



*ugr* | Universidad  
de **Granada**

TRABAJO FIN DE GRADO  
INGENIERÍA INFORMÁTICA

**Implementación optimizada sobre  
sistemas heterogéneos de algoritmos de  
Deep Learning para clasificación de  
imágenes**

---

**Autor**  
David Sánchez Pérez

**Directores**  
José Miguel Mantas Ruiz



ESCUELA TÉCNICA SUPERIOR DE INGENIERÍAS INFORMÁTICA Y DE  
TELECOMUNICACIÓN

—  
Granada, mes de Febrero 2024









**Implementación optimizada sobre  
sistemas heterogéneos de algoritmos de  
Deep Learning para clasificación de  
imágenes**

---

**Autor**

David Sánchez Pérez

**Directores**

José Miguel Mantas Ruiz



**Título del Proyecto: Subtítulo del proyecto**

Nombre Apellido1 Apellido2 (alumno)

**Palabras clave:** palabra\_clave1, palabra\_clave2, palabra\_clave3, .....

**Resumen**

Poner aquí el resumen.



**Project Title: Project Subtitle**

First name, Family name (student)

**Keywords:** Keyword1, Keyword2, Keyword3, ....

**Abstract**

Write here the abstract in English.



---

Yo, **Nombre Apellido1 Apellido2**, alumno de la titulación **TITULACIÓN de la Escuela Técnica Superior de Ingenierías Informática y de Telecomunicación de la Universidad de Granada**, con DNI XXXXXXXXX, autorizo la ubicación de la siguiente copia de mi Trabajo Fin de Grado en la biblioteca del centro para que pueda ser consultada por las personas que lo deseen.

Fdo: Nombre Apellido1 Apellido2

Granada a X de mes de 201 .



---

D. **Nombre Apellido1 Apellido2 (tutor1)**, Profesor del Área de XXXX del Departamento YYYY de la Universidad de Granada.

D. **Nombre Apellido1 Apellido2 (tutor2)**, Profesor del Área de XXXX del Departamento YYYY de la Universidad de Granada.

**Informan:**

Que el presente trabajo, titulado ***Título del proyecto, Subtítulo del proyecto***, ha sido realizado bajo su supervisión por **Nombre Apellido1 Apellido2 (alumno)**, y autorizamos la defensa de dicho trabajo ante el tribunal que corresponda.

Y para que conste, expiden y firman el presente informe en Granada a X de mes de 201 .

**Los directores:**

**Nombre Apellido1 Apellido2 (tutor1)**      **Nombre Apellido1 Apellido2 (tutor2)**



# Agradecimientos

Poner aquí agradecimientos...



# Índice general

<b>1. Introducción</b>	<b>1</b>
1.1. Resumen . . . . .	1
1.2. Objetivos . . . . .	2
1.2.1. Objetivos de aprendizaje . . . . .	3
1.2.2. Objetivos de diseño e implementación . . . . .	3
<b>2. Conceptos previos</b>	<b>5</b>
2.1. Machine Learning . . . . .	5
2.2. Aprendizaje supervisado . . . . .	5
2.3. División de datos en entrenamiento y test . . . . .	6
2.4. Redes Neuronales Totalmente Conectadas . . . . .	6
2.4.1. Neurona . . . . .	7
2.4.2. Estructura por capas . . . . .	8
2.4.3. Funciones de activación . . . . .	8
2.4.4. Capa SoftMax . . . . .	10
2.4.5. Tipos de codificaciones de etiquetas . . . . .	11
2.4.6. Función de error o pérdida . . . . .	11
2.4.7. Descenso del gradiente . . . . .	12
2.4.8. Inicialización de pesos . . . . .	13
2.4.9. Inicialización de sesgos . . . . .	13
2.5. Redes Neuronales Convolucionales . . . . .	14
2.5.1. Introducción a CNNs . . . . .	15
2.5.2. Estructura por capas . . . . .	16
2.5.3. Capa convolucional . . . . .	17
2.5.4. Capa de agrupación máxima . . . . .	23
2.5.5. Capa de aplanado . . . . .	25
2.6. OpenMP . . . . .	27
2.7. CUDA . . . . .	28
2.7.1. Gestión de memoria . . . . .	28
2.7.2. Kernels . . . . .	29
2.7.3. Jerarquía de hebras . . . . .	29
2.8. cuDNN (Deep Neural Network) . . . . .	30
2.8.1. Manejador . . . . .	30

---

2.8.2. Tensores . . . . .	30
2.8.3. Principales funciones . . . . .	32
<b>3. Aportaciones</b>	<b>39</b>
3.1. Planificación . . . . .	39
3.2. Retropropagación en redes neuronales totalmente conectadas	43
3.2.1. Retropropagación en capa SoftMax, [1] [2] . . . . .	43
3.2.2. Retropropagación con 1 capa oculta [3] [4] . . . . .	47
3.2.3. Retropropagación con 2 capas ocultas . . . . .	53
3.2.4. Conclusiones . . . . .	61
3.3. Paralelización mediante OpenMP . . . . .	63
3.4. Retropropagación en redes neuronales convolucionales . . . . .	65
3.4.1. Retropropagación con relleno . . . . .	73
<b>4. Adaptación GPU</b>	<b>85</b>
4.1. GEMM . . . . .	85
4.2. Convolución como GEMM . . . . .	85
4.2.1. Memoria requerida al emplear GEMM . . . . .	87
4.3. Retropropagación GEMM en capa convolucional . . . . .	88
4.4. Capa totalmente conectada como GEMM [5] . . . . .	90
4.4.1. Propagación hacia delante . . . . .	90
4.4.2. Retropropagación . . . . .	92
4.4.3. Multiplicación de matrices en CUDA . . . . .	95
<b>5. Comparación entre distintas implementaciones</b>	<b>99</b>
5.0.1. Modelos a emplear . . . . .	99
5.0.2. Comparación de rendimiento . . . . .	101
<b>6. Conclusiones y trabajo futuro</b>	<b>105</b>

# Índice de figuras

2.1.	Esquema de una neurona . . . . .	7
2.2.	Esquema de una capa de neuronas . . . . .	8
2.3.	Imagen de la función de activación ReLU . . . . .	9
2.4.	Imagen de la función de activación Sigmoide . . . . .	10
2.5.	Imagen de la función de activación SoftMax . . . . .	10
2.6.	Ejemplo de funcionamiento del descenso del gradiente . . . . .	12
2.7.	Esquema de las dimensiones en una capa convolucional de una red neuronal convolucional . . . . .	15
2.8.	Estructura por capas en una red neuronal convolucional . . . . .	16
2.9.	Componentes en una capa convolucional . . . . .	17
2.10.	Propagación hacia delante en una capa convolucional . . . . .	18
2.11.	Propagación hacia delante en una capa convolucional con varios canales de profundidad . . . . .	19
2.12.	Propagación hacia delante en una capa convolucional con varios filtros . . . . .	20
2.13.	Relleno sobre un volumen de entrada X . . . . .	21
2.14.	Relleno sobre un volumen de entrada X . . . . .	21
2.14.	Convolución sobre X con relleno completo . . . . .	22
2.15.	Componentes en una capa de agrupación máxima . . . . .	23
2.16.	Propagación hacia delante en una capa de agrupación máxima	23
2.17.	Propagación hacia delante en una capa de agrupación máxima	24
2.18.	Retropropagación en capa de agrupación máxima . . . . .	24
2.19.	Propagación hacia delante en una capa de aplanado . . . . .	26
2.20.	Retropropagación en una capa de aplanado . . . . .	27
2.21.	Jerarquía de hebras en un grid CUDA . . . . .	29
2.22.	Ejemplo de tensor 4D con dimensiones: N=1, C=1, H=5, y W=6 . . . . .	31
2.23.	Ejemplo de tensor 4D NCHW con dimensiones: N=1, C=1, H=5, y W=6 . . . . .	32
3.1.	Estructura de una red totalmente conectada con softmax en la última capa . . . . .	44
3.2.	Red Neuronal totalmente conectada con 1 capa oculta . . . . .	48

3.3.	Retropropagación en la capa softmax . . . . .	48
3.4.	Retropropagación respecto a pesos entre la capa oculta y la capa SoftMax . . . . .	49
3.5.	Imagen de los 'caminos' desde la capa softmax hasta la neurona $n_0^1$ . . . . .	50
3.6.	Retropropagación respecto a neuronas de la capa oculta h1 .	50
3.7.	Retropropagación respecto a los pesos entre la capa de entrada y la capa oculta h1 . . . . .	51
3.8.	Imagen de los 'caminos' desde la capa oculta h1 hasta $n_0^0$ . .	52
3.9.	Retropropagación en la capa input . . . . .	53
3.10.	Red Neuronal totalmente conectada con 2 capas ocultas . . .	53
3.11.	Retropropagación en la capa softmax . . . . .	54
3.12.	Retropropagación respecto a los pesos entre la capa oculta h2 y la capa SoftMax . . . . .	54
3.13.	Imagen de los 'caminos' desde la capa softmax hasta $n_0^2$ . . .	55
3.14.	Retropropagación en la capa oculta h2 . . . . .	56
3.15.	Retropropagación respecto a los pesos entre las capas ocultas h1 y h2 . . . . .	57
3.16.	'Caminos' desde la capa softmax hasta $n_0^1$ . . . . .	58
3.17.	Retropropagación en la capa oculta h1 . . . . .	58
3.18.	Retropropagación respecto a los pesos entre la capa de entrada (input) y la capa oculta h1 . . . . .	59
3.19.	'Caminos' desde la capa oculta h1 hasta $n_0^0$ . . . . .	60
3.20.	Retropropagación en la capa input . . . . .	61
3.21.	Retropropagación en la capa l . . . . .	62
3.22.	Retropropagación respecto a los pesos entre la capa l-1 y l .	62
3.23.	Ejemplo de propagación hacia delante en una capa convolucional . . . . .	65
3.24.	Cálculo de $Y_{12}^c$ mediante propagación hacia delante en una capa convolucional . . . . .	67
3.25.	Cálculo de $Y_{21}^c$ mediante propagación hacia delante en una capa convolucional . . . . .	68
3.26.	Cálculo de $Y_{22}^c$ mediante propagación hacia delante en una capa convolucional . . . . .	69
3.27.	Cálculo del gradiente de la pérdida respecto a cada filtro como convolución entre X e Y . . . . .	70
3.28.	Invertir pesos en K tanto horizontal como verticalmente . .	71
3.29.	Cálculo del gradiente de la pérdida respecto a cada valor de entrada como convolución entre K e Y . . . . .	72
3.30.	Ejemplo de retropropagación en una capa convolucional con relleno . . . . .	73
3.31.	Retropropagación de $Y_{11}^c$ . . . . .	73
3.32.	Retropropagación de $Y_{12}^c$ . . . . .	74
3.33.	Retropropagación de $Y_{13}^c$ . . . . .	75

3.34. Retropropagación de $Y_{21}^c$ . . . . .	76
3.35. Retropropagación de $Y_{22}^c$ . . . . .	77
3.36. Retropropagación de $Y_{23}^c$ . . . . .	77
3.37. Retropropagación de $Y_{31}^c$ . . . . .	78
3.38. Retropropagación de $Y_{32}^c$ . . . . .	79
3.39. Retropropagación de $Y_{33}^c$ . . . . .	80
3.40. Cálculo del gradiente de la pérdida respecto a cada filtro como convolución entre X e Y . . . . .	81
3.41. Cálculo del gradiente de la pérdida respecto de la entrada como convolución . . . . .	82
3.42. Cálculo del gradiente de la pérdida respecto a la entrada X con uno o dos niveles de relleno . . . . .	83
4.1. Imagen de una convolución estándar frente a una convolución empleando GEMM . . . . .	86
4.2. Retropropagación en una capa convolucional de forma estándar frente a GEMM respecto a la entrada . . . . .	88
4.3. Retropropagación en una capa convolucional de forma estándar frente a GEMM respecto a los pesos . . . . .	89
4.4. Propagación GEMM hacia delante en una capa totalmente conectada . . . . .	90
4.5. Propagación GEMM de un minibatch entero hacia delante en una capa totalmente conectada . . . . .	91
4.6. Cálculo del gradiente respecto a la entrada en una capa totalmente conectada . . . . .	92
4.7. Cálculo del gradiente respecto a la entrada de todo un minibatch en una capa totalmente conectada . . . . .	93
4.8. Cálculo del gradiente respecto a los pesos en una capa totalmente conectada . . . . .	94
4.9. Cálculo del gradiente respecto a los pesos de todo un minibatch en una capa totalmente conectada . . . . .	95
4.10. Tercera implementación de multiplicación matricial con CUDA	96
4.11. Cuarta implementación de multiplicación matricial con CUDA	97
5.1. Secuencial vs OpenMP . . . . .	101
5.2. Ganancia de OpenMP respecto a secuencial . . . . .	102
5.3. OpenMP vs CUDA vs CUDNN . . . . .	102
5.4. CUDA vs CUDNN . . . . .	103



# Índice de cuadros

3.1. Planificación del proyecto . . . . .	43
5.1. Comparación rendimiento CuDNN vs CUDA . . . . .	104



# Capítulo 1

## Introducción

### 1.1. Resumen

En los inicios, las computadoras empleaban exclusivamente CPUs (Central Processing Units) [6] para llevar a cabo tareas de programación de propósito general. Sin embargo, desde la última década empezaron a surgir otros elementos de procesamiento como las GPUs (Graphics Processing Units) [7], las cuales se desarrollaron inicialmente para realizar cálculos gráficos paralelos especializados. Con el tiempo, han ido evolucionando tanto en prestaciones como en versatilidad, permitiendo a día de hoy su uso en tareas de cómputo paralelo de propósito general de alto rendimiento.

Gracias a la difusión de las GPUs en aplicaciones de propósito general, se logró el cambio de sistemas homogéneos a heterogéneos [8], el cual destaca por ser un logro de gran importancia y considerable magnitud en toda la historia de la computación de alto rendimiento.

La computación homogénea emplea uno o más procesadores de la misma arquitectura para ejecutar una aplicación. Por otro lado, la computación heterogénea no se rige por esas reglas y rompe dicha limitación, empleando con ello un conjunto de arquitecturas distintas para ejecutar una misma aplicación, de tal forma que cada arquitectura se encargue de aquellas tareas para las que se encuentra mejor preparada, obteniendo por ello una mejora notable en cuanto a rendimiento.

En el campo de la computación heterogénea destaca el uso agrupado de CPUs y GPUs, pues su conjunto forma una excelente complementación. Mientras que la CPU se encuentra optimizada para tareas dinámicas de ráfagas de cómputo cortas y un flujo de control impredecible, la GPU se especializa justamente en el caso contrario: ráfagas de cómputo muy costosas pero con un flujo de control simple.

De esta forma, si una tarea presenta un número reducido de datos, una lógica de control sofisticada y un bajo nivel de paralelismo, se asignará a la CPU. Si por el contrario, esta presenta una cantidad exuberante de datos,

así como un alto grado de paralelismo al procesar dichos datos, se asignará a la GPU pues presenta un gran número de núcleos y puede dar soporte a una cantidad de hebras mucho mayor que la posible mediante CPU. [9] Tal y como se explicará en detalle en secciones posteriores, el patrón de entrenamiento en redes neuronales convolucionales o CNNs (Convolutional Neural Networks) [10] es computacionalmente intensivo y altamente paralelo [11]. Por ello, se adoptará un enfoque de computación heterogénea en este ámbito, con el propósito de acortar los tiempos de ejecución requeridos en dichos entrenamientos. Las redes neuronales convolucionales son de gran importancia y destacan por ser un subconjunto del aprendizaje automático [12] y el corazón de los algoritmos de aprendizaje profundo [13], además de potenciar las tareas de reconocimiento de imágenes [14] y visión artificial [15]. Entre sus principales usos destacan la detección de objetos en imágenes o videos [16], segmentación de imágenes [17], generación de imágenes [18], análisis de videos [19], procesamiento del lenguaje natural [20], o incluso sistemas autónomos [21], entre otros.

Con el objetivo de lograr una mayor comprensión sobre sistemas heterogéneos aplicados a redes neuronales convolucionales, a lo largo de este proyecto se desarrollarán una serie de implementaciones. Primero se empezará por una implementación secuencial que simplemente use la CPU. Después, se creará otra implementación que aproveche el paralelismo a nivel de grano grueso en CPU mediante la librería OpenMP [22]. Una vez adquiridos conocimientos sobre redes neuronales convolucionales y unas bases paralelismo a nivel de CPU, tendrá lugar la creación de una tercera implementación, caracterizada por ser el primer sistema heterogéneo de este proyecto y emplear el framework CUDA [23]. Por último, una vez entendidas las bases de CNNs, paralelismo tanto a nivel de CPU como a nivel de GPU, y sistemas heterogéneos aplicados a CNNs, se contarán con los conocimientos necesarios para crear y entender cómo funcionan realmente por debajo las librerías del sector. Por ello, se seleccionó una librería de bajo nivel y grandes prestaciones como cuDNN [24] para elaborar una última implementación y lograr obtener un mayor rendimiento, a la vez que se consolida lo aprendido durante todo el proceso.

Cabe destacar que cada una de las implementaciones a desarrollar serán de muy bajo nivel y no se apoyarán en ninguna librería o framework externo que facilite los cálculos, con la única excepción de OpenMP, CUDA y cuDNN, tal y como se comentó anteriormente.

## **1.2. Objetivos**

El principal objetivo de este proyecto es diseñar y desarrollar redes neuronales convolucionales (CNNs) desde sus cimientos, a un nivel de programación relativamente bajo. Esto permite una profunda comprensión de sus

fundamentos y funcionamiento, comunes a bibliotecas especializadas en el campo. Para ello, se desarrollan distintas implementaciones de la misma aplicación, cada una con mejores prestaciones que la anterior.

La principal razón de este proyecto es aprender los fundamentos del machine learning aplicados a redes neuronales convolucionales, así como el diseño y desarrollo de sistemas heterogéneos de altas prestaciones, y el uso de librerías de bajo nivel del ámbito como cuDNN, que a su vez es empleada por otras librerías de más alto nivel del sector como Caffe2 [25], Keras [26], MATLAB [27], Pytorch [28], o TensorFlow [29], entre otras [30].

