrenamiento.

El tamaño de feauture map se controla por parámetros que se necesitan decidir antes de la convolución

La profundidad que corresponde al número de filtros que se han usado para la operación de la convolución. En la imagen siguiente, se muestra la convolución de una imagen de entrada usando tres filtros diferentes, por lo tanto, produce tres diferentes feature maps. Para este caso la profundidad o número de filtros es 3.

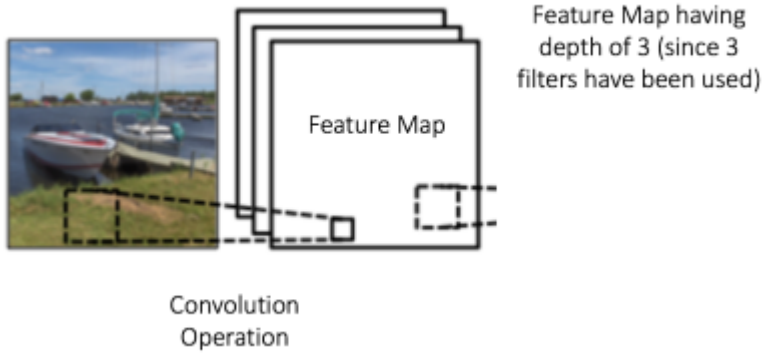


Figura 5-4: Convolución con varios filtros.

5.2 Pooling

Pooling (también llamado subsampling) reduce las dimensiones de cada feature map pero retiene la información más importante. El pooling puede ser de diferentes tipos: máximo, media, suma, etc.

En el caso del max pooling, se define un vecindario (por ejemplo, una ventana de 2x2) y toma el elemento más grande de feature maps dentro de esta ventana. En lugar de tomar el elemento más grande también puede tomarse el promedio (poolinf media) o suma de todos los elementos de la venta. En la práctica, se ha demostrado que max pooling funciona mejor.

La siguiente imagen presenta un ejemplo de max pooling en un fearue map con una ventana de 2x2.

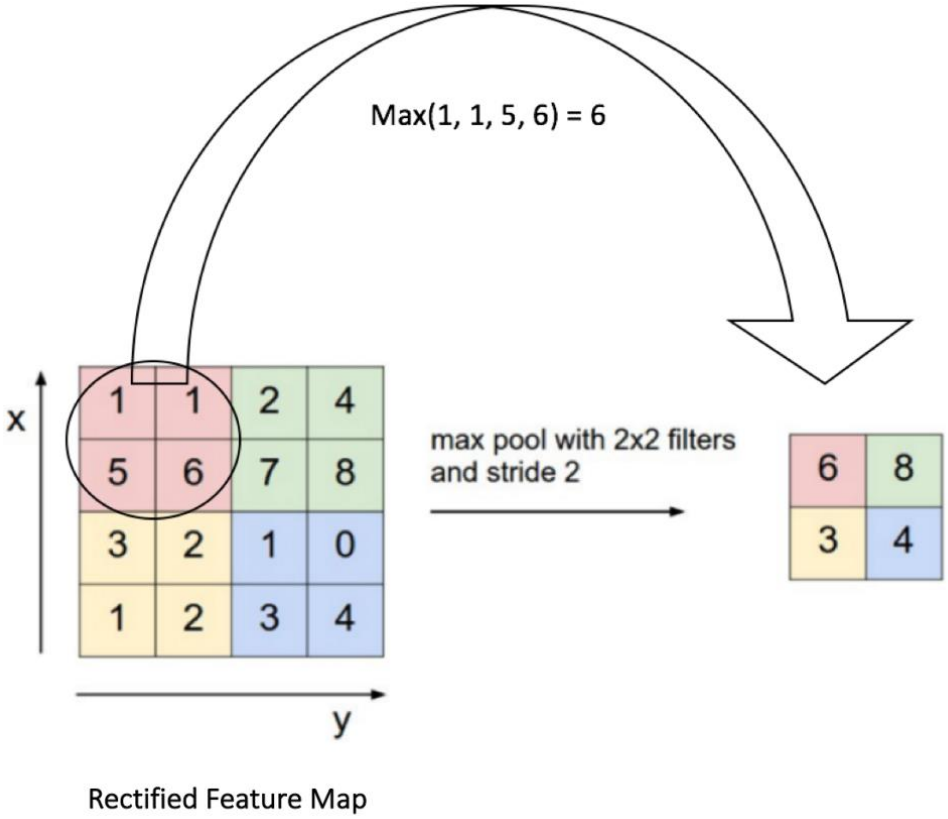


Figura 5-5: Ejemplo de max pool aplicado a una matrix.

Se deslica la ventana de 2x2 en 2 celdas y se toma el valor máximo en cada región. Esto reduce la dimensión del feature map.

El la red que se muestra a continuación, el pooling se aplica de forma separada para cada feature map. Por lo tanto, debido a esto obtenemos tres mapas de salida de tres mapas de entrada.

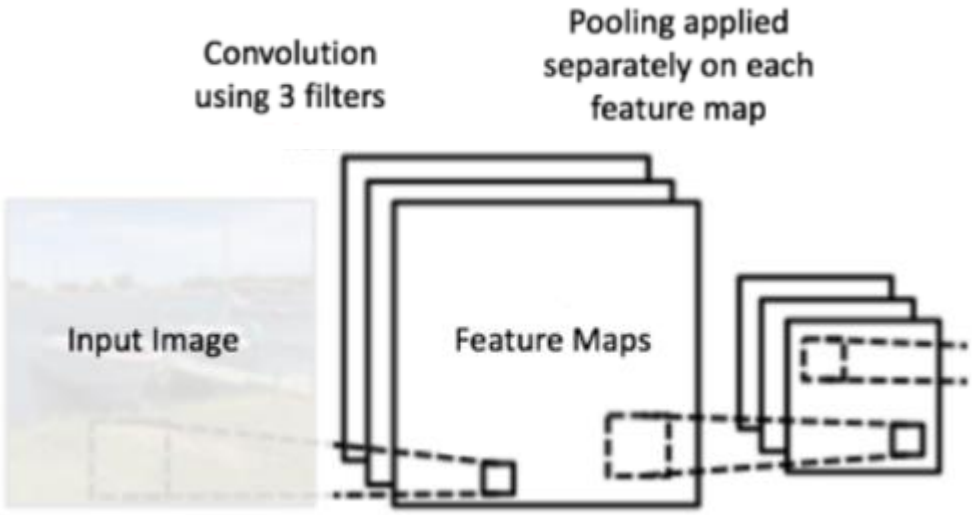


Figura 5-6: Ejemplo de pooling a varias feature maps.

La función de pooling es reducir progresivamente el tamaño espacial de la representación de entrada. En particular, pooling convierte las representaciones de entrada (dimensiones de las características) más pequeñas y manejables, reduce el número de parámetros y cálculos en la red, por lo tanto, controla el overfitting, hace la red invariante a pequeñas transformaciones, distorsiones y translaciones en la imagen de entrada (una pequeña distorsión en la imagen de entrada no cambiará la salida del pooling ya que toma el valor máximo/media en una región), ayuda a llegar a una representación casi invariante a escala de nuestra imagen. Esto es muy útil ya que se puede detectar objetos en una imagen sin importar dónde se encuentren.

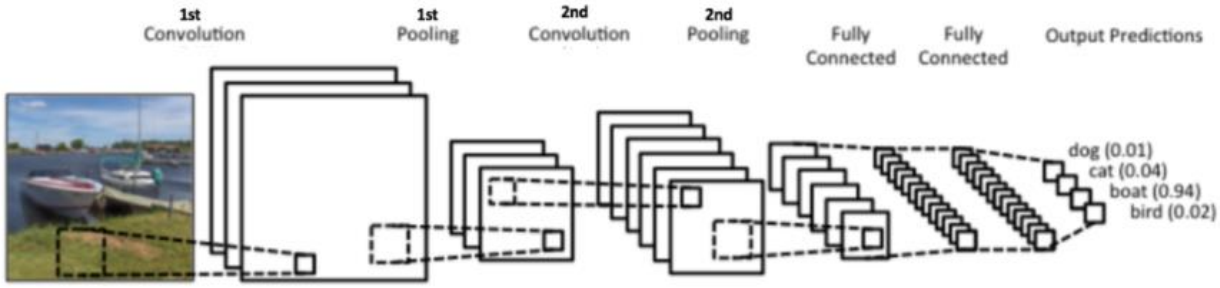


Figura 5-7: Esquema de la red convolucional y todas sus capas.

Hasta ahora se ha visto como funciona la convolucion y el pooling. Es importante entender que estas capas son los componentes básicos de cualquier red convolucional. Como se muesta en la figura anterior, hay dos capas de convolucion y pooling. La segunda capa de convolución realiza la convolución en la salida de la primera capa de pooling usando 6 filtros que producen un total de seis feature maps. A continuación, se realiza la operación max pooling por separado en cada uno de los seis feature maps.

Juntas, estas capas extraen las características útiles de las imágenes, introducen la no-linealidad en la red y reducen la dimensión de las matrices de características.

La salida de la segunda capa de pooling actúa como entrada a la capa de fully connected que se explicará a continuación.

5.3 Stochastic Gradient Descent (SGD)

Algunos métodos de optimición como el LBFGS, usan el conjunto de entrenamiento completo para actualizar los parámetros en cada iteración y tiende a converger de manera eficiente hacia un mínimo absoluto. Sin embargo, en la prácctica el cálculo del coste y del gradient para todo el conjunto de entrenamiento puede resultar muy lento e incluso imposible para una sola máquina si el conjunto de datos es demasiado grande para la memoria principal. Además otro problema con estos métodos de optimización es que no hay forma fácil de incorporar nuevos datos "en línea". El método de Stochastic Gradient Descent resuelve estos dos problemas siguiendo la dirección negativo del gradiente de la función objetivo tras haber visto solo uno o unos pocos ejemplos de entrenamiento. El alto coste de retropropagar el conjunto complete de entrenamiento hace necesario el uso del SGD.