A continuación se desglosan en dos categorías los objetivos específicos que permiten alcanzar el objetivo principal. Los objetivos de aprendizaje se centran en la adquisición de los conocimientos teóricos requeridos para la concepción de este proyecto, mientras que los objetivos de diseño e implementación buscan llevar a la práctica dicho conocimiento teórico adquirido anteriormente, aportando con ello una experiencia de aprendizaje de mayor categoría.

### 1.2.1. Objetivos de aprendizaje

- **OA.1** Conocer los fundamentos del machine learning y como se aplican a CNNs.
- **OA.2** Conocer los distintos componentes de una CNN y la conexión entre los mismos.
- **OA.3** Comprender implementaciones similares a las planteadas en este proyecto para comprender y analizar las funcionalidades y propiedades que se requieren.
- **OA.4** Aprender como diseñar e implementar CNNs empleando tecnologías de programación de bajo nivel como C++.
- **OA.5** Comprender el diseño e implementación de CNNs empleando paralelización a nivel de CPU mediante OpenMP y C++.
- **OA.6** Mejorar mi aprendizaje sobre diseño e implementación de sistemas heterogéneos usando CUDA.
- **OA.7** Comprender el diseño e implementación de CNNs mediante sistemas heterogéneos usando CUDA.
- **OA.8** Aprender como diseñar e implementar CNNs mediante librerías específicas para este ámbito y de bajo nivel como cuDNN.

### 1.2.2. Objetivos de diseño e implementación

- **ODD.1** Diseñar e implementar CNNs a bajo nivel mediante C++.

- **ODD.2** Diseñar e implementar CNNs a bajo nivel mediante C++ y paralelización a nivel de CPU mediante OpenMP.
- **ODD.3** Diseñar e implementar CNNs a bajo nivel como sistema heterogéneo mediante C++ y CUDA.
- **ODD.4** Diseñar e implementar CNNs mediante la librería de bajo nivel cuDNN.

# Capítulo 2

## Conceptos previos

### 2.1. Machine Learning

El término Machine Learning (ML), en castellano aprendizaje automático, designa el campo de las ciencias de computación que en vez de enfocarse en el diseño de algoritmos explícitos, opta por el estudio de técnicas de aprendizaje. Este enfoque tiene un gran éxito en tareas computacionales donde no es factible diseñar un algoritmo de forma explícita. [11]

En vez de averiguar las distintas reglas a seguir para llegar a una solución, esta alternativa permite simplemente suministrar ejemplos de lo que debería pasar en distintas situaciones, y dejar que la máquina aprenda y extraiga ella misma sus propias conclusiones. De esta forma, el procedimiento en aprendizaje supervisado (el tipo de aprendizaje en que nos centramos) consiste en 'entrenar' con una muestra de  $N$  ejemplos, extraer información de ellos, y posteriormente poder evaluar de forma 'correcta' (bajo un margen de error controlado) otra muestra de  $M$  ejemplos, siendo  $M > N$ . [31]

Este enfoque ha contribuido al avance de áreas como reconocimiento de voz, visión por ordenador, procesamiento de lenguaje natural, etc.

### 2.2. Aprendizaje supervisado

El aprendizaje supervisado [32] se caracteriza por, a partir de una serie de datos de entrada  $X = \{x_1, x_2, \dots\}$  y sus etiquetas asociadas  $Y = \{y_1, y_2, \dots\}$  (cada etiqueta  $y_i \in Y$  representa la clase a la que pertenece su elemento asociado  $x_i \in X$  [33]), entrenar un modelo (un programa informático que aproxima el comportamiento objetivo) con estos. Así, mediante un proceso iterativo, este va aprendiendo de forma que al finalizar dicho entrenamiento el mismo modelo sea capaz de tomar mejores decisiones que antes de comenzarlo.

Suponiendo que nuestra muestra tiene  $N$  datos, tanto  $X$  como  $Y$  se unen para formar lo que se conoce como dataset  $D = \{(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)\}$ .

Para que el aprendizaje sea posible, debe existir una función  $F: X \rightarrow Y$  tal que  $y_i = F(x_i)$  para  $i \in \{1 \dots N\}$ . De esta forma, en función del dataset D, el modelo tratará de encontrar una función G que aproxime a F para dicho conjunto. Además, se suelen aplicar técnicas que permitan una mejor generalización del modelo, expandiendo las capacidades del mismo y permitiendo que su conocimiento pueda ser útil incluso fuera de la muestra de datos inicial. [31]

### **2.3. División de datos en entrenamiento y test**

Suponiendo que, a partir de unos datos de entrada y un modelo, logremos que este los emplee para aprender, normalmente el objetivo final es emplear dicho modelo fuera de esa muestra inicial.

Por ejemplo, en el caso de aprender a montar en bici lo que se suele querer es aprender a montar en cualquier bici, no aprender a usar una bici y cada vez que se quiera cambiar de vehículo tener que volver a empezar dicho aprendizaje.

En el caso de los modelos de aprendizaje automático, un ejemplo simple puede ser, dada una imagen de un animal (que puede ser solo de un perro o de un gato), determinar si es una imagen de un perro o de un gato. Si se entrena un modelo con 200 imágenes, suele ser común que su desarrollador quiera emplear dicho modelo entrenado para distinguir gatos de perros con imágenes que este no vio nunca antes.

Es decir, aunque se entrene un modelo con una muestra de N imágenes, es importante saber que, en la mayoría de los casos, lo que se busca no es un buen rendimiento exclusivamente en la muestra con que se entrenó, sino también en muestras en las que no se entrenó.

Para visualizar la generalización del modelo, el conjunto de datos D se suele dividir en 2 subconjuntos, (entrenamiento y test) de forma que se pueda estimar si realmente 'aprende' o solo memoriza.

Una vez realizada la división, se entrena el modelo con los datos del conjunto de entrenamiento. Cuando se termina el entrenamiento, se accede al conjunto test y se visualiza el rendimiento del modelo sobre el mismo. Como los datos de test no se emplearon en ningún momento, aportan una estimación sobre la generalización del modelo fuera de la muestra con la que se entrenó.

### **2.4. Redes Neuronales Totalmente Conectadas**

Antes de analizar las redes neuronales convolucionales (CNNs), tiene sentido empezar por las redes neuronales 'clásicas' o totalmente conectadas. Se define una red neuronal como un programa o modelo de machine learning que toma decisiones de forma similar al cerebro humano, empleando para ello procesos que imitan a los de las neuronas biológicas [34].

Una red neuronal cuenta con una serie de neuronas artificiales organizadas por capas, y se caracteriza por tener una capa de entrada, una o varias capas ocultas y una capa de salida.

#### 2.4.1. Neurona

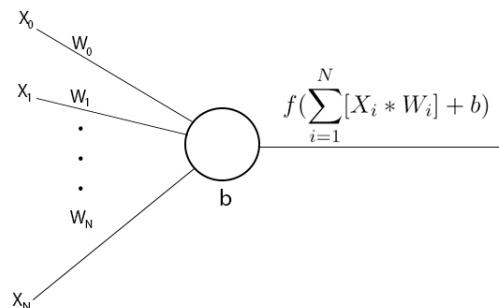


Figura 2.1: Esquema de una neurona

Una neurona parte de una serie de datos de entrada  $X = \{x_1, x_2, \dots, x_N\}$  tal que cada  $x_i \in X$  se encuentra asociado a un peso  $w_i \in W$  y obtiene un único dato de salida o resultado operando con los datos de entrada y sus pesos.

En la Figura 2.1 se muestra cómo esta los emplea para realizar una suma ponderada y posteriormente añadir un sesgo  $b$ , además de aplicar una función (que llamaremos función de activación)  $f$  sobre el resultado obtenido para obtener la salida.

### 2.4.2. Estructura por capas

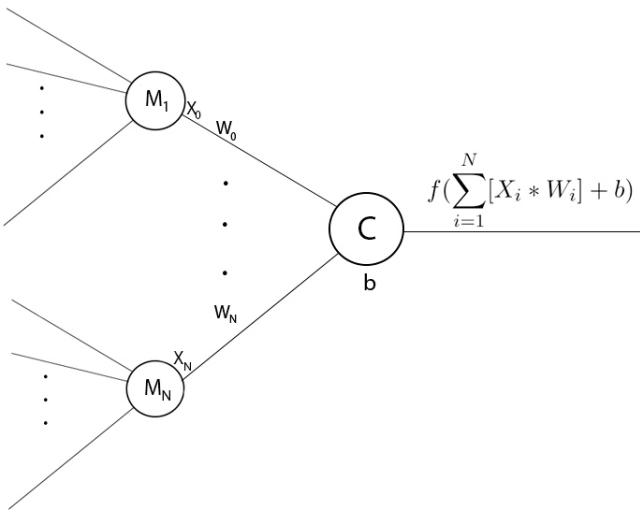


Figura 2.2: Esquema de una capa de neuronas

Las neuronas se suelen agrupar por capas, de tal forma que la salida de una compone la entrada de la siguiente, formando así modelos más sofisticados. En la Figura 2.2, se puede ver cómo la salida de la neurona  $C$  se obtiene tomando como entrada la salida de las neuronas  $M_1, M_2, \dots, M_N$ . En este proyecto se desarrollarán redes neuronales para tareas de clasificación multiclase. Para clasificar  $N$  clases distintas, la capa de salida tendrá  $N$  neuronas. De esta forma, nuestra red totalmente conectada tendrá una capa de entrada (para recibir los datos de entrenamiento), una capa de salida, y las capas intermedias entre estas dos recibirán el nombre de capas ocultas.

### 2.4.3. Funciones de activación

Una función de activación, en el contexto de las redes neuronales, es una función matemática  $f$  que se aplica al resultado de sumar el producto escalar del vector de entrada y el vector de pesos con el sesgo para obtener la salida de una neurona. Su objetivo consiste en introducir no linealidad en el modelo, permitiendo que la red aprenda e identifique patrones complejos en los datos. Sin no linealidad, una red neuronal se comportaría esencialmente como un modelo de regresión lineal, independientemente del número de capas que tenga [35]. Introduciremos a continuación una serie de funciones de activación de interés:

## Función ReLU

$$ReLU(x) = \max(0, x) \quad (2.1)$$

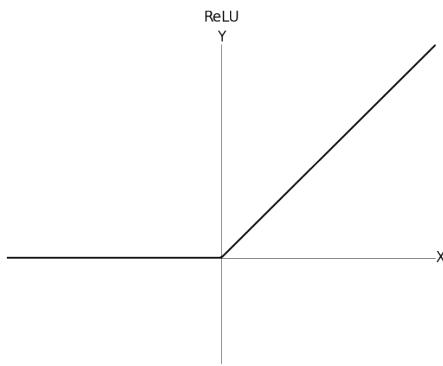


Figura 2.3: Imagen de la función de activación ReLU

A cambio de un bajo coste computacional, la función de activación ReLU (véase la Figura 2.3) aporta no linealidad a la neurona, permitiendo a esta aprender funciones de mayor complejidad.

Como su gradiente es 0 ó 1 (0 para valores negativos, 1 para valores positivos), evita una reducción excesiva del mismo para valores positivos, mitigando así el problema del desvanecimiento del gradiente, caracterizado por la presencia de gradientes muy pequeños en retropropagación (se analizará esto con detalle en secciones posteriores) y provocar un aprendizaje lento [36].

Además, tal y como sugieren los expertos, se empleará ReLU como función de activación en las capas intermedias de los modelos a implementar [37] [38].

## Función Sigmoide

$$\text{sigmoide}(x) = \frac{1}{1 + e^{-x}} \quad (2.2)$$

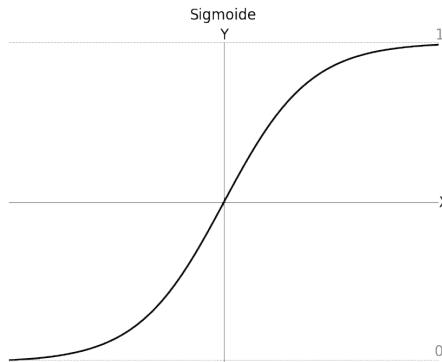


Figura 2.4: Imagen de la función de activación Sigmoide

Se define la función de activación Sigmoide como una función interesante en el ámbito de la clasificación binaria, pues es caracterizada por transformar un valor de entrada en una salida perteneciente al rango [0-1] , tal y como se muestra en la Figura 2.4.

Aunque sea monótona creciente y diferenciable en todos los puntos, tiende a saturarse con valores extremos (positivos o negativos). Por tanto, su aplicación dependerá del caso concreto a tratar. [39]

#### 2.4.4. Capa SoftMax

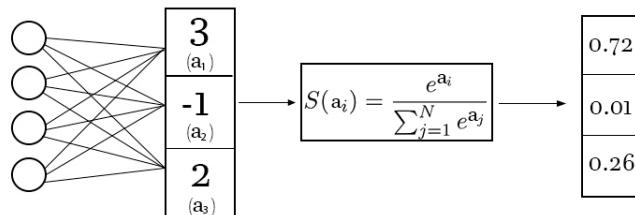


Figura 2.5: Imagen de la función de activación SoftMax

La función SoftMax se suele emplear en la capa de salida de la red. Se caracteriza por, para N entradas, producir N salidas con valores en el rango [0-1] que mantienen la proporción de entrada y cuya suma es 1. Por tanto, cada salida i se puede interpretar como la probabilidad de pertenencia a la clase i, siendo especialmente útil en clasificación multiclas. [40]

De esta forma, en la Figura 2.5 se muestra un ejemplo de clasificación multiclas con 3 clases distintas.

Para una entrada dada, el modelo estima que esta pertenece a la clase 0, pues según este, dicha entrada presenta un 72 % de probabilidad de pertenecer a la clase 0, un 1 % a la clase 1, y un 26 % a la clase 2.

#### 2.4.5. Tipos de codificaciones de etiquetas

En el campo del machine learning existen varios tipos de codificaciones. Así, para codificar 3 clases distintas se podrían codificar o bien mediante  $\{1, 2, 3\}$ , o mediante  $\{100, 010, 001\}$  (codificación one-hot [41]), por ejemplo. En este proyecto se empleará la codificación one-hot por simplicidad y seguir el mismo enfoque seguido en la gran mayoría de bibliografía online.

#### 2.4.6. Función de error o pérdida

En modelos de aprendizaje automático se suele emplear una función de optimización iterativa como descenso del gradiente (se muestra en el siguiente apartado) para, a partir de una función de error y unos datos de entrada, estimar el error del modelo sobre los mismos. Esta estimación del error se puede emplear para ir reduciendo el error iteración a iteración y así ir aprendiendo.

##### Entropía Cruzada

$$E(y, \hat{y}) = - \sum_{i=1}^N [y_i * \log(\hat{y}_i)] \quad (2.3)$$

En los modelos a desarrollar en este proyecto se empleará la Entropía Cruzada como función de error por ser una métrica empleada en aprendizaje automático para medir las prestaciones de un modelo de clasificación. La pérdida o error se mide como un valor en el rango  $[0-1]$ , siendo 0 un modelo perfecto y 1 otro totalmente erróneo. [42]

En la fórmula 2.3 se muestra la fórmula para el cálculo de la entropía cruzada, donde  $H$  es el número de clases al que puede pertenecer cada dato de entrada  $x_i \in X$ ,  $y_i$  representa la etiqueta real de la entrada  $x_i$  empleada, e  $\hat{y}_i$  representa la etiqueta que el modelo predijo para dicha entrada  $x_i$ .

##### Importancia de la función de error en el aprendizaje

Para visualizar el papel de la función de error en el aprendizaje de un modelo, supondremos que los datos actuales cuentan con  $N=3$  clases tal que, para un  $x_i \in X$  dado y empleando la codificación One-Hot para las etiquetas, se tiene que  $y_i = [0, 0, 1]$ .

$$E(y, \hat{y}) = - \sum_{i=1}^N [y_i * \log(\hat{y}_i)] = 0 + 0 + \log(0.1) = -\log(0.1) = 2.3 \quad (2.4)$$

Como la configuración inicial del modelo es desconocida, se supone que sus predicciones iniciales para  $x_i$  son  $\hat{y}_i = [0.6, 0.3, 0.1]$ . En este caso,  $x_i$  pertenece a la clase 3 pero el modelo predice que pertenece a la clase 1, pues 0.6 se interpreta como la probabilidad de pertenecer a la clase 1 y es la probabilidad más grande de todo  $\hat{y}_i$ .

Por tanto, su función de error indicaría que el error obtenido viene dado por la fórmula 2.4.

$$E(y, \hat{y}) = - \sum_{i=1}^N [y_i * \log(\hat{y}_i)] = 0 + 0 + \log(0.6) = -\log(0.6) = 0.5 \quad (2.5)$$

Tras entrenar el modelo durante varias iteraciones,  $y_i$  permanece constante, pero  $\hat{y}_i$  se modifica a  $\hat{y}_i = [0.1, 0.3, 0.6]$ , de forma que el nuevo error obtenido vendría dado por la fórmula 2.5, y ahora el modelo realizaría una predicción correcta sobre la clase de  $x_i$ , aunque todavía le quedaría margen de mejora pues el error es de 0.5 y este se puede minimizar.

De esta forma, se visualiza como, a lo largo del entrenamiento de un modelo, este se centra en reducir el error obtenido, y como consecuencia de ello va modificando sus predicciones, de tal forma que se vayan ajustando a lo especificado por sus datos de entrada y etiquetas asociadas.

#### 2.4.7. Descenso del gradiente

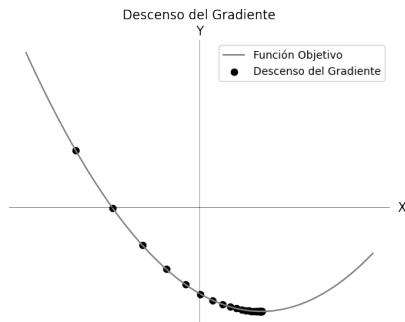


Figura 2.6: Ejemplo de funcionamiento del descenso del gradiente

Tal y como se comentaba anteriormente, el descenso del gradiente es un método de optimización iterativo que busca el mínimo en una función diferenciable. En la figura 2.6 se muestra un ejemplo del mismo, donde la función objetivo a minimizar hace referencia a la función de pérdida, y cada punto sobre ella representa una iteración del algoritmo, indicando cómo el error del modelo sobre la muestra se va reduciendo con cada iteración.

Su nombre viene del término 'gradiente', siendo este una generalización multivariable de la derivada y denominado por el símbolo  $\nabla$ . Para una función

f y un punto p, este indica la dirección del máximo incremento en la misma. El descenso del gradiente usa esta información para, una vez obtenido el gradiente, desplazarse en dirección contraria, es decir, en dirección del mínimo. Además, la distancia que se recorre en cada iteración viene dada por un hiperparámetro alpha denominado “learning rate” [43] [44] [45].

A diferencia de los parámetros de un modelo (pesos y sesgos), sus hiperparámetros son establecidos por el usuario y no se “aprenden” durante el entrenamiento del modelo.

#### 2.4.8. Inicialización de pesos

Tal y como sugieren los expertos, se inicializarán los pesos mediante la “inicialización He” o “inicialización Kaiming He”. Esta consiste en, para un peso  $w$ , inicializarlo según una distribución gaussiana con una media de 0.0 y una desviación típica de  $\sqrt{\frac{2}{N\_in}}$ , siendo  $N\_in$  el número de neuronas en la capa de entrada [46] [47] [48] [37] [38].

#### 2.4.9. Inicialización de sesgos

De la misma forma, se vuelve a seguir la bibliografía, indicando esta vez una inicialización de sesgos a 0.0 [49] [50].

### Actualización de parámetros

El procedimiento para entrenar una red neuronal consiste en, para una entrada  $x_i$  y una etiqueta asociada  $y_i$ , emplear  $x_i$  para realizar una predicción  $\hat{y}_i$  (Propagación hacia delante de  $x_i$  por la red) que posteriormente se podrá comparar con  $y_i$  mediante una función de error  $H(x)$  y obtener una medida de lo buena o mala que fue la misma. Una vez obtenido dicho “error”, se aplica el algoritmo del descenso del gradiente para obtener el gradiente de la pérdida respecto a cada parámetro de la red (Retropropagación) [42].

$$W_i^{t+1} = W_i^t - \alpha * \frac{\partial H(x)}{\partial W_i} \quad (2.6)$$

$$b_{t+1} = b_t - \alpha * \frac{\partial H(x)}{\partial b_t} \quad (2.7)$$

Así, se actualizarán los parámetros de la red neuronal (pesos y sesgos) según las fórmulas 2.6 y 2.7. En ellas,  $W_i^t$  y  $b_t$  indican los valores del peso  $W_i$  y bias b en el instante o iteración t, de la misma forma que  $W_i^{t+1}$  y  $b_{t+1}$  representan los valores de los mismos en el instante t+1 [51].

Es decir, dada una iteración t y unos datos de entrada X, primero se realiza la

propagación hacia delante de los mismos a lo largo del modelo para obtener  $\hat{Y}$  o la predicción de los mismos. Una vez obtenida  $\hat{Y}$ , se emplea la función de error para obtener el error del modelo sobre los datos de entrada. Con ello, se realiza la retropropagación del gradiente de la pérdida o error a lo largo del modelo, de forma que cada parámetro del mismo obtenga el gradiente de la pérdida respecto a él. Una vez cada parámetro tiene asociado su gradiente respecto a la pérdida, lo emplea junto con  $\alpha$  para actualizar su valor, dirigiéndose en sentido contrario al gradiente con una magnitud igual a  $\alpha$ , permitiendo así la reducción del error en la iteración  $t+1$ . Los parámetros se actualizan una vez por iteración, tras realizar la retropropagación.

### Descenso del gradiente estocástico

---

#### Algorithm 1 Descenso del gradiente estocástico [52]

---

```

for época  $p \in \{0, \dots, P - 1\}$  do
    Desordenar datos de entrenamiento.
    for minibatch  $\in [m_{\text{inicio}}, \dots, m_{\text{fin}}]$  do
        Obtener datos del minibatch.
        Realizar propagación hacia delante.
        Obtener error mediante función de error.
        Realizar retropropagación y obtener gradiente de la pérdida
        respecto a cada parámetro del modelo.
        Actualizar parámetros.
    end for
end for

```

---

El descenso del gradiente estocástico es una variante que sustituye el gradiente real por una estimación del mismo, logrando reducir la carga computacional y tiempo de entrenamiento a cambio de una menor tasa de convergencia [53] [54].

Se caracteriza por, en cada época (un paso completo a través de todo el conjunto de datos de entrenamiento durante el proceso de aprendizaje [53]), dividir el conjunto de entrenamiento en varios subconjuntos aleatorios y disjuntos entre ellos (minibatch).

Para cada minibatch, se realiza la propagación hacia delante, el cálculo del error, la retropropagación y la actualización de parámetros en ese orden.

## 2.5. Redes Neuronales Convolucionales

Las redes neuronales convolucionales [10] surgen como una adaptación de las redes neuronales totalmente conectadas para el caso concreto de tratamiento de imágenes. Ambas comparten la idea de contar con una serie

de neuronas artificiales distribuidas por capas de forma de la salida de la capa  $i$  sea la entrada de la capa  $i+1$ . Su principal diferencia reside en los datos de entrada y en las operaciones que se realizan en cada capa. Por ello, en secciones posteriores, se introducirán en detalle dos nuevas capas (capa convolucional y capa de agrupación máxima), cada una caracterizada por realizar un tipo de operación distinto.

### 2.5.1. Introducción a CNNs

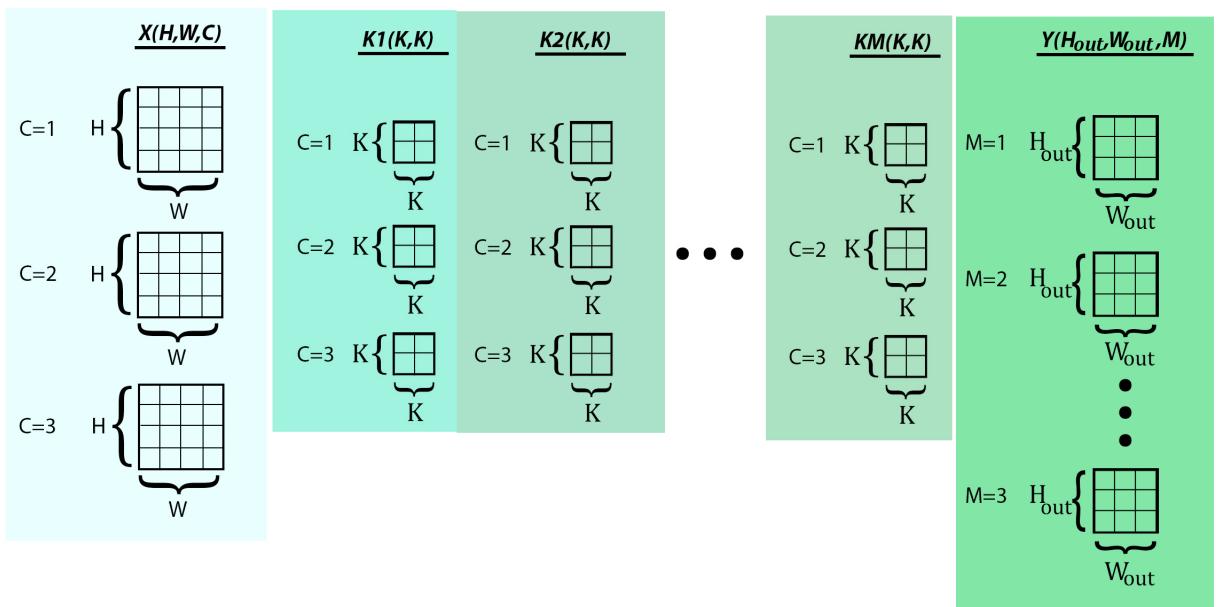


Figura 2.7: Esquema de las dimensiones en una capa convolucional de una red neuronal convolucional

Las redes neuronales convolucionales (CNN, Convolutional Neural Network) se distinguen por su capacidad para trabajar con volúmenes de datos 3D. En este proyecto, se emplearán imágenes RGB como entrada para cada CNN. De este modo, cada imagen RGB puede interpretarse como un conjunto de capas 2D. De esta manera, si definimos  $X$  como un volumen 3D que representa una imagen RGB,  $X$  estará compuesto por 3 imágenes 2D, cada una correspondiente a un color (rojo, verde y azul). Adicionalmente, las dimensiones de  $X$  se definen como  $H$  (filas),  $W$  (columnas) y  $C$  (canales de profundidad), o, de forma abreviada,  $X(H,W,C)$  (véase la Figura 2.7). De manera equivalente, en una capa convolucional, los kernels de pesos se definen como  $\{K_1, K_2, \dots, K_M\}$  o  $\{K_1(K, K), K_2(K, K), \dots, K_M(K, K)\}$ , donde

cada uno de ellos tendrá K filas y K columnas.

Finalmente, en cada capa de una red neuronal convolucional, se define el volumen de salida como Y o  $Y(H_{out}, W_{out}, M)$ , y estará compuesto de  $H_{out}$  filas,  $W_{out}$  columnas, y M canales de profundidad, (uno por kernel de pesos).

### 2.5.2. Estructura por capas

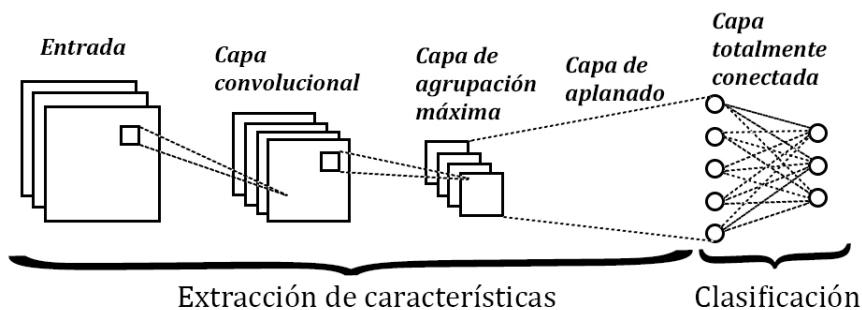


Figura 2.8: Estructura por capas en una red neuronal convolucional

La arquitectura básica de una CNN se compone de dos partes principales (véase la Figura 2.8):

- **Extracción de Características:** Se define como extracción de características, al proceso que emplea múltiples pares de capas convolucionales y de agrupamiento máxima. Estas, se emplean para reducir y resumir las características de especial importancia, presentes en el conjunto de datos original.
- **Clasificación:** Una vez obtenido el resultado del proceso de extracción de características, en la etapa de clasificación, se emplea para predecir la clase de las imágenes de entrada. La capa totalmente conectada, realiza la clasificación final, combinando la información procesada en las etapas anteriores.

A continuación, se define cada una de las capas involucradas en una red neuronal convolucional:

- **Capa convolucional:** La capa convolucional, extrae características de la entrada X mediante la operación de convolución, y esta se realiza con un filtro de tamaño KxK. De esta manera, genera un mapa de características de salida (Y), que preserva la relación espacial entre los píxeles, y resalta elementos clave como bordes y esquinas.
- **Capa de agrupación máxima:** Se define como capa de agrupación máxima, aquella centrada en reducir la dimensionalidad de la entrada.

Dicha reducción se obtiene al seleccionar, dentro de una región definida de la entrada, el valor máximo, preservando así las características más relevantes. Esta operación no solo disminuye la complejidad computacional, sino que también contribuye a que la red sea más invariante a pequeñas traslaciones en la entrada.

- **Capa de aplanado:** Se define como capa de aplanado aquella que transforma un volumen tridimensional en un vector unidimensional. De este modo, facilita la transición de las características extraídas por capas convolucionales y de agrupamiento hacia capas totalmente conectadas.
- **Capa totalmente conectada:** Se define como capa totalmente conectada aquella en la que cada neurona está conectada a todas las neuronas de la capa anterior y posterior. De este modo, permite la combinación de todas las características extraídas para realizar la clasificación multiclas que se llevará a cabo en este proyecto.

### 2.5.3. Capa convolucional

#### Componentes

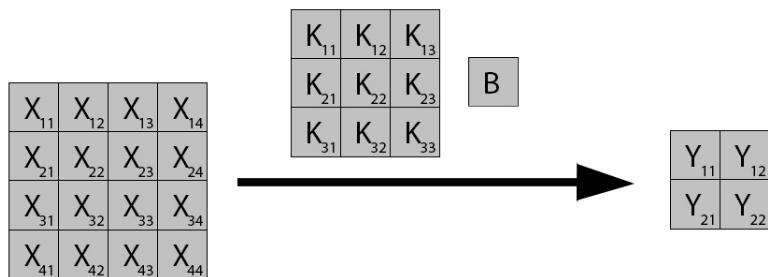


Figura 2.9: Componentes en una capa convolucional

Una capa convolucional parte de un volumen de entrada  $X$ , un kernel de filtros  $K$ , un sesgo  $B$  y una función de activación para, mediante una convolución, obtener un volumen de salida  $Y$  [55] [56].

### Propagación hacia delante

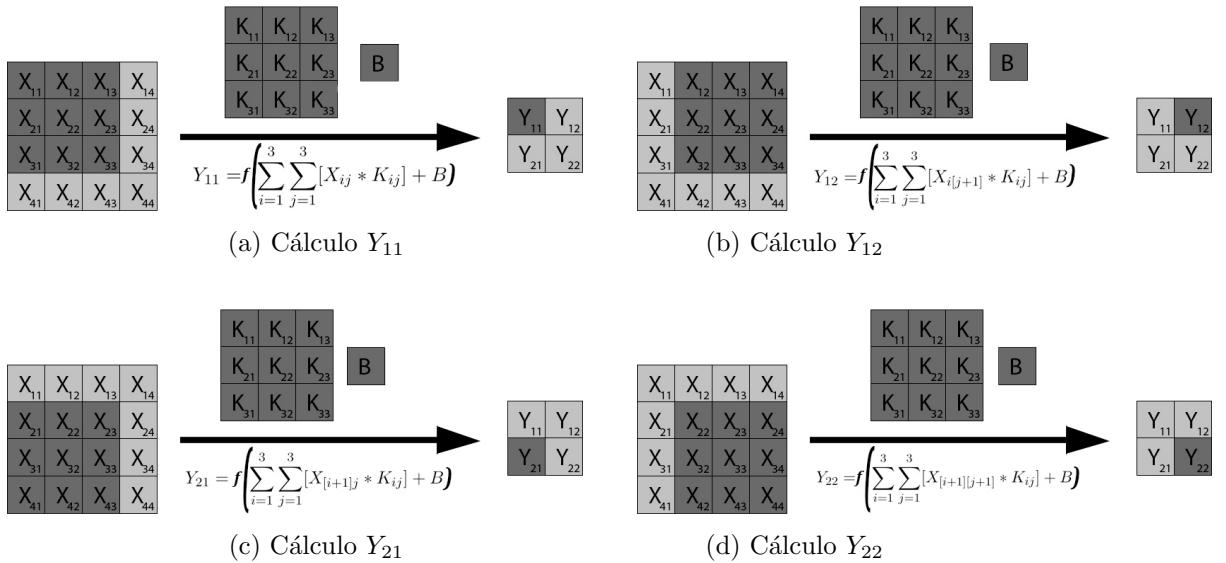


Figura 2.10: Propagación hacia delante en una capa convolucional

Una convolución entre 2 volúmenes de datos  $X$  y  $K$ , consiste en “deslizar  $K$  sobre  $X$ ” tal y como se muestra en la Figura 2.10. De esta forma, en cada “paso” se recorren ambos volúmenes, multiplicando los elementos de  $X$  y  $K$  que se encuentren superpuestos en la misma posición. Posteriormente, se suma cada resultado obtenido, además de un sesgo  $B$  y finalmente se aplica una función de activación [55].

Por simplicidad inicial, en la Figura 2.10 se emplea un volumen  $X$  con un solo canal de profundidad. Sin embargo, este no es el caso común. Por tanto, se denotará como  $X_{ij}^c$  al elemento de  $X$  que se encuentre en la posición  $(i,j)$  del canal de profundidad  $c$ . De la misma forma, se definirá  $K_{ij}^c$  como el peso  $k \in K$  que se encuentre en la posición  $(i, j)$  del canal de profundidad  $c$  del kernel de pesos  $K$  [56].