El algoritmo estándar del gradient descent actualize los parámetros $\boldsymbol{\theta}$ de forma:

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \alpha \nabla_{\theta} E[J(\boldsymbol{\theta})] \quad (5\text{-}1)$$

En la ecuación anterior, la esperanza se aproxima por el gradient de la función coste usando el conjunto complete de entrenamiento. El método del SGD simplement elimina dicha esperanza y calcula el gradient usando solo unos pocos ejemplos de entrenamiento. La nueva actualización viene ahora dada por:

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \alpha \nabla_{\theta} J\left(\boldsymbol{\theta}; x^{(i)}, y^{(i)}\right) \quad (5\text{-}2)$$

El par $\left(x^{(i)}, y^{(i)}\right)$ es un par del conjunto de entrenamiento.

Generalmente, cada actualización en el SGD se calcula con vista a unos pocos ejemplos del set de entrenamiento (llamados minibatch) en lugar de solo a un ejemplo. La razón tras esto es doble: en primer lugar, se reduce la varianza en la actualización de los parámetros y esto permite una convergencia más estable, y, en segundo lugar, permite el uso de álgebra lineal para el eficiente cálculo del coste y del gradiente. El tamaño típico de un minibatch es de 256, aunque el tamaño óptimo puede variar según la aplicación y la arquitectura.

En el SGD el ratio de aprendizaje α normalmente es mucho más pequeño que el usado en el método del gradient descent pues la varianza en la actualización es menor. Escoger el valor adecuado para el ratio de aprendizaje y sus actualizaciones a lo largo del entrenamiento puede ser realmente complicado. Un método estándar que funciona bien en la práctica es usar un ratio pequeño y constante que proporcione una convergencia estable en la epoch (uso completo del set de entrenamiento mediante minibatches) inicial y reducir a la mitad su valor a medida que la convergencia se ralentiza. Otro método bastante utilizado es actualizar el ratio de aprendizaje en cada iteración t como $a/(b + t)$, donde a es el ratio inicial y b la iteración en la que comienta dicha actualización. Los métodos más sofisticados incluyen el uso de backtracking (vuelta atrás) a lo largo de una línea para encontrar la actualización óptima.

Por último, pero no menos importante, en el SGD importa el orden en el que los datos son presentados al algoritmo. Si los datos son presentados en un orden cualquiera, ésto puede introducir un offset en el gradiente y conducir a una convergencia más pobre. Un buen método para evitar esto es, por lo general, "barajar" de forma aleatoria los datos antes de cada epoch de entrenamiento.

Si la función coste tiene la forma análoga a un largo desfiladero poco profundo que conduce al mínimo y con paredes empinadas a los lados, el SGD tenderá a oscilar a lo ancho del estrecho desfiladero, pues la dirección negativa al gradiente apuntará a alguna de las paredes en lugar de al mínimo. Esta forma, que puede parecer muy poco probable, es bastante común en arquitecturas más complejas y, por tanto, la convergencia puede llegar a ser muy lenta. El método Momentum es más óptimo en este caso pues apunta al mínimo más rápido a lo largo del desfiladero. La actualización de los parámetros con este método es:

$$\begin{aligned} \boldsymbol{v} &= \gamma \boldsymbol{v} - \alpha \nabla_{\theta} J\left(\boldsymbol{\theta}; x^{(i)}, y^{(i)}\right) \\ \boldsymbol{\theta} &= \boldsymbol{\theta} - \boldsymbol{v} \end{aligned} \qquad (5\text{-}3)$$

En la ecuación anterior $\boldsymbol{v}$ es el vector velocidad actual, cuyas dimensiones coinciden con las de $\boldsymbol{\theta}$. El ratio de aprendizaje α se toma como se describió previamente, aunque, al usar este método, α tal vez deba ser más pequeño dado que la magnitud del gradiente será mayor. Por último, $\gamma \in (0,1]$ determina por cuantas iteraciones se incorporarán los gradientes anteriores. Generalmente, γ se establece a 0.5 hasta que el aprendizaje se estabiliza, momento en el que se aumenta a 0.9.

5.4 Capa Fully Connected

La capa fully connected es una red multicapa como la explicada anteriromente en el punto 3, que usa una función softmax en la salida de la capa. El término "fully connected" implica que cada neurona de una capa se conecta con todas las neuronas de la capa siguiente.

La salida de las capas convolucionales y de pooling representan características de alto nivel de la imagen de entrada. El propósito de la capa fully connected es utilizar estas características para clasificar la imagen de entrada en varias clases basadas en el dataset de entrenamiento. Por ejemplo, la tarea de clasificación de imágenes que se ha establecido como ejemplo a realizar tiene cuatro salidas posibles como se muestra en la

siguiente imagen.

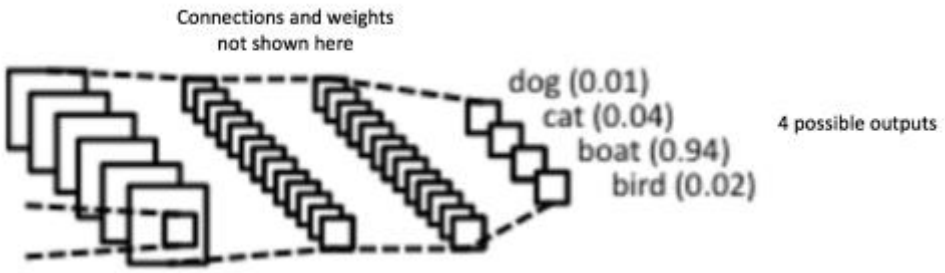


Figura 5-8: Ejemplo de una capa fully connected.

Aparte de la clasificación, la adición de una capa fully connected es también una manera barata de aprender combinaciones no lineales de estas características. La mayoría de las características de las capas convolucionales y de las capas de poooling pueden ser buenas para la tarea de clasificación, pero las combinaciones de esas características podrían ser aún mejor.

La suma de las probabilidades de la capa fully connected es 1. Esto se asegura usando un softmax como función de activación en la capa de salida de la capa fully connected. La función softmax toma un vector con valores reales arbitrarios y lo transforma en un vector cuya suma de componentes es 1.

5.5 Entrenamiento: Backpropagation.

Como se explicaba anteriormente, las capas de convolución y de pooling actúan como extractores de características de la imagen de entrada mientras que la capa fully connected actúa como un clasificador.

Observesé en la siguiente imagen, dado que la imagen de entrada es un barco, la probabilidad objetico es 1 para la clase barco y 0 para el resto de las demás clases.

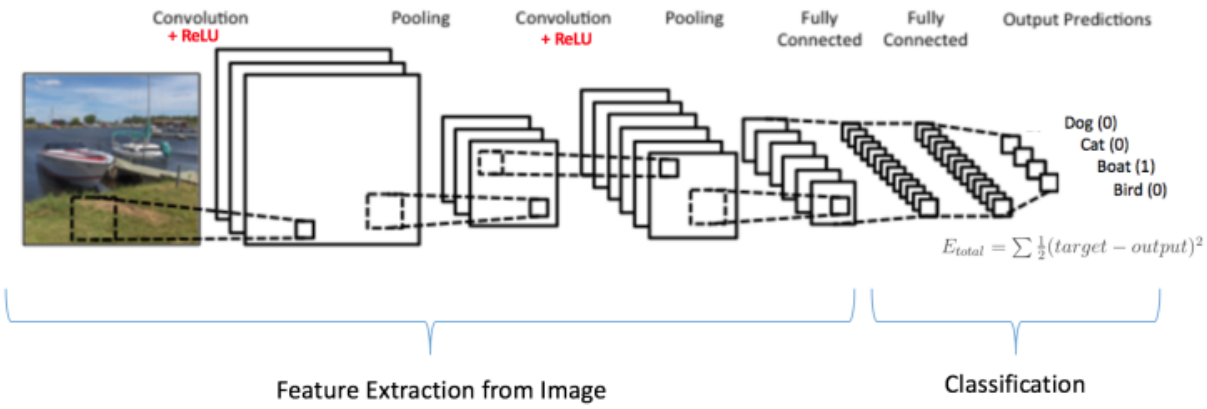


Figura 5-9: Esquema de las partes de una red convolucional.

Todo el proceso de entrenamiento de la red concolucional puede resumirse en los cinco pasos.

El primer paso, se inicializan los filtros y parámetros pesos con valores aleatorios. El segundo paso, la red toma una imagen de entrenamiento como entrada, pasa por el segundo paso, propagación hacia adelante, (convolución y operaciones de pooling junto con la propagación hacia adeante en la capa fully connected) y encuentra las probabilidades de salida para cada clase. En el tercer paso, se calcula el error total en la salida de la red (suma

de las 4 clases)

$$Error_{total} = \sum \frac{1}{2}\left(probabilidad_{objetivo} - probabilidad_{salida}\right)^2 \quad (5\text{-}4)$$

El cuarto paso usa backpropagation para calcular el gradiente del error con respecto a los pesos en la red y usa el gradient descent para actualizar todos los valores de los filtros y pesos y parámetros para minimizar el error de la salida. El quinto paso repite los pasos del 2 al 4 con todas las imágenes en el conjunto de entrenamiento.

Los pasos anteriores entrenan a la CNN, esto significa que todos los pesos y parámetros de la CNN han sido optimizados para clasificar correctamente las imágenes del conjunto de entrenamiento.

Cuando una nueva imagen (nunca antes vista) entra a la red convolucional, la red pasaría por el paso de propagación hacia adelante y emitirá una probabilidad para cada clase (para una nueva imagen, las probabilidades de salida se calculan usando los pesos que han sido optimizados para clasificar correctamente todos los ejemplis de entrenamiento anteriores). Si el conjunto de entrenamiento es lo suficientemente grande para que la red puede clasificar de forma correcta las imágenes de entrada.