## Propagación hacia delante de X con varios canales de profundidad

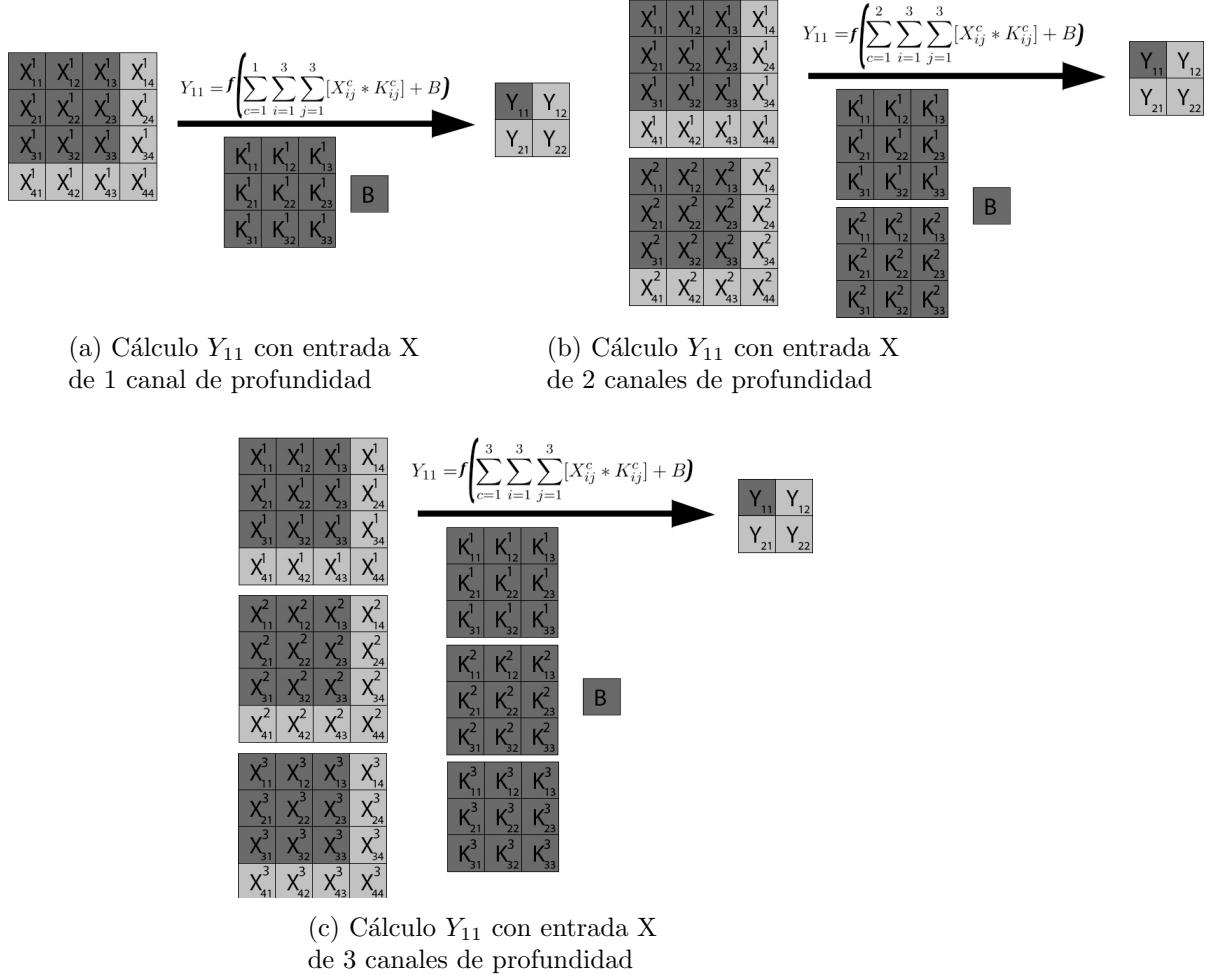


Figura 2.11: Propagación hacia delante en una capa convolucional con varios canales de profundidad

De esta forma, en la Figura 2.11 se muestra como una convolución con  $C$  canales de profundidad se descompone en la suma de  $C$  convoluciones con un canal de profundidad (véase la fórmula 2.8).

Por último, en cada “paso” del “deslizamiento” se suma un solo sesgo y se aplica una sola vez la función de activación, independientemente del número de canales de profundidad que presente la entrada X.

$$\text{convolucion\_C\_canales}(X, K) = \text{convolucion\_1\_canal}(X^1, K^1) + \text{convolucion\_1\_canal}(X^2, K^2) + \dots + \text{convolucion\_1\_canal}(X^C, K^C) \quad (2.8)$$

### Propagación hacia delante de X con varios filtros

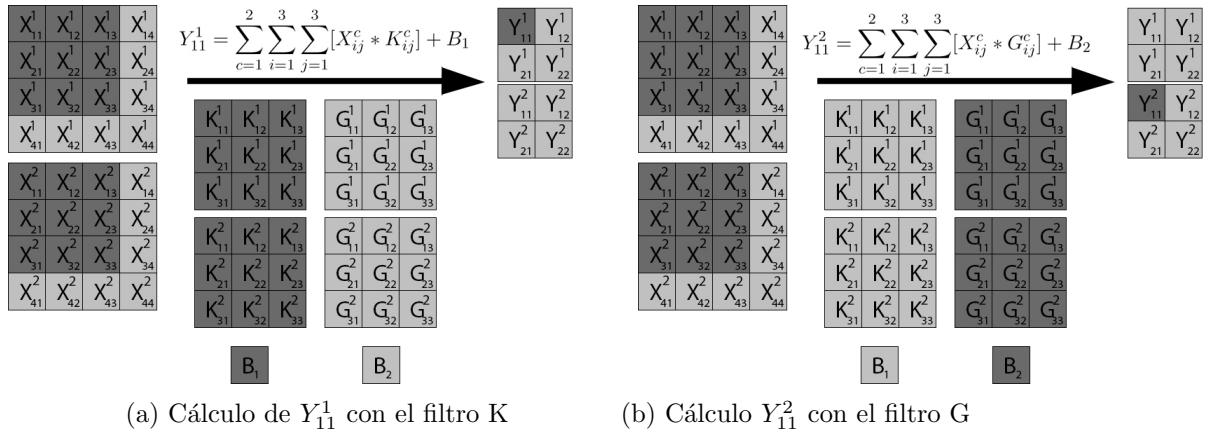


Figura 2.12: Propagación hacia delante en una capa convolucional con varios filtros

Cada convolución entre dos volúmenes 3D produce un volumen de salida 2D. Por tanto, al aplicar M convoluciones entre un volumen de entrada X y una serie de filtros  $K=\{K_1, K_2, \dots, K_M\}$ , se obtendrá un volumen 3D de salida con tantas capas de profundidad como convoluciones se aplicaron (M). En la Figura 2.12 se observa como al aplicar M=2 convoluciones sobre la misma entrada X (una con el filtro K y otra con el filtro G) se obtiene un volumen de salida con 2 capas de profundidad [56].

### Relleno o “Padding”

En la figura 2.10 se visualiza como al realizar una convolución entre un volumen X con dimensiones 1x4x4 y un kernel de pesos K con dimensiones 1x3x3, el resultado obtenido es un volumen Y de dimensiones 1x2x2. La reducción de dimensionalidad es un problema pues afecta directamente al número de convoluciones que se pueden aplicar sobre un volumen. Por tanto, el “relleno” o “padding” [57] se aplica antes de realizar una convolución y es una técnica empleada para conservar las dimensiones espaciales de un volumen de entrada X, expandiendo cada canal del mismo tal y como se muestra en la figura 2.13 [58].

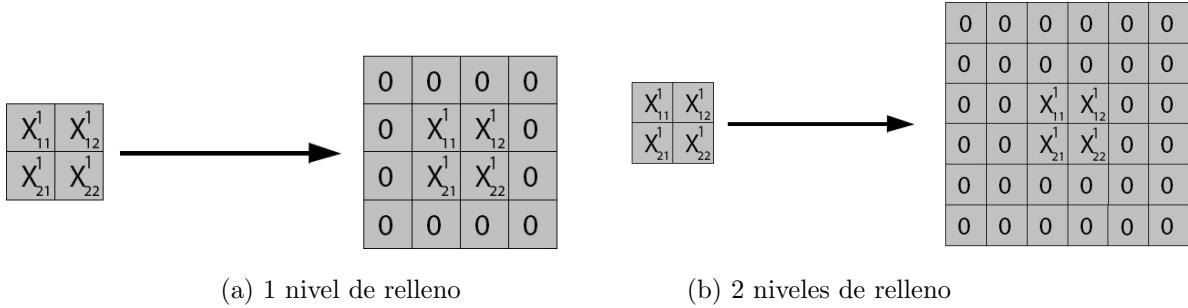


Figura 2.13: Relleno sobre un volumen de entrada X

De esta forma, un relleno a un nivel sobre un volumen añadirá sobre el mismo dos filas y dos columnas con valores igual a 0. Un relleno a dos niveles añadirá cuatro filas y columnas con valores igual a cero, ..., un relleno a n niveles añadirá  $2^n$  filas y columnas con valores igual a 0.

### Relleno completo

Se denomina como relleno completo [59] [60] aquel que asegura que cada valor o casilla de X sea visitada el mismo número de veces que el resto en una operación de convolución.

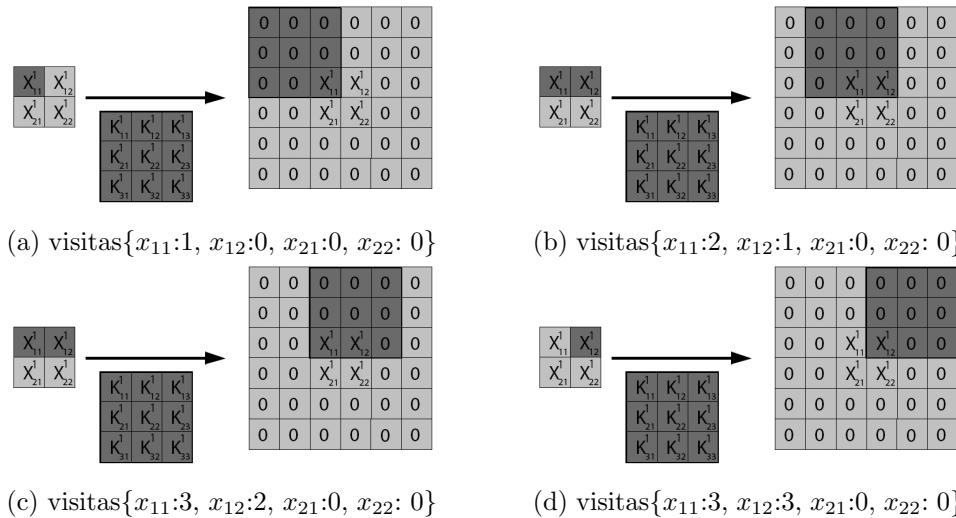
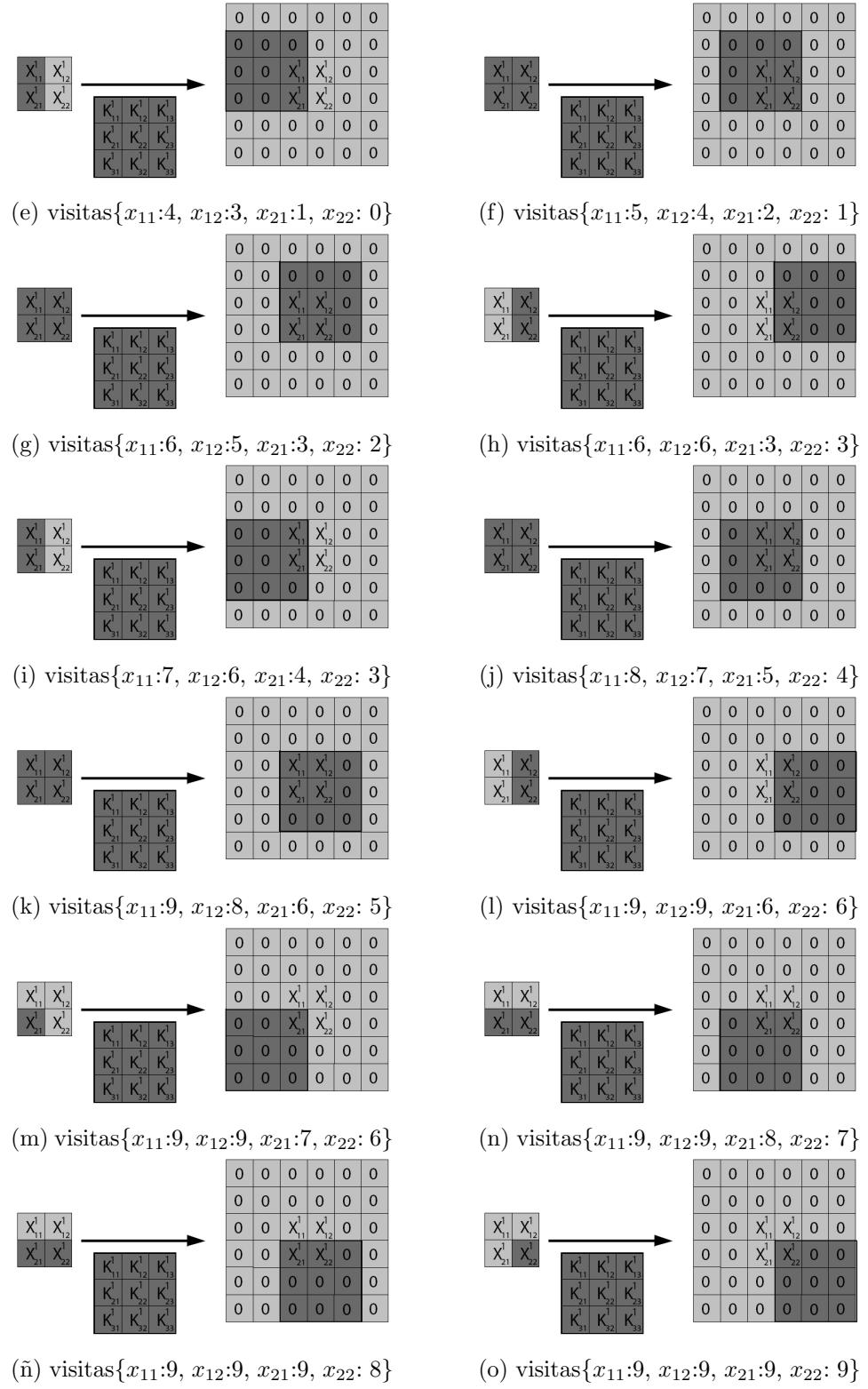


Figura 2.14: Relleno sobre un volumen de entrada X

Figura 2.14: Convolución sobre  $X$  con relleno completo

Tal y como se muestra en la figura 2.14, se realiza un relleno completo pues en la convolución entre X y K se accede a cada valor de X el mismo número de veces (9).

#### 2.5.4. Capa de agrupación máxima

##### Componentes

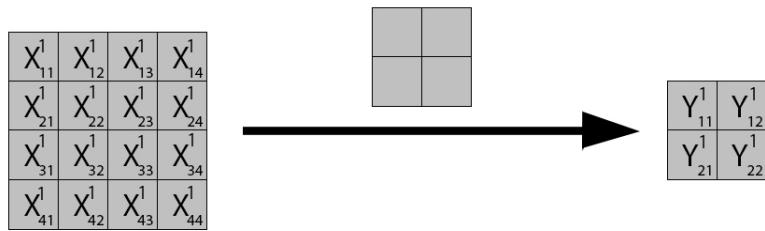


Figura 2.15: Componentes en una capa de agrupación máxima

Al igual que las capas convolucionales, las capas de agrupación máxima también presentan una “ventana” que se irá deslizando por el volumen de entrada. Sin embargo, el resultado en cada iteración viene dado por el valor máximo contenido en ella. Por tanto, no presenta parámetros asociados.

##### Propagación hacia delante

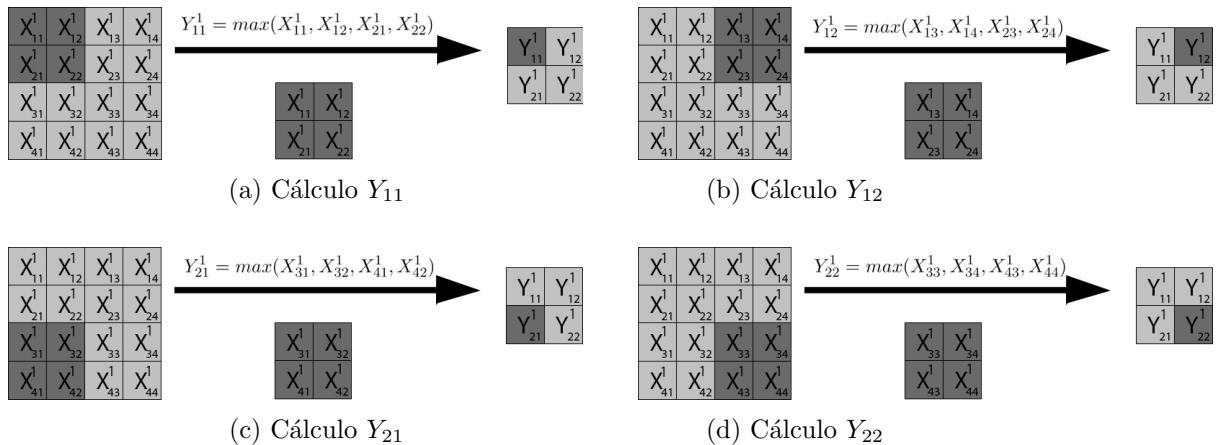


Figura 2.16: Propagación hacia delante en una capa de agrupación máxima

A diferencia de las capas convolucionales, las capas de agrupación máxima no comparten regiones del volumen de entrada entre distintas iteraciones. Este diferente comportamiento se puede comprobar en la figura 2.16 con lo que se describe en la figura 2.10.

### Propagación hacia delante de X con varios canales de profundidad

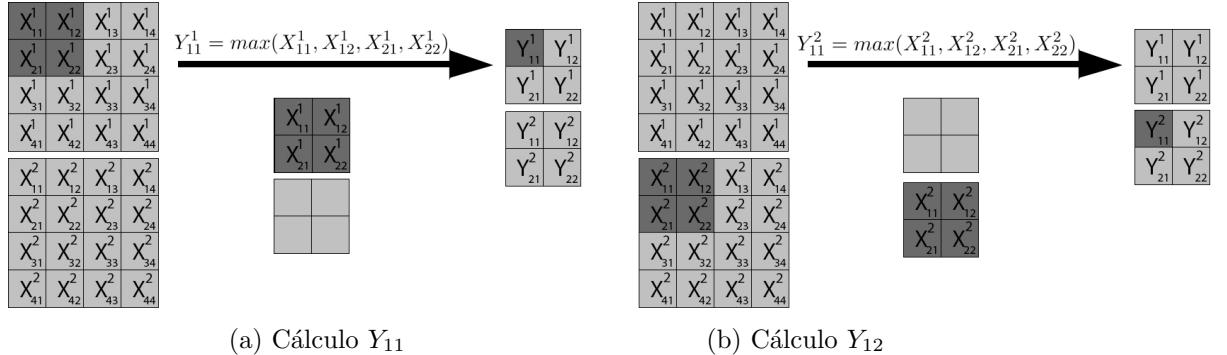


Figura 2.17: Propagación hacia delante en una capa de agrupación máxima

Tal y como se muestra en la figura 2.17, cada “ventana” se desliza sobre el volumen de entrada en un canal de profundidad distinto. Por tanto, al igual que en capas convolucionales, si el volumen de entrada  $X$  cuenta con  $C$  canales de profundidad, la ventana asociada a la capa de agrupación máxima también contará con  $C$  canales de profundidad y cada ventana solo afectará a  $X$  en su respectivo canal. Es decir, para la subventana del canal de profundidad  $c \in C$ , esta solo trabajará sobre la imagen 2D de  $X$  asociada al canal  $c$ .

### Retropropagación en capa de agrupación máxima

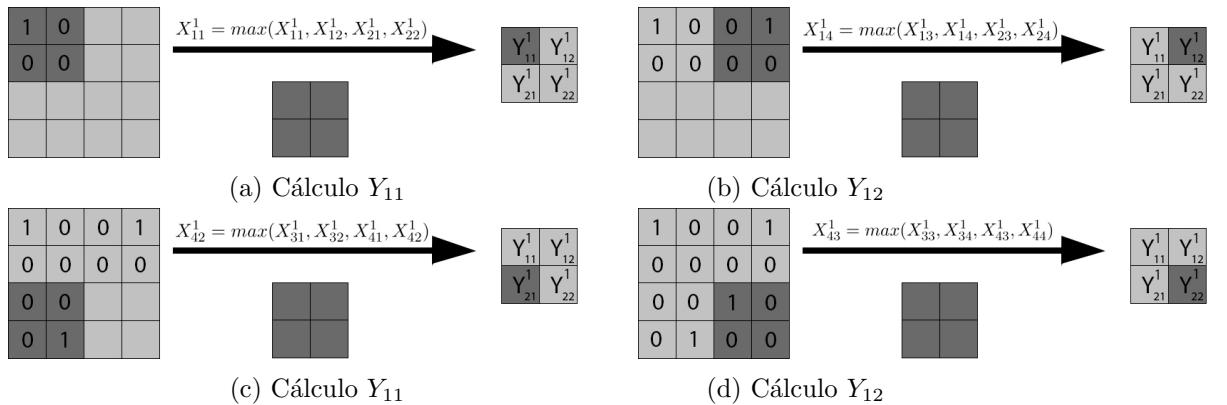


Figura 2.18: Retropropagación en capa de agrupación máxima

En la Figura 2.18 se muestra un ejemplo de retropropagación en una capa de agrupación máxima. Para calcular el gradiente respecto a la entrada ( $X$ ) en una capa  $i$  de agrupación máxima, suponemos que ya se conoce el

gradiente de la pérdida respecto a la salida ( $\frac{\partial E}{\partial Y}$ ) de dicha capa i (se muestra en detalle su cálculo en secciones posteriores). Por tanto, el volumen Y en este caso contendrá dicho gradiente.

Tomando el ejemplo de la Figura 2.18 (a), se entiende que cuando se realizó la propagación hacia delante de dicha capa i,  $Y_{11}^1$  tomó el valor de  $X_{11}^1$ , pues fue el máximo de entre todos los valores de la ventana (formada por  $X_{11}^1$ ,  $X_{12}^1$ ,  $X_{21}^1$ , y  $X_{22}^1$ ). Por tanto, dado que en la propagación hacia delante se obtuvo  $Y_{11}^1 = X_{11}^1$ , tiene sentido que el gradiente de  $Y_{11}^1$  respecto a  $X_{11}^1$  sea 1, pero 0 respecto al resto de valores de X, pues  $X_{11}^1$  fue el único valor en influir sobre  $Y_{11}^1$ . De este modo, el gradiente de la pérdida respecto a la región de X afectada por la ventana  $\{X_{11}^1, X_{12}^1, X_{21}^1, X_{22}^1\}$ , se puede calcular como se muestra en las fórmulas 2.9, 2.10, 2.11, y 2.12:

$$\frac{\partial E}{\partial X_{11}^1} = \frac{\partial E}{\partial Y} * \frac{\partial Y}{\partial X_{11}^1} = \frac{\partial E}{\partial Y} * 1 = \frac{\partial E}{\partial Y} \quad (2.9)$$

$$\frac{\partial E}{\partial X_{12}^1} = \frac{\partial E}{\partial Y} * \frac{\partial Y}{\partial X_{12}^1} = \frac{\partial E}{\partial Y} * 0 \quad (2.10)$$

$$\frac{\partial E}{\partial X_{21}^1} = \frac{\partial E}{\partial Y} * \frac{\partial Y}{\partial X_{21}^1} = \frac{\partial E}{\partial Y} * 0 \quad (2.11)$$

$$\frac{\partial E}{\partial X_{22}^1} = \frac{\partial E}{\partial Y} * \frac{\partial Y}{\partial X_{22}^1} = \frac{\partial E}{\partial Y} * 0 \quad (2.12)$$

Como el resultado obtenido en cada iteración solo depende del valor máximo de la ventana, tiene sentido que la derivada de la salida Y respecto a la entrada X sea igual a 1 solo en dicho caso y 0 en el resto [61] [62].

### 2.5.5. Capa de aplanado

La capa de aplanado tiene como objetivo la creación de un “enlace” entre las capas de convolución y agrupación máxima, con las capas totalmente conectadas. Esto se debe a que, como se mencionó anteriormente, las capas convolucionales y de agrupación máxima trabajan con volúmenes de datos 3D. Sin embargo, las capas totalmente conectadas trabajan con arrays 1D como entrada. Por tanto, la capa de aplanado se encarga de realizar esta conversión de volumen 3D a array 1D y viceversa.

### Propagación hacia delante

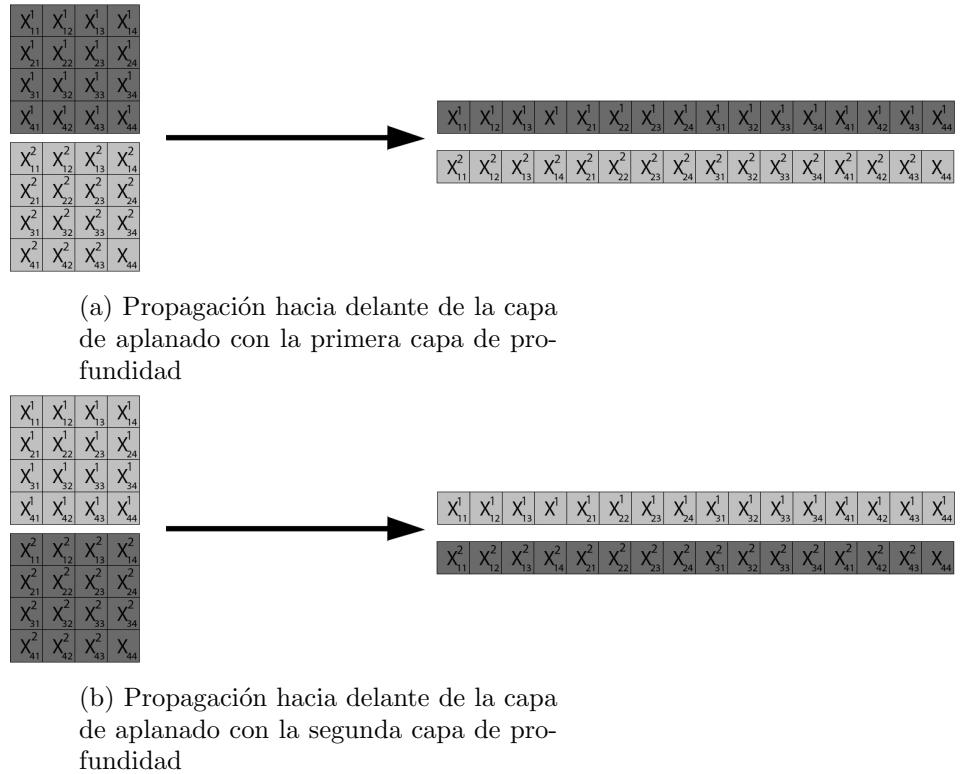


Figura 2.19: Propagación hacia delante en una capa de aplanado

En la propagación hacia delante, se parte de un volumen de entrada 3D para, mediante un “aplanado”, convertirlo en un vector 1D que pueda ser usado como entrada para una capa totalmente conectada, tal y como se muestra en la Figura 2.19.

Como solo se modifica la forma en la que se agrupan los datos, no requiere la presencia de parámetros [63].

### Retropropagación

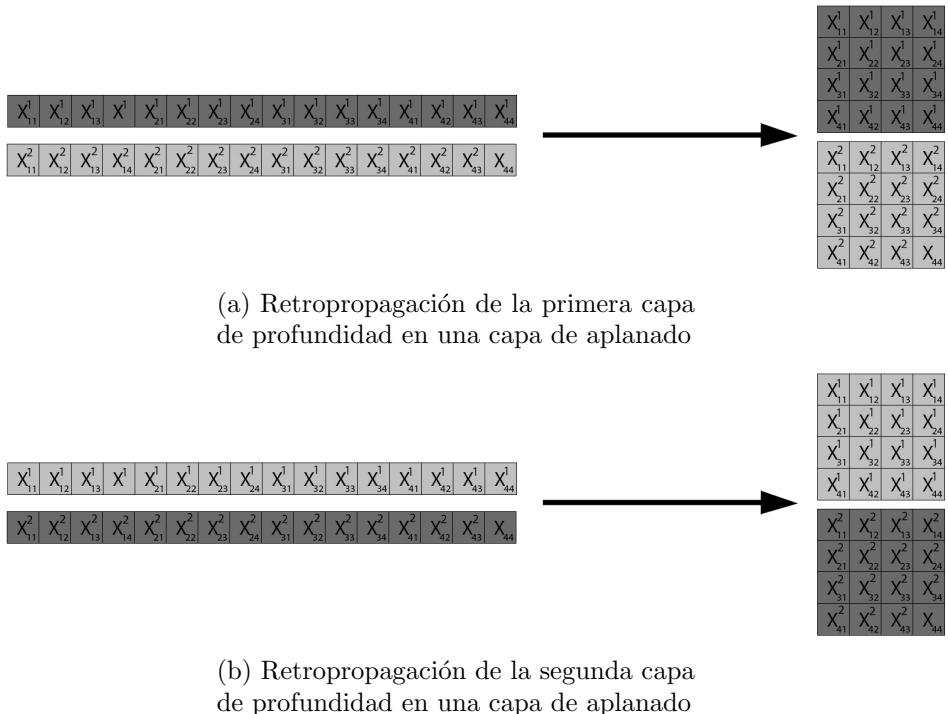


Figura 2.20: Retropropagación en una capa de aplanado

En la retropropagación de una capa de aplanado, se parte de un array 1D y se convierte en un volumen 3D que pueda ser usado para seguir la retropropagación en capas anteriores convolucionales o de agrupación máxima [63].

## 2.6. OpenMP

OpenMP, es una interfaz de programación de aplicaciones (API), que proporciona directivas de compilador, y rutinas de biblioteca, facilitando así la programación paralela en sistemas con memoria compartida. Diseñada para lenguajes como C, C++ y FORTRAN, OpenMP permite a los desarrolladores escribir código que se ejecute de manera concurrente en múltiples núcleos o procesadores, optimizando así el rendimiento de las aplicaciones en entornos de multiprocesamiento. [64].

A continuación se presentan algunas de sus principales directivas [65]:

- **Para uso compartido de trabajo paralelo:**

- **parallel:** Define una región paralela. Es decir, el código que ejecutarán varios subprocessos en paralelo.
- **for:** Permite que el trabajo realizado en un bucle **for** dentro de una región paralela se divida entre subprocessos.
- **sections:** Identifica las secciones de código que se van a dividir entre todos los subprocessos.
- **single:** Permite especificar que se debe ejecutar una sección de código en un único subprocesso, (no necesariamente en el subprocesso principal).
- **Para el subprocesso principal y la sincronización:**
  - **maestra:** Especifica que solo el subprocesso principal debe ejecutar una sección del programa.
  - **crítica:** Especifica que el código solo se ejecuta en un subprocesso cada vez.
  - **barrier:** Sincroniza todos los subprocessos de un equipo; todos los subprocessos se pausan en la barrera hasta que todos los subprocessos ejecutan la barrera.
  - **atomic:** Especifica una ubicación de memoria que se actualizará atómicamente.

## 2.7. CUDA

CUDA (Compute Unified Device Architecture [23]), se define como una plataforma de cómputo paralelo de propósito general, y como un modelo de programación. CUDA se distingue, principalmente, por su capacidad para aprovechar la potencia computacional de las GPUs de NVIDIA. Esto, permite la resolución eficiente de problemas complejos, y computacionalmente intensivos. Al facilitar la programación de GPUs, y permitir la ejecución de tareas en paralelo, logra acelerar significativamente aplicaciones en áreas como la inteligencia artificial, la simulación científica, y el procesamiento de imágenes, aprovechando la capacidad masiva de cómputo de las GPUs.

### 2.7.1. Gestión de memoria

CUDA permite el desarrollo de aplicaciones en entornos de computación heterogéneos, diferenciando entre memoria asociada a CPU, y memoria asociada a GPU, (separadas por un bus PCI-Express). De esta manera, para ejecutar una sección de código en CPU, sus datos asociados deberán encontrarse en la memoria de la CPU, y, para ejecutar código en GPU, sus datos asociados deberán encontrarse en la memoria asociada a la GPU.

### 2.7.2. Kernels

CUDA C++ (el empleado en este proyecto) extiende el lenguaje C++ al permitir al programador definir funciones C++ denominadas “kernels”, que, cuando se invocan, se ejecutan N veces en paralelo mediante N hebras CUDA distintas, a diferencia de las funciones C++ tradicionales que se ejecutan una sola vez. Cuando se lanza una función kernel desde la CPU, la ejecución se traslada a la GPU. Una vez en la GPU, esta genera un gran número de hebras y cada una de ellas ejecuta las órdenes especificadas en dicho kernel [66].

### 2.7.3. Jerarquía de hebras

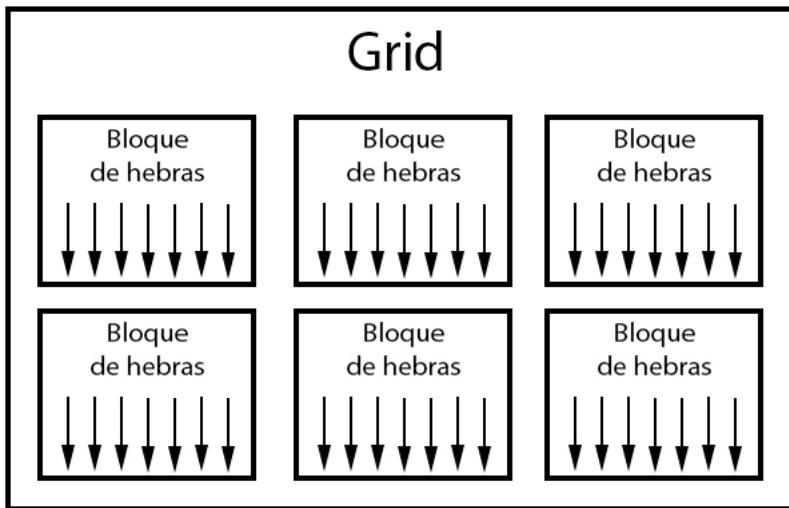


Figura 2.21: Jerarquía de hebras en un grid CUDA

Las hebras CUDA se organizan en una cuadrícula o grid compuesta por varios bloques de hebras (véase la Figura 2.21). De esta manera, cada hebra pertenece a un bloque de hebras, y cada bloque de un mismo kernel pertenece a un mismo grid. Todas las hebras de un mismo grid comparten la misma memoria global. Por tanto, hebras de distintos grids no pueden cooperar. A su vez, las hebras de un mismo bloque se pueden sincronizar y compartir memoria a nivel de bloque, siendo esta más escasa pero presentando una latencia considerablemente menor que la memoria global. CUDA organiza los grids y bloques mediante estructuras que pueden ser 1D, 2D o incluso 3D [9].

## 2.8. cuDNN (Deep Neural Network)

Es una librería de primitivas acelerada por GPU para redes neuronales profundas. Proporciona implementaciones altamente optimizadas para rutinas estándar como la propagación hacia delante y hacia detrás en capas convolucionales, de agrupación máxima, e incluso con funciones de activación como ReLU o sigmoide, entre otras [24].

Su meta es obtener el mejor rendimiento posible en GPUs de NVIDIA para casos importantes de aprendizaje profundo. Dados sus buenos resultados, se emplea en gran cantidad de frameworks de aprendizaje profundo, siendo algunos de ellos Caffe2, Keras, MATLAB, Pytorch, o TensorFlow [30].

Está diseñada para ser utilizada en aplicaciones de aprendizaje profundo que requieran un poder computacional intensivo, permitiendo a desarrolladores e investigadores aprovechar al máximo las prestaciones de las GPUs de NVIDIA. Entre ellas, destaca la compatibilidad con múltiples arquitecturas de GPU, su optimización de la memoria, y su flexibilidad y facilidad de integración. Por todo esto y más, se ha convertido en un componente esencial en el desarrollo de soluciones de inteligencia artificial, facilitando la investigación y la innovación en el campo del aprendizaje profundo.

### 2.8.1. Manejador

cuDNN asume que los datos necesarios se encuentran ya en GPU y accesibles desde device. Se hablará de esto con más detalle en la sección sobre cuda y GPU.

Una aplicación que use cuDNN requiere de la inicialización de un manejador o handle. Dicho manejador será requerido por cuDNN en cada operación que se quiera realizar con dicha librería, permitiendo al usuario un control explícito sobre el funcionamiento de la misma aunque este emplee múltiples hebras o GPUs.

Por ejemplo, en el caso de múltiples GPUs, se pueden asociar diferentes dispositivos con diferentes hebras del host, de forma que cada una tenga un manejador de cuDNN distinto. Así, las llamadas a cuDNN con distinto manejador se ejecutarán en distinta GPU [67].

### 2.8.2. Tensores

Las operaciones en cuDNN reciben tensores como entrada y producen tensores como salida [67].

#### Tensor 3D

Un tensor 3D se suele emplear para representar un volumen 3D como podría ser una imagen RGB. Sus dimensiones se describen mediante 3 letras: B, M y N.