6 HARDWARE

El proyecto se ha realizado con un ordenador portátil Lenovo Z50-70 que dispone de un procesador Core i7-4510U, velocidad del procesador de 2 GHz, dos procesadores, una memoria RAM con capacidad de 16 GB y tecnología de la memoria DDR3-SDRAM.

Figura 6-1: Ordenador empleado para la realización del proyecto.

7 SOFTWARE

Para realizar el proyecto se ha hecho uso de un sistema operativo Windows 10, Microsoft Visual Studio 2017 y lalibrería OpenCV.

7.1 Microsoft Visual Studio 2017

Se trata de un entorno de desarrollo integrado para sistemas operativos Windows. Soporta múltiples lenguajes de programación, entre ellos C++ que es el lenguaje en el que se basa el proyecto.

A continuación se explica detalladamente cómo instalar Visual Studio 2017:

1. Descargar Visual Studio 2017 con la edición Visual Studio Community. A continuación, ejecute el archivo de programa **vs_community.exe**.
2. Aceptar los términos de icencia y la declaración de privcidad. Haga clic en Instalar para continuar.
3. Una vez que se termine la instalación del instalador, se seleccionarán las caragr de trabajo. Para este proyecto basta con las recuadradas en rojo en la siguiente imagen:

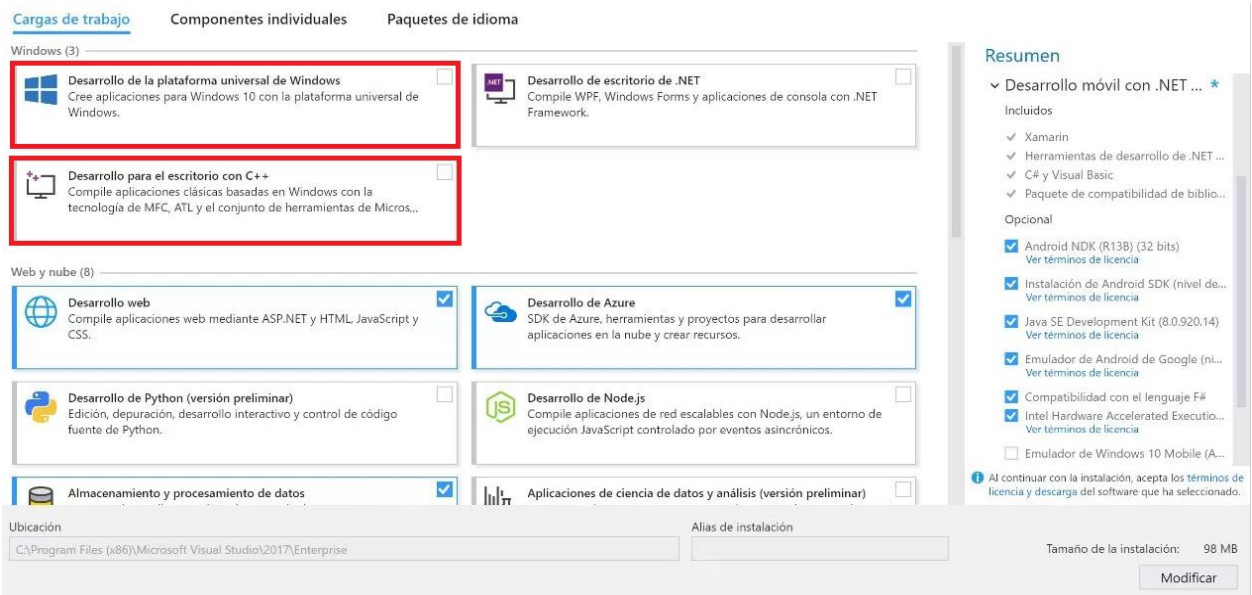


Figura 7-1: Instalación de los programas y librerías (I).

4. Después de seleccionar las cargas de trabajo, haga clic en Instalar. Una vez instalados los nuevos componentes y cargas de trabajo, haga clicl en Iniciar.
5. Finalmente, se instala el idioma que se prefiera.

7.2 OpenCV

OpenCV es una biblioteca libre de vision artificial que se utiliza para reconocimiento de objetos, calibración de cámaras, visión estérea, visión robótica, etc. Simplifica las operaciones matemáticas como la multiplicación de matrices.

A continuación se explica cada uno de los pasos para instalar OpenCV en Microsoft Visual Studio 2017:

1. Descargar OpenCV desde la página oficial y ejecutamos el **.exe**.
2. Abrir Visual Studio 2017 y crear un Nuevo Proyecto:

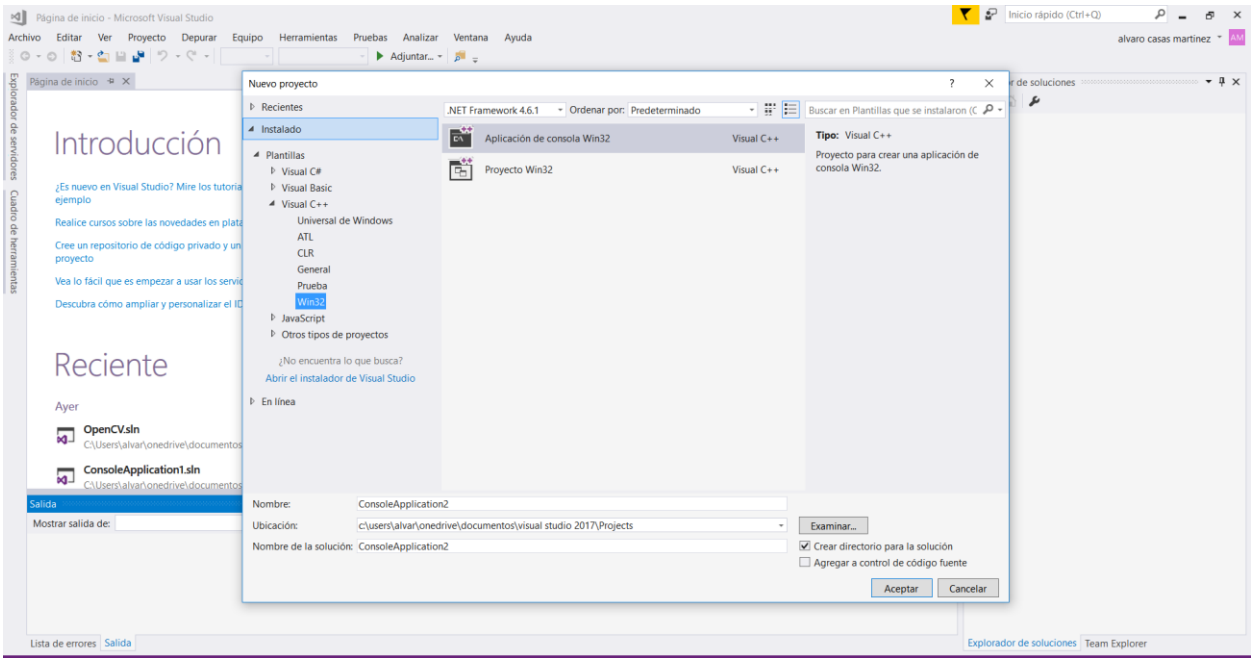


Figura 7-2: Instalación de los programas y librerías (II).

3. En la ventana Ver → Otras ventanas → Administrador de propiedades.

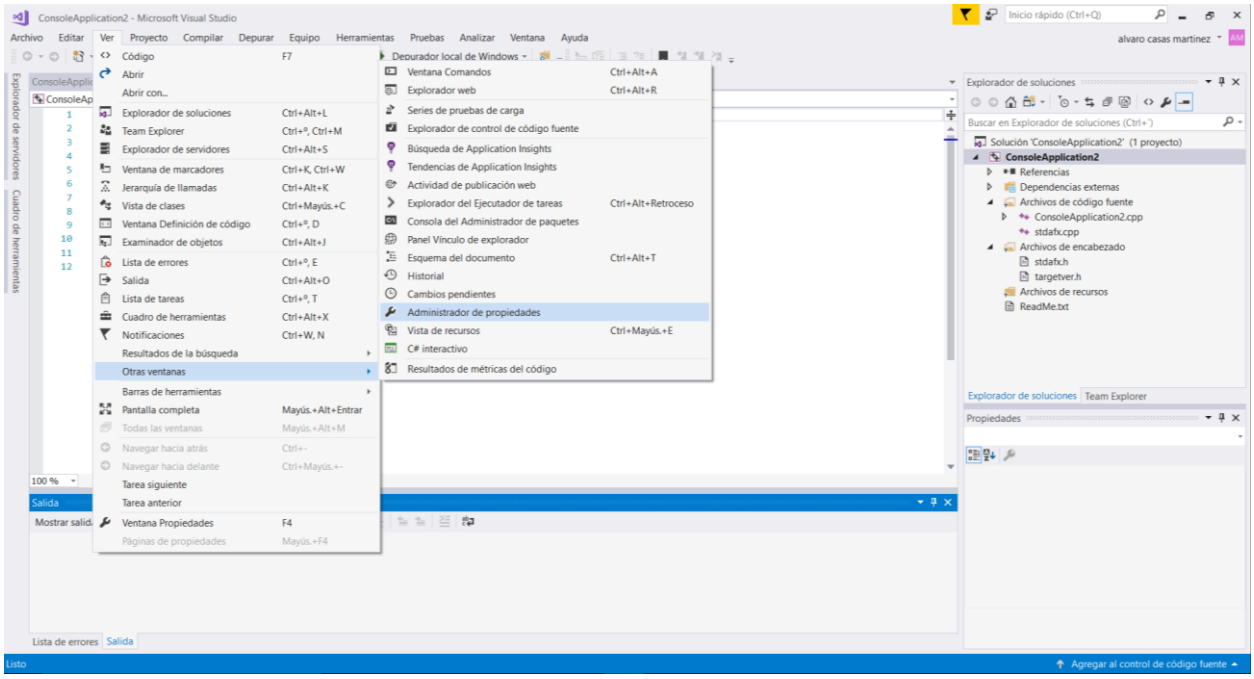


Figura 7-3: Instalación de los programas y librerías (III).

4. En la ventana Administrador de propiedades, agregar una nueva hoja de propiedades de proyecto a Debug | x64:

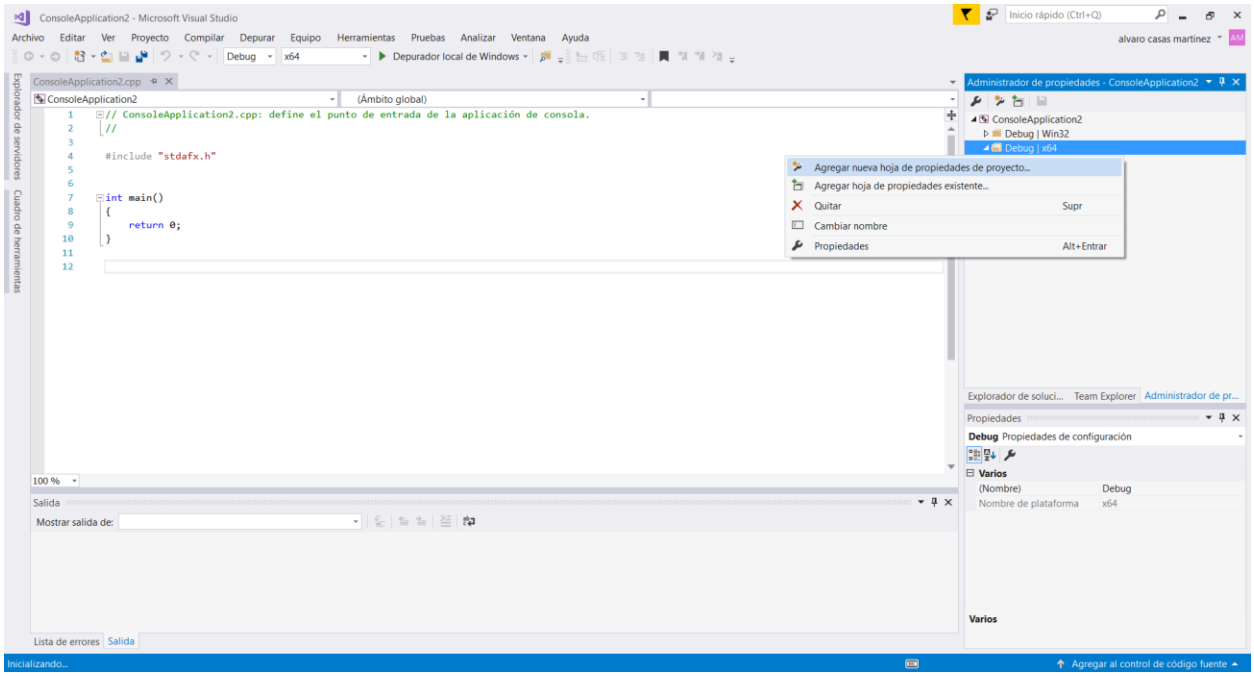


Figura 7-4: Instalación de los programas y librerías (IV).

5. Abrir la nueva hoja y dentro de la pestaña Directorios de VC++, se añade a Directorio de archivos ejecutables la ruta **opencv\build\x64\vc14\bin** y a Directorios de biblioteca la ruta **opencv\build\x64\vc14\lib**.

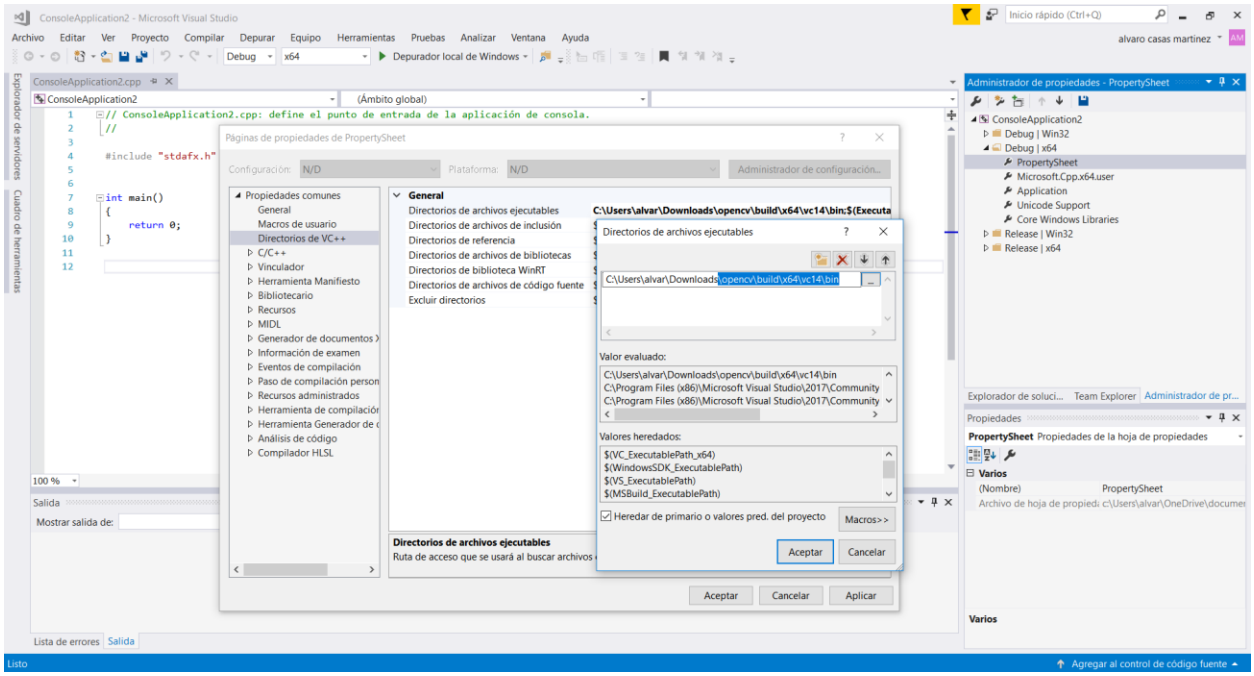


Figura 7-5: Instalación de los programas y librerías (V).

6. Dentro de la pestaña C/C++ se añade a Directorios de inclusión adicionales la ruta

opencv\build\include.

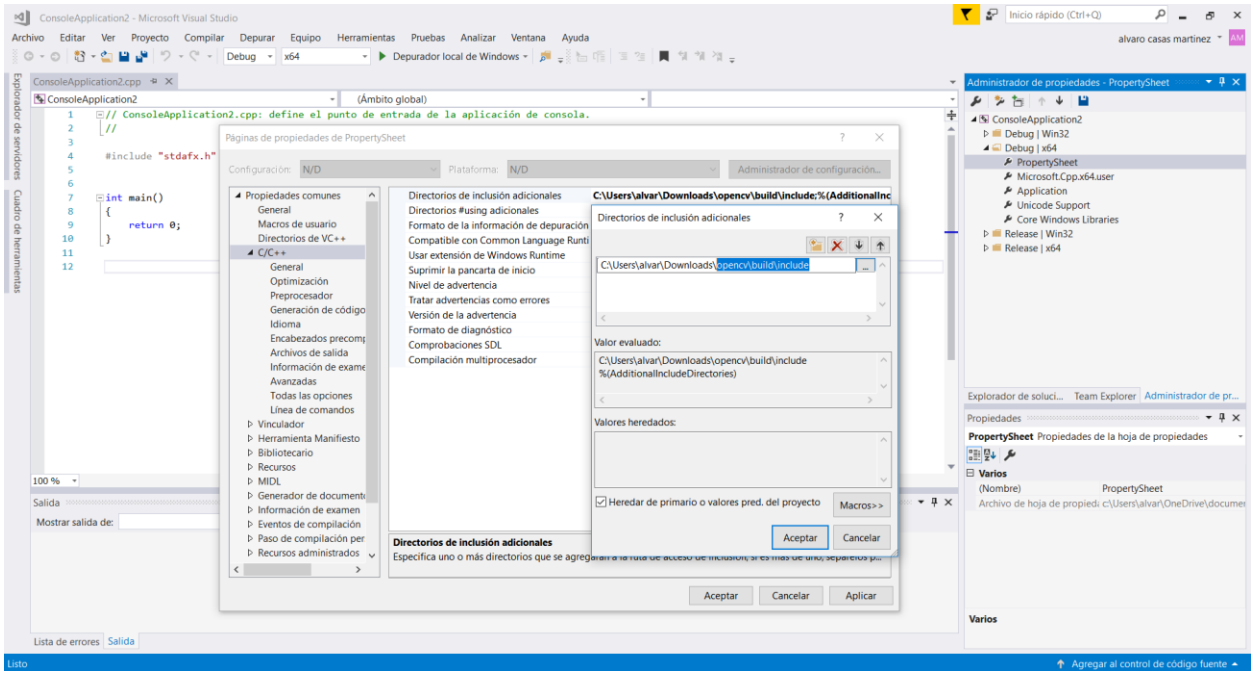


Figura 7-6: Instalación de los programas y librerías (VI).

7. Dentro de la pestaña Vinculador→General, añadimos a Directorios de biblioteca adicionales, la ruta **opencv\build\x64\vc14\lib**.