1. **B**: Tamaño del batch
2. **M**: Filas por imagen 2D
3. **N**: Columnas por imagen 2D

### Tensor 4D

Suele representar conjuntos de imágenes 2D. Sus dimensiones son N, C, H, W

1. **N**: Tamaño del batch
2. **C**: Número de imágenes 2D
3. **H**: Filas por imagen 2D
4. **W**: Columnas por imagen 2D

cuDNN permite varios formatos pero usaremos NCHW por compatibilidad con el resto de implementaciones desarrolladas en este proyecto.

### Representación de un tensor 4D

C = 1						
Dimensiones	1	2	3	4	5	6
	7	8	9	10	11	12
	13	14	15	16	17	18
	19	20	21	22	23	24
	25	26	27	28	29	30

C = 2						
N = 1	31	32	33	34	35	36
	37	38	39	40	41	42
	43	44	45	46	47	48
	49	50	51	52	53	54
	55	56	57	58	59	60

Figura 2.22: Ejemplo de tensor 4D con dimensiones: N=1, C=1, H=5, y W=6

En la figura 2.22 se muestra un conjunto de imágenes 3D con las siguientes dimensiones:

1. **N**: Tamaño del batch, 1
2. **C**: Número de imágenes 2D o canales por imagen 3D, 2
3. **H**: Filas por imagen 2D, 5
4. **W**: Columnas por imagen 2D, 6

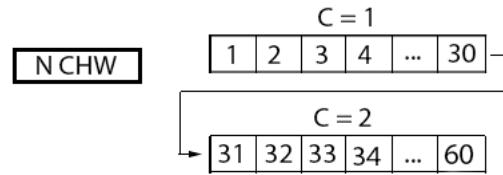


Figura 2.23: Ejemplo de tensor 4D NCHW con dimensiones: N=1, C=1, H=5, y W=6

En la figura 2.23 se muestra como se almacena en memoria el volumen de datos de la figura 2.22 según el formato NCHW. Esto es, por cada imagen 3D  $n \in \{0, \dots, N\}$ , almacenar cada canal  $c \in \{0, \dots, C\}$  según una ordenación por filas.

### 2.8.3. Principales funciones

A continuación se mencionarán las principales funciones de cuDNN que se han empleado en este proyecto, siendo las responsables tanto de la propagación hacia delante como de la retropropagación en capas convolucionales y de agrupación máxima, entre otras.

#### cudnnConvolutionForward

Realiza la propagación hacia delante en una capa convolucional.

```
cudnnStatus_t cudnnConvolutionForward(
    cudnnHandle_t                  handle,
    const void                      *alpha,
    const cudnnTensorDescriptor_t   xDesc,
    const void                      *x,
    const cudnnFilterDescriptor_t   wDesc,
    const void                      *w,
    const cudnnConvolutionDescriptor_t convDesc,
    cudnnConvolutionFwdAlgo_t       algo,
    void                            *workSpace,
    size_t                           workSpaceSizeInBytes,
    const void                      *beta,
```

```
const cudnnTensorDescriptor_t      yDesc,
void                           *y)
```

1. **handle**: Manejador.
2. **alpha, beta**: Punteros a escalares empleados para combinar los resultados con valores anteriores tal que  $\text{valor\_final} = \text{alpha} * \text{result} + \text{beta} * \text{valor\_anterior}$ .
3. **xDesc**: Descriptor asociado al tensor de entrada.
4. **x**: Puntero a los datos de entrada en GPU asociados con el descriptor de tensor xDesc.
5. **wDesc**: Descriptor asociado al tensor de pesos.
6. **w**: Puntero a los pesos en GPU asociados con el descriptor de tensor wDesc.
7. **convDesc**: Descriptor de convolución.
8. **algo**: Especifica qué algoritmo de convolución aplicar.
9. **workSpace**: Puntero a un espacio de trabajo en memoria de GPU.
10. **workSpaceSizeInBytes**: Especifica el tamaño en bytes de workSpace.
11. **yDesc**: Descriptor asociado al tensor de salida.
12. **y**: Puntero a los datos de salida en GPU asociados con el descriptor de tensor yDesc.

[68]

### cudnnPoolingForward

Se encarga de la propagación hacia delante en una capa de agrupación máxima.

```
cudnnStatus_t cudnnPoolingForward(
    cudnnHandle_t                  handle,
    const cudnnPoolingDescriptor_t  poolingDesc,
    const void                      *alpha,
    const cudnnTensorDescriptor_t   xDesc,
    const void                      *x,
    const void                      *beta,
    const cudnnTensorDescriptor_t   yDesc,
    void                           *y)
```

1. **handle**: Manejador.
2. **poolingDesc**: Descriptor de la operación de agrupación.
3. **alpha, beta**: Punteros a escalares empleados para combinar los resultados con valores anteriores tal que  $\text{valor\_final} = \text{alpha} * \text{result} + \text{beta} * \text{valor\_anterior}$ .
4. **xDesc**: Descriptor asociado al tensor de entrada.
5. **x**: Puntero a los datos de entrada en GPU asociados con el descriptor de tensor xDesc.
6. **yDesc**: Descriptor asociado al tensor de salida.
7. **y**: Puntero a los datos de salida en GPU asociados con el descriptor de tensor yDesc.

[69]

### cudnnPoolingBackward

Realiza la retropropagación en una capa de agrupación máxima.

```
cudnnStatus_t cudnnPoolingBackward(
    cudnnHandle_t                  handle,
    const cudnnPoolingDescriptor_t  poolingDesc,
    const void                      *alpha,
    const cudnnTensorDescriptor_t   yDesc,
    const void                      *y,
    const cudnnTensorDescriptor_t   dyDesc,
    const void                      *dy,
    const cudnnTensorDescriptor_t   xDesc,
    const void                      *xData,
    const void                      *beta,
    const cudnnTensorDescriptor_t   dxDesc,
    void                           *dx)
```

1. **handle**: Manejador.
2. **poolingDesc**: Descriptor de la operación de agrupación.
3. **alpha, beta**: Punteros a escalares empleados para combinar los resultados con valores anteriores tal que  $\text{valor\_final} = \text{alpha} * \text{result} + \text{beta} * \text{valor\_anterior}$ .
4. **yDesc**: Descriptor asociado al tensor de salida.

5. **y**: Puntero a los datos de salida en GPU asociados con el descriptor de tensor yDesc.
6. **dyDesc**: Descriptor asociado al tensor que almacena el gradiente de la pérdida respecto a los datos de salida.
7. **dy**: Puntero al gradiente de la pérdida respecto a los datos de salida en GPU asociados con el descriptor de tensor dyDesc.
8. **xDesc**: Descriptor asociado al tensor de entrada.
9. **x**: Puntero a los datos de entrada en GPU asociados con el descriptor de tensor xDesc.
10. **dxDesc**: Descriptor asociado al tensor que almacena el gradiente de la pérdida respecto a los datos de entrada.
11. **dx**: Puntero al gradiente de la pérdida respecto a los datos de entrada en GPU asociados con el descriptor de tensor dxDesc.

[69]

### cudnnConvolutionBackwardFilter

Realiza la retropropagación respecto a los pesos en una capa convolucional.

```
cudnnStatus_t cudnnConvolutionBackwardFilter(
    cudnnHandle_t           handle,
    const void*              *alpha,
    const cudnnTensorDescriptor_t   xDesc,
    const void*              *x,
    const cudnnTensorDescriptor_t   dyDesc,
    const void*              *dy,
    const cudnnConvolutionDescriptor_t convDesc,
    cudnnConvolutionBwdFilterAlgo_t algo,
    void*                   *workSpace,
    size_t                  workSpaceSizeInBytes,
    const void*              *beta,
    const cudnnFilterDescriptor_t dwDesc,
    void*                   *dw)
```

1. **handle**: Manejador.
2. **alpha, beta**: Punteros a escalares empleados para combinar los resultados con valores anteriores tal que  $\text{valor\_final} = \text{alpha} * \text{result} + \text{beta} * \text{valor\_anterior}$ .

3. **xDesc**: Descriptor asociado al tensor de entrada.
4. **x**: Puntero a los datos de entrada en GPU asociados con el descriptor de tensor xDesc.
5. **dyDesc**: Descriptor asociado al tensor que almacena el gradiente de la pérdida respecto a los datos de salida.
6. **dy**: Puntero al gradiente de la pérdida respecto a los datos de salida en GPU asociados con el descriptor de tensor dyDesc.
7. **convDesc**: Descriptor de convolución.
8. **algo**: Especifica qué algoritmo de convolución aplicar.
9. **workSpace**: Puntero a un espacio de trabajo en memoria de GPU.
10. **workSpaceSizeInBytes**: Especifica el tamaño en bytes de workSpace.
11. **dwDesc**: Descriptor del tensor asociado al gradiente de la pérdida respecto a los pesos.
12. **dw**: Puntero al gradiente de los pesos en GPU asociados con el descriptor de tensor dwDesc.

[70]

### cudnnConvolutionBackwardData

Realiza la retropropagación respecto a los datos de entrada en una capa convolucional.

```
cudnnStatus_t cudnnConvolutionBackwardData(
    cudnnHandle_t                  handle,
    const void                     *alpha,
    const cudnnFilterDescriptor_t   wDesc,
    const void                     *w,
    const cudnnTensorDescriptor_t   dyDesc,
    const void                     *dy,
    const cudnnConvolutionDescriptor_t convDesc,
    cudnnConvolutionBwdDataAlgo_t   algo,
    void                           *workSpace,
    size_t                          workSpaceSizeInBytes,
    const void                     *beta,
    const cudnnTensorDescriptor_t   dxDesc,
    void                           *dx)
```

1. **handle**: Manejador.
2. **alpha, beta**: Punteros a escalares empleados para combinar los resultados con valores anteriores tal que  $\text{valor\_final} = \text{alpha} * \text{result} + \text{beta} * \text{valor\_anterior}$ .
3. **wDesc**: Descriptor asociado al tensor de pesos.
4. **w**: Puntero a los pesos en GPU asociados con el descriptor de tensor wDesc.
5. **dyDesc**: Descriptor asociado al tensor que almacena el gradiente de la pérdida respecto a los datos de salida.
6. **dy**: Puntero al gradiente de la pérdida respecto a los datos de salida en GPU asociados con el descriptor de tensor dyDesc.
7. **convDesc**: Descriptor de convolución.
8. **algo**: Especifica qué algoritmo de convolución aplicar.
9. **workSpace**: Puntero a un espacio de trabajo en memoria de GPU.
10. **workSpaceSizeInBytes**: Especifica el tamaño en bytes de workSpace.
11. **dxDesc**: Descriptor asociado al tensor que almacena el gradiente de la pérdida respecto a los datos de entrada.
12. **dx**: Puntero al gradiente de la pérdida respecto a los datos de entrada en GPU asociados con el descriptor de tensor dxDesc.



# Capítulo 3

## Aportaciones

### 3.1. Planificación

Para el desarrollo de este proyecto se ha requerido llevar a cabo una serie de tareas con diferentes dificultades e importancias. A continuación se muestra una planificación general del mismo en la tabla 3.1 junto con las fases que componen el desarrollo, así como una planificación temporal de cada apartado por separado. Cabe destacar que cada apartado se basa en los conocimientos adquiridos en los apartados anteriores a la vez que introduce conceptos nuevos y mejora las prestaciones del modelo, por lo que cada uno supondrá retos nuevos nunca antes vistos y, si un apartado anterior presenta algún fallo desconocido en el momento, se deberá volver a la etapa anterior y arreglarlo y, tras ello, seguir con la etapa posterior. Además, dada la naturaleza de ‘caja oscura’ de las redes neuronales, estas presentan cierta dificultad a la hora de depurar el código. Por tanto, esto supondrá bastante tiempo de depuración en todas y cada una de las implementaciones, tal y como se verá a continuación.

- **Estudio previo:** Consiste en el estudio y comprensión de cuestiones generales dentro del campo de aprendizaje automático y visión por computador, comunes a redes neuronales totalmente conectadas y redes neuronales convolucionales.
  - Investigación, comprensión y selección de funciones de activación a emplear (4 horas)
  - Investigación y comprensión sobre el algoritmo del descenso del gradiente (3 horas)
  - Investigación y comprensión sobre el entrenamiento una red neuronal (2 horas)
  - Investigación y comprensión de sistemas heterogéneos y sistemas homogéneos (3 horas)

- Investigación y comprensión de sistemas heterogéneos aplicados a CNNs (4 horas)
- **Investigación y desarrollo de redes neuronales totalmente conectadas:** En este periodo me centré en la investigación y comprensión total sobre las redes neuronales totalmente conectadas a bajo nivel. De esta forma, sabía que podría generar cualquier tipo de red totalmente conectada de manera dinámica, sin necesidad de realizar ningún tipo de cálculo posterior, independientemente del lenguaje de programación empleado, así como del uso o no de librerías que faciliten el proceso.
  - Investigación y desarrollo teórico de la retropropagación en una red totalmente conectada con 1 capa oculta (40 horas)
  - Investigación y desarrollo teórico de la retropropagación en una red totalmente conectada con 2 capas ocultas (9 horas)
  - Desarrollo teórico de la retropropagación en una red totalmente conectada con N capas ocultas (5 horas)
  - Investigación y desarrollo teórico sobre la retropropagación en una capa SoftMax (7 horas)
  - Implementación en C++ sobre la retropropagación en una capa softmax (8 horas)
  - Investigación y desarrollo del algoritmo del descenso del gradiente en una red totalmente conectada en C++ (2 horas)
  - Investigación sobre distintas inicializaciones de pesos e implementación de la inicialización de pesos HE (2 horas)
  - Realización de pruebas prácticas de funcionamiento con distintas bases de datos y arquitecturas de modelos (10 horas)
  - Implementación en C++ sobre cuestiones generales de una red neuronal totalmente conectada como entrenamiento, lectura de imágenes, creación, gestión y actualización de pesos y sesgos, evaluación del modelo sobre un conjunto de datos, creación de estructura general, etc (60 horas).
- **Investigación y desarrollo de redes neuronales convolucionales:** Una vez familiarizado con redes neuronales totalmente conectadas, se trató de comprender de igual forma las redes neuronales convolucionales, pues se encuentran ampliamente relacionadas.
  - Investigación y desarrollo teórico de la propagación hacia delante en una capa convolucional de una red neuronal convolucional (5 horas)
  - Investigación y desarrollo teórico de la retropropagación en una capa convolucional con y sin relleno en una red neuronal convolucional (21 horas)

- Investigación y desarrollo teórico sobre el relleno en una red neuronal convolucional (12 horas)
  - Desarrollo de conclusiones teóricas respecto a retropropagación en capa convolucional con y sin relleno (4 horas)
  - Investigación y desarrollo teórico sobre una capa de agrupación máxima en una red neuronal convolucional (8 horas)
  - Investigación y desarrollo teórico sobre una capa de aplanado en una red neuronal convolucional (1 hora)
  - Implementación en C++ sobre la propagación hacia delante en una capa convolucional de una red neuronal convolucional (8 horas).
  - Implementación en C++ sobre la retropropagación en una capa convolucional de una red neuronal convolucional (12 horas).
  - Implementación en C++ sobre la propagación hacia delante en una capa de agrupación máxima de una red neuronal convolucional (4 horas).
  - Implementación en C++ sobre la retropropagación en una capa de agrupación máxima de una red neuronal convolucional (6 horas).
  - Implementación en C++ sobre una capa de aplanado de una red neuronal convolucional (1 hora).
  - Implementación en C++ sobre cuestiones generales de una CNN como entrenamiento, lectura de imágenes, evaluación del modelo sobre un conjunto de datos, creación de estructura general, etc (70 horas).
- **Investigación y desarrollo de sistemas homogéneos con OpenMP:**  
Una vez comprendido el funcionamiento tanto de las redes neuronales totalmente conectadas como de las redes neuronales convolucionales, me centré en reducir los tiempos de cómputo requeridos en ellas mediante un paralelismo orientado a datos con OpenMP.
- Investigación sobre tipos de paralelismo con OpenMP orientados tanto a redes neuronales totalmente conectadas como a redes neuronales convolucionales (4 horas)
  - Investigación y desarrollo teórico sobre paralelismo a nivel de datos con OpenMP en el algoritmo del descenso del gradiente (3 horas).
  - Implementación en C++ y OpenMP sobre el algoritmo del descenso del gradiente en la red neuronal totalmente conectada (10 horas).

- Implementación en C++ y OpenMP sobre el algoritmo del descenso del gradiente en la red neuronal convolucional (30 horas).
  - Implementación en C++ y OpenMP sobre aspectos generales de una red neuronal totalmente conectada (20 horas).
  - Implementación en C++ y OpenMP sobre cuestiones generales de una red neuronal convolucional (30 horas).
  - Realización de pruebas finales para verificar un correcto entrenamiento y aprendizaje (6 horas)
- **Investigación y desarrollo de sistemas heterogéneos con CUDA y cuDNN:** Teniendo ya los conocimiento teóricos y prácticos sobre sistemas homogéneos paralelos respecto a las redes neuronales totalmente conectadas y las redes neuronales convolucionales, se da un salto y nos adentramos en los sistemas heterogéneos aplicados a las redes neuronales totalmente conectadas y a las redes neuronales convolucionales.
    - Investigación y desarrollo teórico sobre la propagación hacia delante y retropropagación en una capa convolucional como GEMM (8 horas)
    - Investigación y desarrollo teórico sobre la propagación hacia delante y retropropagación en una capa de agrupación máxima con CUDA (5 horas)
    - Investigación y comprensión de la librería cuBLAS (5 horas)
    - Implementación en C++ sobre la propagación hacia delante en una capa convolucional como GEMM con cuBLAS (6 horas)
    - Implementación en C++ sobre la retropropagación en una capa convolucional como GEMM con cuBLAS (26 horas)
    - Investigación y desarrollo teórico y práctico sobre el algoritmo de multiplicación matricial con CUDA empleado (30 horas)
    - Implementación en C++ y CUDA sobre la capa de agrupación máxima con CUDA (14 horas)
    - Investigación y desarrollo teórico y práctico sobre la capa totalmente conectada como GEMM (31 horas)
    - Investigación y desarrollo teórico sobre el consumo de memoria con GEMM (5 horas)
    - Implementación de aspectos generales de una red neuronal convolucional con CUDA (71 horas).
    - Sustitución de la clase Vector de la STL en todo el proyecto por punteros necesarios para emplear CUDA (30 horas).