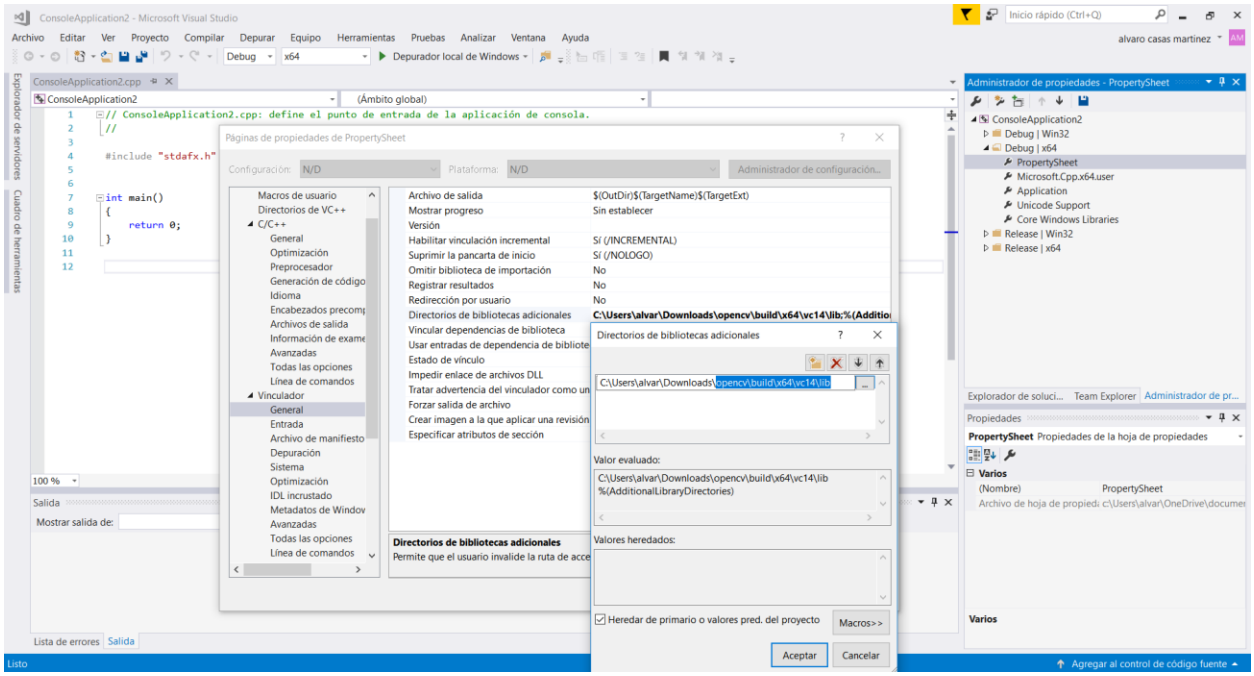


Figura 7-7: Instalación de los programas y librerías (VII).

8. Repetir los pasos desde el punto 4, pero en vez de usar la pestaña Debug | x64, se usa la pestaña Release | x64.

8 PARTES DEL CÓGIGO

El código se compone de varias funciones:

- Función main: Esta función sirve para leer las imágenes y labels del conjunto de entrenamiento, permite entrenar a la red y ver la precisión con la que la CNN es capaz de reconocer imágenes de números.
- Función read_Mnist: Lee las imágenes del MNIST y las guarda en un vector de matrices.
- Función read_Mnist_Label: Lee las etiquetas del MNIST y las guarda en una matriz.
- Función sigmoid: Se trata la función sigmoidea.
- Función rot90: Rota a la matriz 90*k grados, donde k es un entero que entra en la función.
- Función conv2: Realiza la convolución de dos matrices.
- Función cnnInitParams: Inizializa los pesos y bias de la red.
- Función cnnConvolve: Capa de convolución de la red.
- Función cnnPool: Capa de pooling de la red.
- Función cnnParamsToStack: Transforma los parámtros de la red en forma vector a pesos y bias.
- Función cnnCost: Calcula la probabilidad de pertenecer a cada clase, el coste y el gradiente.
- Función minFunSGD: Entrena la red convolucional utilizando el algoritmo SGD con Momentum.

9 RESULTADOS Y EXPERIMENTOS

Al ejecutar el código y tras 702 iteraciones de reducción del coste y calculo del valor de los pesos y bias de la red, la red está entrenada. Para comprobar su funcionamiento se hace uso de un conjunto diferente (con diferentes imágenes que las del entrenamiento). Este nuevo conjunto de imágenes entra en la red, las entradas se multiplican con los parámetros calculados y se obtienen las salidas (los números que la red predice). Con este código se ha logrado un porcentaje de acierto de un 97.08% con un conjunto de 10000 imágenes diferentes.

El tiempo necesario para entrenar la red con 60000 imágenes fue de una hora y treinta minutos y el tiempo de testeo fue de tres horas con el set de 10000 imágenes.

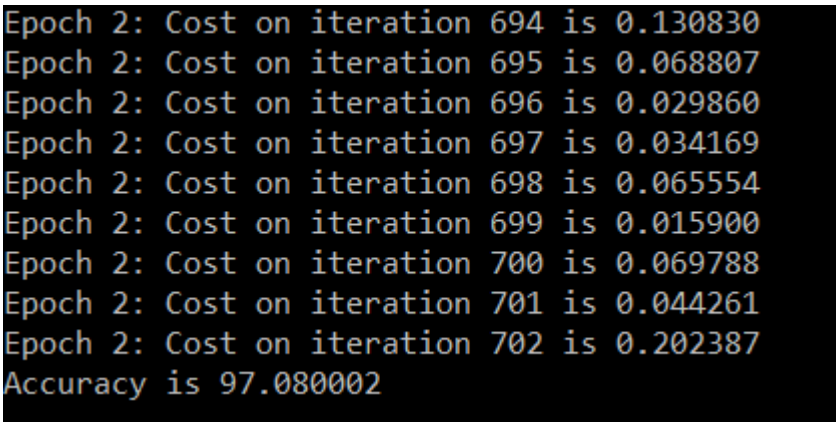


Figura 9-1: Resultado tras la ejecución del programa.

10 CONCLUSIONES

El mundo de las redes neuronales está avanzando cada vez más y se abre camino entre la industria y ramas de la ciencia. Además, se llega a la conclusión de que no es necesario conocer algoritmos muy complejos de los sistemas para poder obtener una salida con una alta fiabilidad (mayor del 97%), esto hace que las redes neuronales sean una herramienta muy útil y muy rápida (una vez sea entrenada la red).

Uno de los problemas de las redes neuronales es encontrar el número de filtros, neuronas ocultas,… óptimos para un buen resultado a la salida. Este problema sólo se puede resolver con la práctica y la experiencia. Si bien es posible hacer un estudio del número de neuronas, éste puede o no ser el mejor.

Otro problema es el tiempo de entrenamiento, esta red ha tratado imágenes de 28x28 pixeles, el tiempo de entrenamiento podría ser demasiado lento con imágenes muy grandes y a veces podría ser intratable. Por eso es conveniente disponer de un ordenador con buenas prestaciones y en su defecto simplificar las entradas de la mejor manera posible para reducir el tiempo de entrenamiento y que no se vea afectados los valores de los parámetros de la red.

Este proyecto ha sido usado para reconocimiento de números pero también podría haber sido útil para reconocer señales de tráfico (muy útil para los vehículos autónomos), tratamiento de imágenes con el objetivo de reconocer una cierta figura en una zona de una imagen, usar este proyecto en un área industrial, por ejemplo, para clasificar diferentes cajas que contengan diferente materia prima. El área industrial podría ser una buena idea ya que aunque normalmente no se encuentran ordenadores muy modernos con los que entrenar la red pero una vez entrenada la red se pueden guardar los parámetros de la red en un fichero para que un ordenador pueda leerlo y de esta manera el problema de reconocimiento sería muy rápido.

Por último, con este proyecto de pretende que el lector pueda tener una introducción a las redes neuronales, y sobretodo a las redes convolucionales y a su método de retropropagación que ha sido desde varios años un misterio en las redes neuronales. Además, se incita al lector a investigar en esta rama de la tecnología y probar a usar el código con otro tipo de imgánes.

REFERENCIAS

[1] Jeff Heaton. (2012). *Introduction to the Math of Neural Networks.*

[2] Beale, M., Hagan, M., Demuth, H. (2016). *Neural Networks Toolbox.*

[3] Dr. Timothy Masters. (2016). *Deep Belief Nets in C++ and CUDA C.*

[4] Christopher M. Bishop. (1996). *Neural Networks for Pattern Recognition.*

[5] Hagan, M., Demuth, H., Beale, M. and De Jesús, O. (2014). *Neural network design. 2nd ed.*

[6] Stackoverflow.com. (2017). *Stack Overflow*. [online] Available at: https://stackoverflow.com.

[7] Ujjwalkarn. (2016). *An Intuitive Explanation of Convolutional Neural Networks*. [online] Available at: https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/.

[8] Ujjwalkarn. (2016). *A Quick Intro of Neuronal networks*. [online] Available at: https://ujjwalkarn.me/2016/08/09/quick-intro-neural-networks/

[9] Jacob S. (2015). *MNIST Hand Written Digits Classification Benchmark*. [online] Available at: http://knowm.org/mnist-hand-written-digits-classification-benchmark/

[10] Michael Nielsen. (2017). *Deep learning*. [online] Available at: http://neuralnetworksanddeeplearning.com/chap6.html

[11] Adit Deshpande. (2016). *Going Deeper Through the Network.* [online] Available at: http://www.kdnuggets.com/2016/09/beginners-guide-understanding-convolutional-neural-networks

[12] Mike O'Neill. (2006). *Neural Network for Recognition of Handwritten Digits.*[online] Available at: https://www.codeproject.com/Articles/16650/Neural-Network-for-Recognition-of-Handwritten-Digi

[13] Chris McCormick. (2015). *Understanding the DeepLearnToolbox CNN Example.* (2015). [online] Available at: http://mccormickml.com/2015/01/10/understanding-the-deeplearntoolbox-cnn-example/

[14] Jefkine. (2015). *Backpropagation In Convolutional Neural Networks.* [online] Available at: http://www.jefkine.com/general/2016/09/05/backpropagation-in-convolutional-neural-networks/.