- Investigación teórica y adaptación práctica de la implementación con CUDA a cuDNN (51 horas).

Apartado	Tiempo (Horas)
Estudio previo	16
Investigación y desarrollo de redes neuronales totalmente conectadas	143
Investigación y desarrollo de redes neuronales convolucionales	152
Investigación y desarrollo de sistemas homogéneos con OpenMP	103
Investigación y desarrollo de sistemas heterogéneos con CUDA y cuDNN	282
Tiempo total:	696

Cuadro 3.1: Planificación del proyecto

### 3.2. Retropropagación en redes neuronales totalmente conectadas

En esta sección se analizará en profundidad el cálculo del gradiente de la pérdida respecto a cada parámetro entrenable de una red totalmente conectada, así como también respecto a la entrada y salida de cada capa. Primero se mostrarán los cálculos para unos ejemplos concretos, y una vez conocidas las bases sobre ellos, se mostrará cómo aplicarlos para cualquier tipo de red totalmente conectada.

#### 3.2.1. Retropropagación en capa SoftMax, [1] [2]

Tal y como se comentó en secciones anteriores, se empleará SoftMax en la última capa totalmente conectada. Así, se definen los valores de entrada a la misma como  $Z$ , y los de salida como  $O$ , tal y como se muestra en la Figura 3.1.

## 3.2. Retropropagación en redes neuronales totalmente conectadas

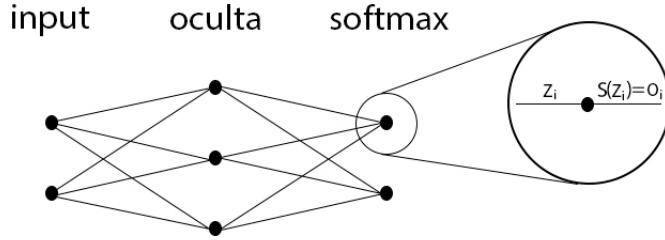


Figura 3.1: Estructura de una red totalmente conectada con softmax en la última capa

$$E = - \sum_{i=1}^H [y_i * \log(O_i)] \quad (3.1)$$

Según esta notación, la función de error 2.3 se convierte en la fórmula 3.1.

### Cálculo del gradiente de la función de error

Para comenzar la retropropagación, empezaremos por calcular el gradiente de la función de pérdida respecto a cada parámetro de entrada de la capa en la cual se aplicó SoftMax, esto se muestra en la fórmula 3.2.

$$\frac{\partial E}{\partial Z_k} = \frac{\partial(-\sum_{i=1}^H [y_i * \log(O_i)])}{\partial Z_k} = - \sum_{i=1}^H \left[ \frac{\partial(y_i * \log(O_i))}{\partial Z_k} \right] \quad (3.2)$$

Como  $y_i$  (etiqueta real) es independiente respecto a  $Z_k$  (neurona artificial de entrada) en la fórmula 3.2, esta se trata como una constante.

$$\frac{\partial E}{\partial Z_k} = - \sum_{i=1}^H [y_i * \frac{\partial(\log(O_i))}{\partial Z_k}] \quad (3.3)$$

Por simplificar los cálculos y eliminar el logaritmo de la ecuación, se aplica la regla de la cadena y como resultado se obtienen las fórmulas 3.4 y 3.5.

$$\frac{\partial E}{\partial Z_k} = - \sum_{i=1}^H [y_i * \frac{\partial(\log(O_i))}{\partial O_i} * \frac{\partial O_i}{\partial Z_k}] \quad (3.4)$$

$$\frac{\partial E}{\partial Z_k} = - \sum_{i=1}^H [\frac{y_i}{O_i} * \frac{\partial O_i}{\partial Z_k}] \quad (3.5)$$

**Derivada de softmax respecto de su entrada,  $\frac{\partial O_i}{\partial Z_k}$** 

Una vez obtenida la fórmula 3.5 nos disponemos a calcular la derivada de  $O_i$  respecto  $Z_i$ .

Sin embargo, hay que contemplar 2 casos posibles, siendo estos  $\frac{\partial S(Z_i)}{\partial Z_i}$  y  $\frac{\partial S(Z_i)}{\partial Z_j}$ , donde  $i \neq j$ .

**Caso  $\frac{\partial S(Z_i)}{\partial Z_i}$**

$$\frac{\partial f(x)}{\partial g(x)} = \frac{f'(x) * g(x) - g'(x) * f(x)}{g(x)^2} \quad (3.6)$$

$$S(z_i) = \frac{e^{Z_i}}{e^{Z_1} + \dots + e^{Z_H}} \quad (3.7)$$

$$\frac{\partial S(Z_1)}{\partial Z_1} = \frac{[\frac{\partial e^{Z_1}}{\partial Z_1} * (e^{Z_1} + \dots + e^{Z_H})] - [\frac{\partial(e^{Z_1} + \dots + e^{Z_H})}{\partial Z_1} * e^{Z_1}]}{(e^{Z_1} + \dots + e^{Z_H})^2} \quad (3.8)$$

Se aplica  $\frac{\partial e^{Z_1}}{\partial Z_1} = e^{Z_1}$

$$\frac{\partial S(Z_1)}{\partial Z_1} = \frac{[e^{Z_1} * \sum_{i=1}^H e^{Z_i}] - [e^{Z_1} * e^{Z_1}]}{(\sum_{i=1}^H e^{Z_i})^2} \quad (3.9)$$

(3.10)

Se saca factor común  $e^{Z_1}$

$$\frac{\partial S(Z_1)}{\partial Z_1} = \frac{e^{Z_1}([\sum_{i=1}^H e^{Z_i}] - e^{Z_1})}{(\sum_{i=1}^H e^{Z_i})^2} \quad (3.11)$$

$$\frac{\partial S(Z_1)}{\partial Z_1} = \frac{e^{Z_1}}{\sum_{i=1}^H e^{Z_i}} * \frac{[\sum_{i=1}^H e^{Z_i}] - e^{Z_1}}{\sum_{i=1}^H e^{Z_i}} \quad (3.12)$$

Se recuerda que  $\frac{\sum_{i=1}^H e^{Z_i}}{\sum_{i=1}^H e^{Z_i}} = 1$  y que  $S(Z_1) = \frac{e^{Z_1}}{\sum_{i=1}^H e^{Z_i}}$

$$\frac{\partial S(Z_1)}{\partial Z_1} = S(Z_1) * (1 - S(Z_1)) \quad (3.13)$$

## 3.2. Retropropagación en redes neuronales rotalmente conectadas

Caso  $\frac{\partial S(Z_i)}{\partial Z_j}$ , con  $i \neq j$

$$\frac{\partial S(Z_2)}{\partial Z_1} = \frac{[\frac{\partial e^{Z_2}}{\partial Z_1} * (e^{Z_1} + \dots + e^{Z_H})] - [\frac{\partial(e^{Z_1} + \dots + e^{Z_H})}{\partial Z_1} * e^{Z_2}]}{(e^{Z_1} + \dots + e^{Z_H})^2} \quad (3.14)$$

$$\frac{\partial S(Z_2)}{\partial Z_1} = \frac{[0 * [\sum_{i=1}^H e^{Z_i}]] - [e^{Z_1} * e^{Z_2}]}{(\sum_{i=1}^H e^{Z_i})^2} \quad (3.15)$$

$$\frac{\partial S(Z_2)}{\partial Z_1} = \frac{-e^{Z_1} * e^{Z_2}}{(\sum_{i=1}^H e^{Z_i})^2} \quad (3.16)$$

$$\frac{\partial S(Z_2)}{\partial Z_1} = \frac{-e^{Z_1}}{\sum_{i=1}^H e^{Z_i}} * \frac{e^{Z_2}}{\sum_{i=1}^H e^{Z_i}} \quad (3.17)$$

$$\frac{\partial S(Z_2)}{\partial Z_1} = -S(Z_1) * S(Z_2) \quad (3.18)$$

### Combinación de casos

De esta forma, tendremos que dividir el proceso en 2 partes, cuando  $i$  sea igual a  $j$ , y cuando  $i \neq j$ .

Como todos los casos menos uno pertenecen al caso  $i \neq k$ , en la fórmula 3.19 se aprecia como la “parte izquierda” hace referencia al caso  $i=k$ , mientras que la “parte derecha” a  $i=k$ .

Retomamos la fórmula 3.5, aplicando 3.18 en la parte izquierda y 3.13 en la derecha.

$$\frac{\partial E}{\partial Z_k} = -[\sum_{i=k}^H \left[ \frac{y_i}{O_i} * -O_i * O_k \right] + \frac{y_k}{O_k} * O_k * (1 - O_k)] \quad (3.19)$$

Una vez obtenida la fórmula 3.19, se simplifica  $O_i$  en la parte izquierda y  $O_k$  en la derecha.

$$\frac{\partial E}{\partial Z_k} = -[\sum_{i=k}^H [-y_i * O_k] + [y_k * (1 - O_k)]] \quad (3.20)$$

Se extrae  $O_k$  de la suma en la fórmula 3.20, pues es independiente respecto al índice  $i$  y se obtiene la fórmula 3.21.

$$\frac{\partial E}{\partial Z_k} = -[-O_k \sum_{i=k}^H [-y_i] + [y_k * (1 - O_k)]] \quad (3.21)$$

### Simplificación One-Hot

Al emplear la codificación one-hot en  $Y$ , se sabe que la suma de sus elementos es igual a 1, pues para un ejemplo de entrada  $x_i \in X$ , su etiqueta asociada  $y_i \in Y$  presenta todos sus valores iguales a 0 menos uno de ellos con el valor de 1.

Con estos datos, se calculan las fórmulas 3.22 y 3.23.

$$\sum_{i=1}^H y_i = 1 \quad (3.22)$$

$$\sum_{i=k}^H y_i = \sum_{i=1}^H [y_i] - y_k = 1 - y_k \quad (3.23)$$

Una vez obtenidas dichas fórmulas, se emplea 3.23 para simplificar la suma anterior obtenida en 3.21, y como resultado de ello se elaboran las siguientes fórmulas 3.24 y 3.25.

$$\frac{\partial E}{\partial Z_k} = [O_k * (1 - y_k)] - [y_k * (1 - O_k)] \quad (3.24)$$

$$\frac{\partial E}{\partial Z_k} = O_k - O_k * y_k - y_k + O_k * y_k \quad (3.25)$$

Por último, en la fórmula 3.25 se simplifica  $O_k * y_k$  y se obtiene la fórmula final 3.26.

$$\frac{\partial E}{\partial Z_k} = O_k - y_k = \text{gradiente}_Z_k \quad (3.26)$$

#### 3.2.2. Retropropagación con 1 capa oculta [3] [4]

En esta sección se tratará de calcular el gradiente de la pérdida respecto a cada parámetro de la red totalmente conectada mostrada en la Figura 3.2. Para no repetir cálculos, en esta y secciones posteriores no se volverá a calcular la retropropagación a través de la capa SoftMax, pues los cálculos son siempre los mismos por ser la última capa de la arquitectura.

## 3.2. Retropropagación en redes neuronales totalmente conectadas

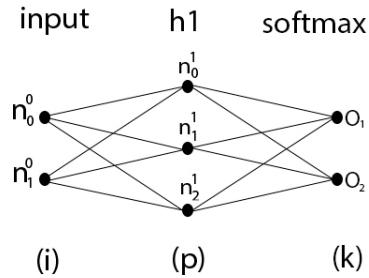


Figura 3.2: Red Neuronal totalmente conectada con 1 capa oculta

La Figura 3.2 se compone de 'puntos' interconectados mediante líneas, representando neuronas y pesos que las conectan respectivamente. Cada punto corresponde a una neurona, y cada línea a un peso.

La Figura 3.2 presenta 3 capas (input,  $h_1$ , softmax) que corresponden a capa de entrada, capa oculta  $h_1$ , y capa de salida respectivamente. El superíndice indica la capa a la que pertenece una neurona o peso, mientras que el subíndice indica el número del mismo en su respectiva capa. En el caso de los pesos, se requieren 2 subíndices para identificar a cada uno (pues un peso une 2 neuronas).

Así, la capa de entrada se compone de 2 neuronas ( $n_0^0$  y  $n_1^0$ ), la capa oculta  $h_1$  tiene 3 neuronas ( $n_0^1$ ,  $n_1^1$ , y  $n_2^1$ ), y el peso  $W_{jk}^i$  referencia al peso que une las neuronas  $n_j^i$  y  $n_k^{i+1}$ .

De forma adicional, se recuerda que  $Z_i$  representa la entrada  $i$  de la capa SoftMax, y  $O_i$  su salida.

### Capa SoftMax

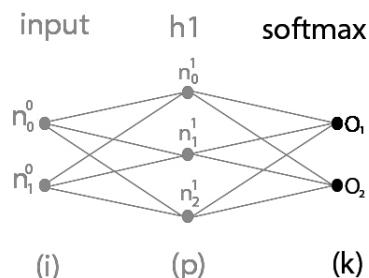


Figura 3.3: Retropropagación en la capa softmax

Sea la neurona  $n_j^i$ , se define como  $a_j^i$  el valor de dicha neurona antes de aplicar sobre ella su función de activación asociada, y  $z_j^i$  el obtenido tras aplicarla.

Tal y como se calculó previamente, el gradiente de la función de pérdida respecto a cada  $Z_i$  viene dado por la fórmula 3.26.

### Pesos capas h1-SoftMax

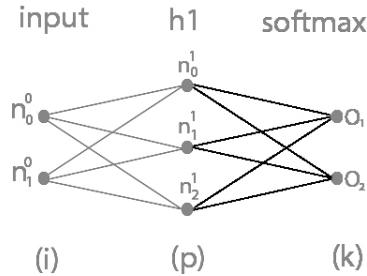


Figura 3.4: Retropropagación respecto a pesos entre la capa oculta y la capa SoftMax

Una vez calculado el gradiente hasta la entrada de la capa softmax, se puede calcular el gradiente respecto a cada peso  $W_{pk}^1$  que se encuentra conectado a esta desde la capa anterior. Es decir, para cada  $h_p^1 \in h_1$ , se calcula  $\frac{dE(x)}{dW_{pk}^1}$ . Usando la regla de la cadena, equivale a realizar lo ilustrado en las fórmulas 3.27 y 3.28.

$$\frac{\partial Z_k}{\partial W_{pk}^1} = \frac{\partial(z_p^1 * W_{pk}^1 + b_k^2)}{\partial W_{pk}^1} = z_p^1 \quad (3.27)$$

$$\frac{\partial E(x)}{\partial W_{pk}^1} = \text{gradiente\_}Z_k * \frac{\partial Z_k}{\partial W_{pk}^1} = \text{gradiente\_}Z_k * z_p^1 \quad (3.28)$$

### Sesgos capa softmax

De la misma forma, se calcula el gradiente de la pérdida respecto a cada sesgo de las neuronas de la capa softmax tal y como se muestra en las fórmulas 3.29, 3.30 y 3.31.

$$\frac{\partial E}{\partial b_k^2} = \frac{\partial E}{\partial Z_k} * \frac{\partial Z_k}{\partial b_k^2} \quad (3.29)$$

$$\frac{\partial Z_k}{\partial b_k^2} = \frac{\partial([\sum_{c=1}^P z_c^1 * W_{pk}^1] + b_k^2)}{\partial b_k^2} = 1 \quad (3.30)$$

$$\frac{\partial E}{\partial b_k^2} = \text{gradiente\_}Z_k \quad (3.31)$$

## 3Ω. Retropropagación en redes neuronales rotalmente conectadas

### Capa oculta h1

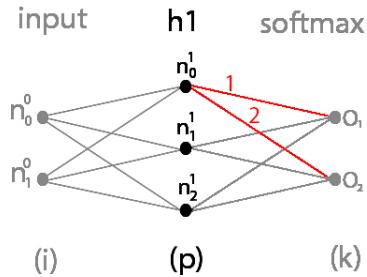


Figura 3.5: Imagen de los 'caminos' desde la capa softmax hasta la neurona  $n_p^1$

En la figura 3.5 se muestra como hay más de un 'camino' desde la capa softmax hasta  $n_p^1$ . Por tanto, para obtener el gradiente de la pérdida respecto a  $n_p^1$ , habría que calcular la suma de todos los 'caminos' hacia este, tal y como se muestra en las fórmulas 3.32 y 3.33.

$$\frac{\partial E_{total}}{\partial a_p^1} = \sum_{k=1}^K \frac{\partial E_k}{\partial a_p^1} = \sum_{k=1}^K \text{gradiente}_Z k * \frac{\partial Z_k}{\partial z_p^1} * \frac{\partial z_p^1}{\partial a_p^1} \quad (3.32)$$

$$\frac{\partial Z_k}{\partial z_p^1} = \frac{\partial([\sum_{c=1}^P z_c^1 * W_{ck}^1] + b_k^2)}{\partial z_p^1} = W_{pk}^1 \quad (3.33)$$

Para calcular dichos gradientes, se requiere calcular  $\frac{\partial z_p^1}{\partial a_p^1}$ . Como se mencionó anteriormente, “a” se refiere al valor de una neurona antes de aplicar la función de activación asociada, y “z” consiste en dicho valor tras aplicar la función de activación. Por tanto, para calcular  $\frac{\partial z_p^1}{\partial a_p^1}$  se requiere saber dicha función de activación.