[15] Mazur M. (2015). *A Step by Step Backpropagation Example.* [online] Available at: https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/.

[16] N4n0. (2017). *Tutorial de Redes Neuronaes con VREP C++ y Linux*. [online] Available at: https://robologs.net/2017/01/22/tutorial-de-redes-neuronales-con-vrep-c-y-linux/

[17] Yann LeCun, Corinna Cortes, Christopher J.C. Burges. (2017). *The MNIST database*. [online] Available at: http://yann.lecun.com/exdb/mnist/

[18] OpenCV.com. (2017). *OpenCV*.[online] Available at: http://opencv.org/

[19] Deeplearning.stanford.edu. (2017). *Unsupervised Feature Learning and Deep Learning Tutorial*. [online] Available at: http://deeplearning.stanford.edu/tutorial/.

[20] Lawebdelprogramador.com. (2017). La Web del Programador. [online] Available at: http://www.lawebdelprogramador.com.

APÉNDICE: CÓDIGO DE LA CNN

```
#include "opencv2/core/core.hpp"
#include "opencv2/imgproc/imgproc.hpp"
#include "opencv2/highgui/highgui.hpp"
#include <math.h>
#include <fstream>
#include <iostream>
#include <time.h>


using namespace cv;
using namespace std;

#define db_numFilters 20
#define db_filterDim 9
#define db_poolDim 2
#define ATD at<double>
#define CONV_FULL 0
#define CONV_SAME 1
#define CONV_VALID 2

bool pred = false;

struct mystruct
{
	int epochs;
	int minibatch;
	float alpha;
	float momentum;
};



int
ReverseInt(int i) {
	unsigned char ch1, ch2, ch3, ch4;
	ch1 = i & 255;
	ch2 = (i >> 8) & 255;
	ch3 = (i >> 16) & 255;
	ch4 = (i >> 24) & 255;
	return((int)ch1 << 24) + ((int)ch2 << 16) + ((int)ch3 << 8) + ch4;
}

Mat
concatenateMat(vector<vector<Mat> > &vec) {

	int subFeatures = vec[0][0].rows * vec[0][0].cols;
	int height = vec[0].size() * subFeatures;
	int width = vec.size();
	Mat res = Mat::zeros(height, width, CV_64FC1);

	for (int i = 0; i<vec.size(); i++) {
		for (int j = 0; j<vec[i].size(); j++) {
			Rect roi = Rect(i, j * subFeatures, 1, subFeatures);
			Mat subView = res(roi);
```

```
                Mat ptmat = vec[i][j].reshape(0, subFeatures);
                ptmat.copyTo(subView);
            }
        }
        return res;
}

Mat
concatenateMat(vector<Mat> &vec) {

        int height = vec[0].rows;
        int width = vec[0].cols;
        Mat res = Mat::zeros(height * width, vec.size(), CV_64FC1);
        for (int i = 0; i<vec.size(); i++) {
            Mat img(vec[i]);
            // reshape(int cn, int rows=0), cn is num of channels.
            Mat ptmat = img.reshape(0, height * width);
            Rect roi = cv::Rect(i, 0, ptmat.cols, ptmat.rows);
            Mat subView = res(roi);
            ptmat.copyTo(subView);
        }
        return res;
}

void
unconcatenateMat(Mat &M, vector<vector<Mat> > &vec, int vsize) {

        int sqDim = M.rows / vsize;
        int Dim = sqrt(sqDim);
        for (int i = 0; i<M.cols; i++) {
            vector<Mat> oneColumn;
            for (int j = 0; j<vsize; j++) {
                Rect roi = Rect(i, j * sqDim, 1, sqDim);
                Mat temp;
                M(roi).copyTo(temp);
                Mat img = temp.reshape(0, Dim);
                oneColumn.push_back(img);
            }
            vec.push_back(oneColumn);
        }
}

Mat
kron(Mat &a, Mat &b) {

        Mat res = Mat::zeros(a.rows * b.rows, a.cols * b.cols, CV_64FC1);
        for (int i = 0; i<a.rows; i++) {
            for (int j = 0; j<a.cols; j++) {
                Rect roi = Rect(j * b.cols, i * b.rows, b.cols, b.rows);
                Mat temp = res(roi);
                Mat c = b.mul(a.ATD(i, j));
                c.copyTo(temp);
            }
        }
        return res;
}

void
read_Mnist(string filename, vector<Mat> &vec) {
        ifstream file(filename, ios::binary);
        if (file.is_open()) {
```

```
		int magic_number = 0;
		int number_of_images = 0;
		int n_rows = 0;
		int n_cols = 0;
		file.read((char*)&magic_number, sizeof(magic_number));
		magic_number = ReverseInt(magic_number);
		file.read((char*)&number_of_images, sizeof(number_of_images));
		number_of_images = ReverseInt(number_of_images);
		file.read((char*)&n_rows, sizeof(n_rows));
		n_rows = ReverseInt(n_rows);
		file.read((char*)&n_cols, sizeof(n_cols));
		n_cols = ReverseInt(n_cols);
		for (int i = 0; i < number_of_images; ++i) {
			Mat tpmat = Mat::zeros(n_rows, n_cols, CV_8UC1);
			for (int r = 0; r < n_rows; ++r) {
				for (int c = 0; c < n_cols; ++c) {
					unsigned char temp = 0;
					file.read((char*)&temp, sizeof(temp));
					tpmat.at<uchar>(r, c) = (int)temp;
				}
			}
			vec.push_back(tpmat);
		}
	}
}

void
read_Mnist_Label(string filename, Mat &mat)
{
	ifstream file(filename, ios::binary);
	if (file.is_open()) {
		int magic_number = 0;
		int number_of_images = 0;
		int n_rows = 0;
		int n_cols = 0;
		file.read((char*)&magic_number, sizeof(magic_number));
		magic_number = ReverseInt(magic_number);
		file.read((char*)&number_of_images, sizeof(number_of_images));
		number_of_images = ReverseInt(number_of_images);
		for (int i = 0; i < number_of_images; ++i) {
			unsigned char temp = 0;
			file.read((char*)&temp, sizeof(temp));
			mat.ATD(0, i) = (double)temp;
		}
	}
}

void
readData(vector<Mat> &x, Mat &y, string xpath, string ypath, int number_of_images) {

	//read MNIST iamge into OpenCV Mat vector
	read_Mnist(xpath, x);
	for (int i = 0; i<x.size(); i++) {
		x[i].convertTo(x[i], CV_64FC1, 1.0 / 255, 0);
	}
	//read MNIST label into double vector
	y = Mat::zeros(1, number_of_images, CV_64FC1);
	read_Mnist_Label(ypath, y);
}

Mat
```

```
sigmoid(Mat &M) {
	Mat temp;
	exp(-M, temp);
	return 1.0 / (temp + 1.0);
}

// Mimic rot90() in Matlab/GNU Octave.
Mat
rot90(Mat &M, int k) {
	Mat res;
	if (k == 0) return M;
	else if(k == 1) {
		flip(M.t(), res, 0);
	}
	else {
		flip(rot90(M, k - 1).t(), res, 0);
	}
	return res;
}


// A Matlab/Octave style 2-d convolution function.
// from http://blog.timmlinder.com/2011/07/opencv-equivalent-to-matlabs-conv2-
function/
Mat
conv2(Mat &img, Mat &kernel, int convtype) {
	Mat dest;
	Mat source = img;
	if (CONV_FULL == convtype) {
		source = Mat();
		int additionalRows = kernel.rows - 1, additionalCols = kernel.cols - 1;
		copyMakeBorder(img, source, (additionalRows + 1) / 2, additionalRows /
2, (additionalCols + 1) / 2, additionalCols / 2, BORDER_CONSTANT, Scalar(0));
	}
	Point anchor(kernel.cols - kernel.cols / 2 - 1, kernel.rows - kernel.rows / 2
- 1);
	int borderMode = BORDER_CONSTANT;
	Mat fkernal;
	flip(kernel, fkernal, -1);
	filter2D(source, dest, img.depth(), fkernal, anchor, 0, borderMode);

	if (CONV_VALID == convtype) {
		dest = dest.colRange((kernel.cols - 1) / 2, dest.cols - kernel.cols / 2)
			.rowRange((kernel.rows - 1) / 2, dest.rows - kernel.rows / 2);
	}
	return dest;
}

vector<int>
create(int poolDim, int convolvedDim)
{
	vector<int> m(convolvedDim / poolDim);
	int j = 0;
	for (int i = 0; i < convolvedDim / poolDim; i++)
	{
		m[i] = j;
		j = j + poolDim;
	}

	return m;
}
```

```
Mat
extractMat(Mat &m, vector<int> &vec)
{
	Mat x = Mat::zeros(vec.size(), vec.size(), CV_64F);
	for (int i = 0; i < vec.size(); i++)
	{
		for (int j = 0; j < vec.size(); j++)
		{
			x.at<double>(i, j) = m.at<double>(vec[i], vec[j]);
		}
	}

	return x;
}


vector<Mat>
reshape2D3D(Mat &A, int dim1, int dim2, int dim3)
{
	Mat m = A.col(0);
	for (int i = 0; i<A.cols - 1; i++)
		vconcat(m, A.col(i + 1), m);


	Mat z;
	Mat spm;
	Mat dest = Mat(dim2, dim3, CV_64F);
	vector<Mat> B;
	int r = 0;

	for (int k = 0; k<dim1; k++)
	{
		z = m(Range(dim2*dim3 * r, dim2*dim3 * r + dim2*dim3), Range(0, 1));
		dest = z.reshape(1, dim2);
		dest = dest.t();
		B.push_back(dest);
		r = r++;
	}
	return B;
}


Mat
reshape3D2D(vector<Mat> &input, int numImages)
{

	Mat algo;
	Mat dest;
	Mat res;
	dest = input[0].col(0);
	for (int k = 1; k<input[0].cols; k++)
		vconcat(dest, input[0].col(k), dest);
	int t;
	for (int i = 0; i<size(input); i++)
	{
		if (i == 0)
		{
			t = 0;
		}
		else
```

```
		{
			algo = (input[i]);
			for (int k = 0; k<algo.cols; k++)
				vconcat(dest, algo.col(k), dest);
		}
	}
	int filas = (size(input)*input[0].size[0] * input[0].size[0]) / numImages;

	res = Mat(filas, numImages, CV_64F);

	for (int i = 0; i < numImages; i++)
	{
		(dest.colRange(0, 1).rowRange(i*filas, (i +
1)*filas)).copyTo(res.col(i));
	}

	return res;
}


Mat
reshape4D2D(vector<vector<Mat>> &input, int numImages)
{
	Mat algo;
	Mat dest;
	Mat res;

	Mat aux;
	aux = input[0][0].t();
	dest = aux.reshape(0, 1).t();
	for (int i = 0; i<size(input); i++)
	{
		for (int j = 0; j<size(input[0]); j++)
		{
			if (!(i == 0 && j == 0))
			{
				aux = (input[i][j]).t();
				aux = aux.reshape(0, 1);
				aux = aux.t();
				vconcat(dest, aux, dest);
			}
		}
	}
	int filas = (size(input)*size(input[0])*input[0][0].size[0] *
input[0][0].size[0]) / numImages;

	res = Mat(filas, numImages, CV_64F);


	for (int i = 0; i < numImages; i++)
	{
		(dest.colRange(0, 1).rowRange(i*filas, (i +
1)*filas)).copyTo(res.col(i));
	}

	return res;
}


vector<vector<Mat>>
reshape2D4D(Mat &A, int dim1, int dim2, int dim3, int dim4)
```

```
{

	Mat m = A.t();
	m = m.reshape(0, A.size[0]*A.size[1]);

	Mat z;
	Mat spm;
	Mat dest = Mat(dim3, dim4, CV_64F);
	vector<vector<Mat>> B;
	Mat M;
	int r = 0;

	for (int k = 0; k<dim1; k++)
	{
		B.push_back(vector<Mat>());
		for (int l = 0; l<dim2; l++)
		{
			z = m(Range(dim3*dim4 * r, dim3*dim4 * r + dim3*dim4), Range(0,
1));
			dest = z.reshape(1, dim3);
			dest = dest.t();
			B[k].push_back(dest);
			r = r++;
		}
	}

	return B;
}




Mat
prediction(Mat &A)
{
	Mat maximo = Mat::zeros(1, A.size[1], CV_64F);
	Mat m = Mat::zeros(1, A.size[1], CV_64F);
	for (int i = 0; i < A.size[1]; i++)
	{
		maximo.at<double>(0, i) = A.at<double>(0, i);
		for (int j = 0; j < A.size[0]; j++)
		{
			if (maximo.at<double>(0, i) < A.at<double>(j, i))
			{
				maximo.at<double>(0, i) = A.at<double>(j, i);
				m.at<double>(0, i) = j;
			}
		}
	}
	return m;
}

Mat
bsxfun_div(Mat &A)
{
	Mat res = Mat::zeros(A.size[0], A.size[1], CV_64F);
	for (int i = 0; i < A.size[1]; i++)
	{
		res.col(i) = A.col(i) / sum(A.col(i))[0];
	}
```

```
	return res;
}

// cnnInitParams
Mat
cnnInitParams(int imageDim, int filterDim, int numFilters, int poolDim, int
numClasses) {

	assert(filterDim < imageDim);

	Mat mean = Mat::zeros(1, 1, CV_64F);
	Mat sigma = Mat::ones(1, 1, CV_64F);

	Mat Wc = Mat::zeros(filterDim, filterDim*numFilters, CV_64F);
	randn(Wc, 0, 1);
	Wc = 0.1*Wc;

	int outDim = imageDim - filterDim + 1;

	assert(outDim%poolDim == 0);

	outDim = outDim / poolDim;
	int hiddenSize = outDim*outDim*numFilters;

	double r = sqrt(6) / sqrt(numClasses + hiddenSize + 1);

	Mat Wd;
	Wd = Mat::zeros(numClasses, hiddenSize, CV_64F);
	randn(Wd, 0, 1);
	Wd = Wd * 2 * r - r;

	Mat bc;
	bc = Mat::zeros(numFilters, 1, CV_64F);

	Mat bd;
	bd = Mat::zeros(numClasses, 1, CV_64F);

	Wc = Wc.reshape(0, 1);
	Wd = Wd.reshape(0, 1);
	bc = bc.reshape(0, 1);
	bd = bd.reshape(0, 1);

	Mat res;
	hconcat(Wc, Wd, res);
	hconcat(res, bc, res);
	hconcat(res, bd, res);

	return res;
}

//cnnConvolved
vector<vector<Mat>>
cnnConvolve(int filterDim, int numFilters, vector<Mat> &images, vector<Mat> &W, Mat
&b)
{
	int numImages = size(images);
	int imageDim = images[0].size[0];
	int convDim = imageDim - filterDim + 1;

	vector<vector<Mat>> convolvedFeatures;
```

```
	for (int i = 0; i<numImages; i++) {
		convolvedFeatures.push_back(vector<Mat>());
		for (int j = 0; j<numFilters; j++)
			convolvedFeatures[i].push_back(Mat(convDim, convDim, CV_64F,
Scalar::all(0)));
	}
	Mat filter;
	Mat convolvedImage;
	Mat im;
	for (int imageNum = 0; imageNum < numImages; imageNum++)
	{
		im = images[imageNum];
		for (int filterNum = 0; filterNum < numFilters; filterNum++)
		{
			convolvedImage = Mat::zeros(convDim, convDim, CV_64F);
			filter = W[filterNum];
			filter = rot90(filter, 2);
			convolvedImage = convolvedImage + conv2(im, filter, CONV_VALID);
			convolvedImage = convolvedImage + b.at<double>(filterNum, 0);
			convolvedImage = sigmoid(convolvedImage);
			convolvedImage.copyTo(convolvedFeatures[imageNum][filterNum]);
		}
	}

	return convolvedFeatures;
}


//cnnPool
vector<vector<Mat>>
cnnPool(int poolDim, vector<vector<Mat>> &convolvedFeatures)
{
	int numImages = size(convolvedFeatures);
	int numFilters = size(convolvedFeatures[0]);
	int convolvedDim = convolvedFeatures[0][0].size[0];

	vector<vector<Mat>> pooledFeatures;
	for (int i = 0; i<numImages; i++) {
		pooledFeatures.push_back(vector<Mat>());
		for (int j = 0; j<numFilters; j++)
			pooledFeatures[i].push_back(Mat(convolvedDim / poolDim,
convolvedDim / poolDim, CV_64F, Scalar::all(0)));
	}

	int poolDimSquared = poolDim * poolDim;
	Mat poolFilter = Mat::ones(poolDim, poolDim, CV_64F);
	Mat feature;
	Mat convolvedFeature = Mat::zeros(19, 19, CV_64F);;
	Mat A = Mat::zeros(10,10, CV_64F);
	vector<int> poolIndex = create(poolDim, convolvedDim);
	Mat extra;
	for(int imageNum = 0; imageNum < numImages; imageNum++)
	{
		for(int filterNum = 0; filterNum < numFilters; filterNum++)
		{
			feature = convolvedFeatures[imageNum][filterNum];
			convolvedFeature = conv2(feature, poolFilter, CONV_VALID);
			extra = extractMat(convolvedFeature, poolIndex) / poolDimSquared;
			extra.copyTo(pooledFeatures[imageNum][filterNum]);
		}
	}
```

```
	return pooledFeatures;
}


vector<Mat>
cnnParamsToStack(Mat &theta,int imageDim,int filterDim,int numFilters,int poolDim,int
numClasses)
{

	int outDim = (imageDim - filterDim + 1) / poolDim;
	int hiddenSize = outDim*outDim*numFilters;

	int indS = 0;
	int indE = filterDim * filterDim * numFilters;
	vector<Mat> Wc;
	Wc = reshape2D3D(theta, numFilters, filterDim, filterDim);
	indS = indE + 1;
	indE = indE + hiddenSize*numClasses;
	Mat Wd = Mat::zeros(numClasses, hiddenSize, CV_64F);

	Wd = theta(Range(0, 1), Range(indS-1,indS + numClasses*hiddenSize-1));
	Wd = Wd.t();
	Mat Wd_aux = Mat::zeros(10, hiddenSize, CV_64F);
	for (int i = 0; i < hiddenSize; i++)
	{
		(Wd.colRange(0, 1)).rowRange(i * 10, (i + 1) *
10).copyTo(Wd_aux.col(i));
	}

	indS = indE + 1;
	indE = indE + numFilters;
	Mat bc = Mat::zeros(numFilters, 1, CV_64F);
	bc = theta(Range(0, 1), Range(indS-1, indS + numFilters-1));
	bc = bc.t();

	indS = indS + 19;
	Mat bd = Mat::zeros(10, 1, CV_64F);
	bd = theta(Range(0, 1), Range(indS, indS + 10));
	bd = bd.t();

	vector<Mat> res;
	Mat Wc_conc;
	Wc_conc = reshape3D2D(Wc, 1);
	res.push_back(Wc_conc);
	res.push_back(Wd_aux);
	res.push_back(bc);
	res.push_back(bd);


	return res;

}




vector<Mat>
cnnCost(Mat &theta,vector<Mat> &images,Mat &labels,int numClasses,int filterDim,int
numFilters,int poolDim,bool pred)
{
```

```
	int imageDim = images[0].size[0];
	int numImages = size(images);
	vector<Mat> params;
	params = cnnParamsToStack(theta, imageDim, filterDim, numFilters, poolDim,
numClasses);
	Mat Wc = params[0];
	Mat Wd = params[1];
	Mat bc = params[2];
	Mat bd = params[3];

	vector<Mat> Wc3D;
	Wc3D = reshape2D3D(Wc, numFilters, filterDim, filterDim);
	vector<Mat> Wc_grad;
	for (int i = 0; i < size(Wc3D); i++)
	{
		Wc_grad.push_back(Mat::zeros(Wc3D[0].size(), CV_64F));
	}
	Mat Wd_grad = Mat::zeros(Wd.size[1], Wd.size[0], CV_64F);
	Mat bc_grad = Mat::zeros(bc.size[1], bc.size[0], CV_64F);
	Mat bd_grad = Mat::zeros(bd.size[1], bd.size[0], CV_64F);
	int convDim = imageDim - filterDim + 1;
	int outputDim = (convDim) / poolDim;

	vector<vector<Mat>> activations;
	for (int i = 0; i<numImages; i++) {
		activations.push_back(vector<Mat>());
		for (int j = 0; j<numFilters; j++)
			activations[i].push_back(Mat(convDim, convDim, CV_64F,
Scalar::all(0)));
	}

	vector<vector<Mat>> activationsPooled;
	for (int i = 0; i<numImages; i++) {
		activationsPooled.push_back(vector<Mat>());
		for (int j = 0; j<numFilters; j++)
			activationsPooled[i].push_back(Mat(outputDim, outputDim, CV_64F,
Scalar::all(0)));
	}
	activations = cnnConvolve(filterDim, numFilters, images, Wc3D, bc);
	activationsPooled = cnnPool(poolDim, activations);
	Mat activationsPooled_r = reshape4D2D(activationsPooled, numImages);
	Mat probs = Mat::zeros(numClasses, numImages, CV_64F);
	int M = size(images);
	Mat aux1 = Wd*activationsPooled_r +repeat(bd, 1, M);
	Mat maxR;
	reduce(aux1, maxR, 0, REDUCE_MAX);
	maxR = repeat(maxR, aux1.size[0], 1);
	Mat aux2 = aux1 - maxR;
	Mat aux3;
	exp(aux2, aux3);
	probs = bsxfun_div(aux3);
	aux2.release();
	aux3.release();
	Mat groundTruth = Mat::zeros(10, numImages, CV_64F);
	for (int i = 0; i < labels.size[1]; i++)
	{
		groundTruth.at<double>(labels.at<double>(0, i), i) = 1;
	}
	Mat aux4 = groundTruth.mul(probs);
	aux4 = aux4.reshape(0, 1);
	Mat aux5;
```

```
	for (int i = 0; i < aux4.size[1]; i++)
	{
		if (aux4.ATD(0, i) != 0)
		{
			aux5.push_back(log(aux4.ATD(0, i)));
		}
	}
	Mat cost = Mat::zeros(1,1,CV_64F);
	cost = -mean(aux5)[0];
	aux4.release();
	aux5.release();
	Mat preds;
	if (pred == true)
	{
		preds = prediction(probs);
	}
	Mat deriv_1 = Mat(groundTruth.size[1], groundTruth.size[0], CV_64F);
	double numero = 1.0 / M;
	deriv_1 = -numero*(groundTruth - probs);
	groundTruth.release();
	Wd_grad = deriv_1*activationsPooled_r.t();
	activationsPooled.clear();
	activationsPooled_r.release();
	bd_grad = deriv_1*(Mat::ones(M, 1, CV_64F));
	Mat deriv_2_pooled_sh = Wd.t()*deriv_1;
	deriv_1.release();


	vector<vector<Mat>> deriv_2_pooled;
	deriv_2_pooled = reshape2D4D(deriv_2_pooled_sh, numImages, numFilters,
outputDim, outputDim);
	vector<vector<Mat>> deriv_2_upsampled;
	for (int i = 0; i<numImages; i++) {
	deriv_2_upsampled.push_back(vector<Mat>());
	for (int j = 0; j<numFilters; j++)
	deriv_2_upsampled[i].push_back(Mat(convDim, convDim, CV_64F, Scalar::all(0)));
	}
	Mat A = Mat::ones(poolDim, poolDim, CV_64F);
	Mat f_now;
	Mat noww = Mat::ones(9,9,CV_64F);
	Mat im;
	Mat A1;
	for(int imageNum=0;imageNum<numImages;imageNum++)
	{
		im = images[imageNum];
		for(int filterNum=0;filterNum<numFilters;filterNum++)
		{
			aux3 = (1.0 / (poolDim *
poolDim))*kron(deriv_2_pooled[imageNum][filterNum], A);
			A1 = (aux3.mul(activations[imageNum][filterNum])).mul(1.0 -
activations[imageNum][filterNum]);
			A1.copyTo(deriv_2_upsampled[imageNum][filterNum]);
			deriv_2_upsampled[imageNum][filterNum].copyTo(f_now);
			noww = conv2(im, rot90(f_now, 2), CONV_VALID);
			Wc_grad[filterNum] = Wc_grad[filterNum] + noww;
			bc_grad.at<double>(0, filterNum) = bc_grad.at<double>(0,
filterNum) + sum(f_now)[0];
		}
	}
```

```
	activations.clear();
	deriv_2_pooled.clear();
	deriv_2_upsampled.clear();
	images.clear();
	Mat Wc_conc;
	Wc_conc = reshape3D2D(Wc_grad, 1);
	Wc_conc = Wc_conc.reshape(0, 1);
	//
	Wd_grad = Wd_grad.t();
	//
	Wd_grad = Wd_grad.reshape(0, 1);
	bc_grad = bc_grad.reshape(0, 1);
	bd_grad = bd_grad.reshape(0, 1);

	Mat grad;
	hconcat(Wc_conc, Wd_grad, grad);
	hconcat(grad, bc_grad, grad);
	hconcat(grad, bd_grad, grad);

	vector<Mat> res;
	res.push_back(cost);
	res.push_back(grad);
	res.push_back(preds);
	return res;
}



Mat
minFuncSGD(Mat &theta,vector<Mat> &data,Mat &labels, mystruct &options,int
numClasses,int filterDim,int numFilters,int poolDim)
{
	int epochs = options.epochs;
	double alpha = options.alpha;
	alpha = 0.2;
	int minibatch = options.minibatch;
	int m = 60000;
	float mom = 0.5;
	mom = 0.6;
	float momIncrease = 20;
	Mat velocity = Mat::zeros(theta.size[0], theta.size[1], CV_64F);
	vector<int> rp;
	for (int i = 0; i < m; ++i) rp.push_back(i);

	vector<Mat> mb_data;

	Mat mb_labels;
	double cost = 0;
	Mat grad;
	vector<Mat> params;
	int it = 0;
	Mat theta_1 = theta;

		for (int e = 0; e < epochs; e++)
		{
			std::random_shuffle(rp.begin(), rp.end());
			for (int s = 0; s < m - minibatch; s++)
			{
				it = it + 1;
```

```
                        if (it == momIncrease)
                        {
                                mom = options.momentum;
                        }

                        for (int i = 0; i < minibatch; i++)
                        {
                                mb_data.push_back(data[i + s]);
                        }
                        mb_labels = labels(Range(0, 1), Range(s, s + minibatch));
                        params = cnnCost(theta_1, mb_data, mb_labels, numClasses,
filterDim, numFilters, poolDim, 0);
                        cost = params[0].at<double>(0, 0);
                        grad = params[1];
                        velocity = (mom*velocity) + (alpha*grad);
                        theta_1 = theta_1 - velocity;
                        printf("Epoch %d: Cost on iteration %d is %f\n", e, it,
cost);
                        s = s + 255;
                }
                alpha = alpha / 2.0;
        }
    return theta_1;
}


int main()
{


    int imageDim = 28;
    int numClasses = 10;
    int filterDim = 9;
    int numFilters = 20;
    int poolDim = 2;


    static vector<Mat> trainX;
    static vector<Mat> testX;
    Mat trainY, testY;
    readData(trainX, trainY, "mnist/train-images-idx3-ubyte", "mnist/train-labels-
idx1-ubyte", 60000);
    readData(testX, testY, "mnist/t10k-images-idx3-ubyte", "mnist/t10k-labels-
idx1-ubyte", 10000);
    //cout << trainY.cols <<endl;

    cout << "Read trainX successfully, including " << trainX[0].cols *
trainX[0].rows << " features and " << trainX.size() << " samples." << endl;
    cout << "Read trainY successfully, including " << trainY.cols << " samples" <<
endl;
    cout << "Read testX successfully, including " << testX[0].cols * testX[0].rows
<< " features and " << testX.size() << " samples." << endl;
    cout << "Read testY successfully, including " << testY.cols << " samples" <<
endl;


    cout << testY;

    Mat theta;
    theta = cnnInitParams(imageDim, filterDim, numFilters, poolDim, numClasses);
```

```
	mystruct options;
	options.epochs = 3;
	options.minibatch = 256;
	options.alpha = 0.1;
	options.momentum = 0.95;
	Mat opttheta;
	opttheta = minFuncSGD(theta, trainX, trainY, options, numClasses, filterDim,
numFilters, poolDim);

	vector<Mat> final;
	final = cnnCost(opttheta, testX, testY, numClasses, filterDim, numFilters,
poolDim, true);
	float suma = 0;
	Mat preds = final[2];
	cout << preds <<endl;
	for (int i = 0; i < preds.size[1]; i++)
	{
		if (preds.at<double>(0, i) == testY.at<double>(0, i))
		{
			suma++;
		}
	}
	float acc = (suma / preds.size[1])*100.0;

	printf("Accuracy is %f\n",acc);
	cout << preds.type() << endl;
	cout << testY.type() << endl;

	system("\n\npause");
	return 0;
}
```

GLOSARIO

SGD: Stochastic Gradient Descent 27

CNN: Convolutional Neuronal Network 33