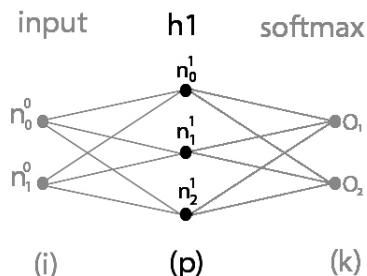


Figura 3.6: Retropropagación respecto a neuronas de la capa oculta h1

En este ejemplo, en la capa oculta h1 se emplea la función de activación sigmoide, y su derivada viene dada por las fórmulas 3.34 y 3.35.

$$\text{sigmoide}(x) = \frac{1}{1 + e^{-x}} \quad (3.34)$$

$$\text{sigmoide}'(x) = \frac{\text{sigmoide}(x)}{1 - \text{sigmoide}(x)} \quad (3.35)$$

De esta forma, ahora sí se puede calcular  $\frac{\partial z_p^1}{\partial a_p^1}$ , y se muestra en la fórmula 3.36.

$$\frac{\partial z_p^1}{\partial a_p^1} = \frac{\partial \text{sigmoide}(a_p^1)}{\partial a_p^1} = \text{sigmoide}(a_p^1) * (1 - \text{sigmoide}(a_p^1)) \quad (3.36)$$

Así, se retoma la fórmula 3.32 mediante la aplicación de 3.33 y 3.36, y se obtiene 3.37.

$$\frac{\partial E_{total}}{\partial a_p^1} = \sum_{k=1}^K \text{gradiente\_Z}_k * W_{pk}^1 * \text{sigmoide}(a_p^1) * (1 - \text{sigmoide}(a_p^1)) \quad (3.37)$$

$$\frac{\partial E_{total}}{\partial a_p^1} = \text{gradiente\_h1}_p \quad (3.38)$$

### Pesos capas entrada-h1

Una vez realizada la retropropagación hasta las neuronas de entrada de la capa oculta h1, se puede seguir con el proceso hacia la capa anterior (capa de entrada).

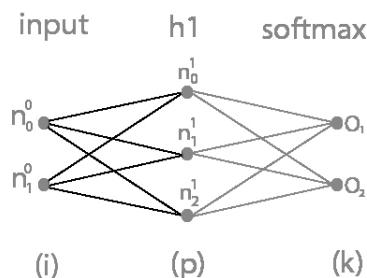


Figura 3.7: Retropropagación respecto a los pesos entre la capa de entrada y la capa oculta h1

## 32. Retropropagación en redes neuronales rotalmente conectadas

$$\frac{\partial a_p^1}{\partial W_{ip}^0} = \frac{\partial [\sum_{c=1}^I z_c^0 * W_{cp}^0] + b_p^1)}{\partial W_{ip}^0} = z_i^0 \quad (3.39)$$

$$\frac{\partial E}{\partial W_{ip}^0} = \frac{\partial E_{total}}{\partial a_p^1} * \frac{\partial a_p^1}{\partial W_{ip}^0} \quad (3.40)$$

$$\frac{\partial E(x)}{\partial W_{ip}^0} = \text{gradiente\_h1}_p * \frac{\partial a_p^1}{\partial W_{ip}^0} = \text{gradiente\_h1}_p * z_i^0 \quad (3.41)$$

De forma similar a los pesos entre la capa oculta h1 y la capa softmax, se calcula el gradiente de la pérdida respecto a los pesos entre la capa de entrada y la capa oculta h1. El proceso se muestra mediante las fórmulas 3.39, 3.40, y 3.41.

### Sesgos capa h1

$$\frac{\partial E}{\partial b_p^1} = \frac{\partial E_{total}}{\partial a_p^1} * \frac{\partial a_p^1}{\partial b_p^1} \quad (3.42)$$

$$\frac{\partial a_p^1}{\partial b_p^1} = \frac{\partial ([\sum_{c=1}^I z_c^0 * W_{ip}^0] + b_p^1)}{\partial b_p^1} = 1 \quad (3.43)$$

$$\frac{\partial E}{\partial b_p^1} = \text{gradiente\_h1}_p \quad (3.44)$$

Del mismo modo, las figuras 3.42, 3.43, y 3.44 muestran el cálculo del gradiente respecto a los sesgos de la capa oculta h1.

### Capa de entrada

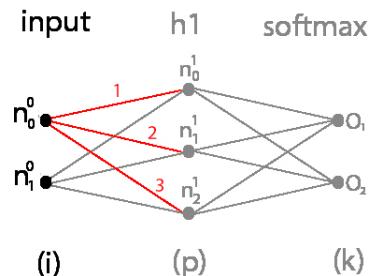


Figura 3.8: Imagen de los 'caminos' desde la capa oculta h1 hasta  $n_0^0$

Por último, en esta sección se calcula el gradiente respecto a las neuronas de entrada de la capa de entrada. Normalmente no sería necesario calcular

estos gradientes, pero como el objetivo es crear una CNN, esta red totalmente conectada estará enlazada a capas convolucionales y de agrupación máxima, por lo que es necesario calcular dichos gradientes para poder seguir calculando la retropropagación en dichas capas anteriores a esta.

$$\frac{\partial E_{total}}{\partial a_i^0} = \sum_{p=1}^P \frac{\partial E_{total}}{\partial a_p^1} * \frac{\partial a_p^1}{\partial z_i^0} * \frac{\partial z_i^0}{\partial a_i^0} \quad (3.45)$$

$$\frac{\partial a_p^1}{\partial z_i^0} = \frac{\partial([\sum_{c=1}^I z_c^0 * W_{ip}^0] + b_p^1)}{\partial z_i^0} = W_{ip}^0 \quad (3.46)$$

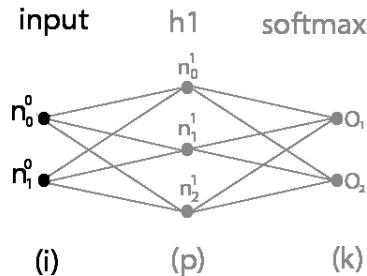


Figura 3.9: Retropropagación en la capa input

Como la capa input no presenta ninguna función de activación asociada,  $z_i^0$  es igual a  $a_i^0$ .

$$\frac{\partial z_i^0}{\partial a_i^0} = 1 \quad (3.47)$$

$$\frac{\partial E_{total}}{\partial a_i^0} = \sum_{p=1}^P \text{gradiente\_h1}_p \quad (3.48)$$

### 3.2.3. Retropropagación con 2 capas ocultas

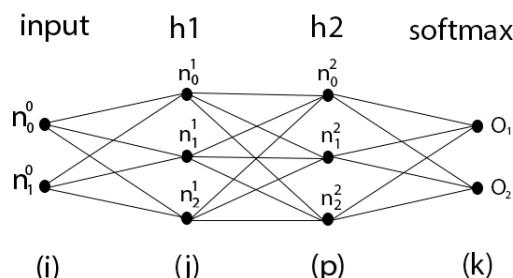


Figura 3.10: Red Neuronal totalmente conectada con 2 capas ocultas

## 3.2. Retropropagación en redes neuronales totalmente conectadas

A diferencia del apartado anterior, en este caso se emplea una red totalmente conectada con 2 capas ocultas ( $h_1$  y  $h_2$ ), tal y como se muestra en la Figura 3.10.

### Capa SoftMax

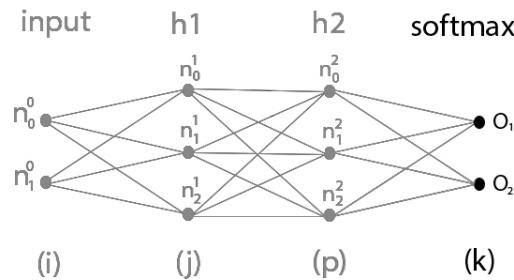


Figura 3.11: Retropropagación en la capa softmax

De igual forma que en los casos anteriores, el gradiente de la función de pérdida respecto a cada  $Z_i$  viene dado por la fórmula 3.26. Por tanto, no se repetirán los cálculos.

### Pesos capas h2-SoftMax

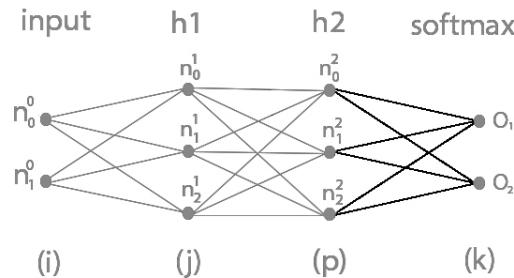


Figura 3.12: Retropropagación respecto a los pesos entre la capa oculta  $h_2$  y la capa SoftMax

Se realiza el cálculo del gradiente de la función de pérdida respecto a cada peso  $W_{pk}^2$  que une las neuronas de la capa oculta  $h_2$  con las de la capa softmax.

$$\frac{\partial Z_k}{\partial W_{pk}^2} = \frac{\partial(z_p^2 * W_{pk}^2 + b_k^3)}{\partial W_{pk}^2} = z_p^2 \quad (3.49)$$

$$\frac{\partial E(x)}{\partial W_{pk}^2} = \text{gradiente\_}Z_k * \frac{\partial Z_k}{\partial W_{pk}^2} = \text{gradiente\_}Z_k * z_p^2 \quad (3.50)$$

Como es de esperar, las fórmulas 3.49 y 3.50 son casi idénticas a 3.27 y 3.28 respectivamente, salvo por el superíndice empleado ( $1 \neq 2$ ). Esto tiene sentido pues esta parte también es común al apartado anterior.

### Sesgos capa softmax

$$\frac{\partial E}{\partial b_k^3} = \frac{\partial E}{\partial Z_k} * \frac{\partial Z_k}{\partial b_k^3} \quad (3.51)$$

$$\frac{\partial Z_k}{\partial b_k^3} = \frac{\partial([\sum_{c=1}^P z_c^2 * W_{pk}^2] + b_k^3)}{\partial b_k^3} = 1 \quad (3.52)$$

$$\frac{\partial E}{\partial b_k^3} = \text{gradiente\_}Z_k \quad (3.53)$$

De igual forma, el cálculo del gradiente de la pérdida respecto a los sesgos de la capa softmax también permanece inalterado, por lo que la única diferencia entre las fórmulas {3.51, 3.52, 3.53} y {3.29, 3.30, 3.31} son los superíndices empleados.

### Capa oculta h2

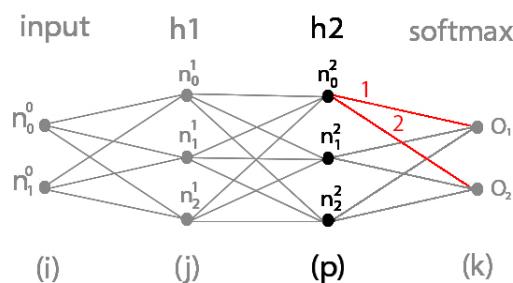


Figura 3.13: Imagen de los 'caminos' desde la capa softmax hasta  $n_0^2$

Tal y como se comentó anteriormente, hay más de un 'camino' desde la capa softmax hasta  $n_p^2$ . Por tanto, para obtener el gradiente de la pérdida respecto a cada  $n_p^2$ , habría que calcular la suma de todos los ellos, tal y como se muestra en las fórmulas 3.54 y 3.55.

## 3.2. Retropropagación en redes neuronales totalmente conectadas

$$\frac{\partial E_{total}}{\partial a_p^2} = \sum_{k=1}^K \frac{\partial E_k}{\partial a_p^2} = \sum_{k=1}^K \text{gradiente\_} Z_k * \frac{\partial Z_k}{\partial z_p^2} * \frac{\partial z_p^2}{\partial a_p^2} \quad (3.54)$$

$$\frac{\partial Z_k}{\partial z_p^2} = \frac{\partial([\sum_{c=1}^P z_c^2 * W_{ck}^2] + b_k^3)}{\partial z_p^2} = W_{pk}^2 \quad (3.55)$$

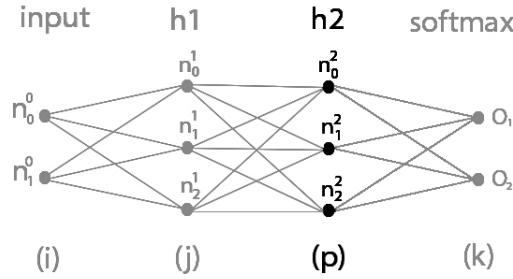


Figura 3.14: Retropropagación en la capa oculta h2

Por coincidir con el ejemplo anterior, en la última capa oculta (h2 en este caso) se vuelve a emplear sigmoide como función de activación. Así, se vuelve a mostrar la derivada de dicha función en las fórmulas 3.56 y 3.57 por facilitar la comprensión del lector.

$$\text{sigmoide}(x) = \frac{1}{1 + e^{-x}} \quad (3.56)$$

$$\text{sigmoide}'(x) = \frac{\text{sigmoide}(x)}{1 - \text{sigmoide}(x)} \quad (3.57)$$

Empleando la derivada de sigmoide se consigue la fórmula 3.58.

$$\frac{\partial z_p^2}{\partial a_p^2} = \frac{\partial \text{sigmoide}(a_p^2)}{\partial a_p^2} = \text{sigmoide}(a_p^2) * (1 - \text{sigmoide}(a_p^2)) \quad (3.58)$$

Tras ello, se retoma la fórmula 3.54 mediante la aplicación de 3.55 y 3.58 para obtener 3.59.

$$\frac{\partial E_{total}}{\partial a_p^2} = \sum_{k=1}^K \text{gradiente\_} Z_k * W_{pk}^2 * \text{sigmoide}(a_p^2) * (1 - \text{sigmoide}(a_p^2)) \quad (3.59)$$

$$\frac{\partial E_{total}}{\partial a_p^2} = \text{gradiente\_} h2_p \quad (3.60)$$

Una vez más, la fórmula obtenida (3.59) coindice con la calculada previamente (3.37) a excepción de los superíndices empleados.

### Pesos capas h1-h2

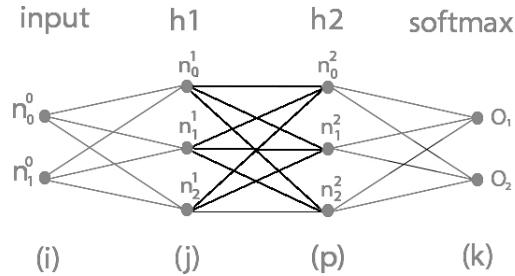


Figura 3.15: Retropropagación respecto a los pesos entre las capas ocultas h1 y h2

$$\frac{\partial a_p^2}{\partial W_{jp}^1} = \frac{\partial [\sum_{c=1}^J z_c^1 * W_{cp}^1] + b_p^2}{\partial W_{jp}^1} = z_j^1 \quad (3.61)$$

$$\frac{\partial E}{\partial W_{jp}^1} = \frac{\partial E_{total}}{\partial a_p^2} * \frac{\partial a_p^2}{\partial W_{jp}^1} \quad (3.62)$$

$$\frac{\partial E(x)}{\partial W_{jp}^1} = \text{gradiente\_h2}_p * \frac{\partial a_p^2}{\partial W_{jp}^1} = \text{gradiente\_h2}_p * z_j^1 \quad (3.63)$$

Como esta parte también es común al caso anterior, la fórmula 3.63 vuelve a coincidir con 3.41

### Sesgos capa h2

$$\frac{\partial E}{\partial b_p^2} = \frac{\partial E_{total}}{\partial a_p^2} * \frac{\partial a_p^2}{\partial b_p^2} \quad (3.64)$$

$$\frac{\partial a_p^2}{\partial b_p^2} = \frac{\partial ([\sum_{c=1}^J z_c^1 * W_{jp}^1] + b_p^2)}{\partial b_p^2} = 1 \quad (3.65)$$

$$\frac{\partial E}{\partial b_p^2} = \text{gradiente\_h2}_p \quad (3.66)$$

Una vez más, la fórmula 3.66 coincide con 3.44. Es importante notar aquellas partes comunes que comparten ambos casos de cara a una posterior generalización del modelo y poder automatizar dichos cálculos.

## 3.2. Retropropagación en redes neuronales rotalmente conectadas

### Capa oculta h1

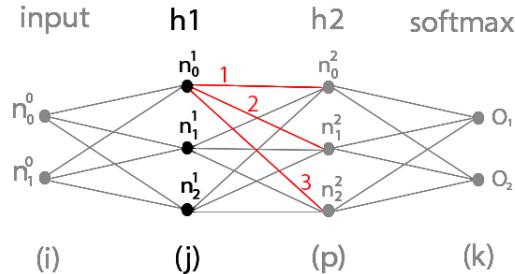


Figura 3.16: 'Caminos' desde la capa softmax hasta  $n_j^1$

De igual forma que se realizó en la capa h2, se calcula la suma de todos los 'caminos' hacia cada neurona  $n_j^1$ .

$$\frac{\partial E_{total}}{\partial a_j^1} = \sum_{k=1}^K \frac{\partial E_k}{\partial a_j^1} = \sum_{p=1}^P \text{gradiente\_h2}_p * \frac{\partial a_p^2}{\partial z_j^1} * \frac{\partial z_j^1}{\partial a_j^1} \quad (3.67)$$

$$\frac{\partial a_p^2}{\partial z_j^1} = \frac{\partial (\sum_{c=1}^J z_c^1 * W_{cp}^1 + b_p^2)}{\partial z_j^1} = W_{jp}^1 \quad (3.68)$$

Como es de esperar, se calcula el gradiente respecto a cada neurona de la capa h1 teniendo en cuenta cada 'camino' del gradiente desde la capa siguiente (h2).

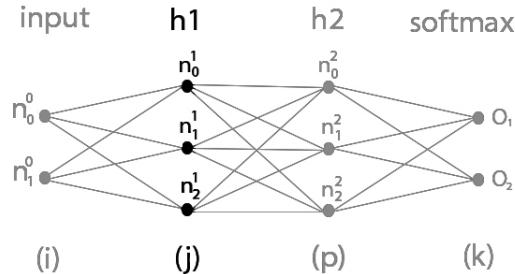


Figura 3.17: Retropropagación en la capa oculta h1

En este caso, en la capa oculta h1 se emplea la función de activación ReLU, y su derivada viene dada por la fórmula 3.70.

$$ReLU(x) = \max(0, x) \quad (3.69)$$

$$ReLU'(x) = 1 \text{ si } x > 0, 0 \text{ en caso contrario} \quad (3.70)$$

Esta se emplea para obtener el gradiente  $\frac{\partial z_j^1}{\partial a_j^1}$ , así como seguir la retropropagación por la capa, tal y como se muestra en las siguientes fórmulas (3.71, 3.72, y 3.73).

$$\frac{\partial z_j^1}{\partial a_j^1} = 1 \text{ si } x > 0, \ 0 \text{ en caso contrario} \quad (3.71)$$

$$\frac{\partial E_{total}}{\partial a_j^1} = \sum_{p=1}^P \text{gradiente\_h2}_p * W_{jp}^1 * \text{ReLU}'(a_j^1) \quad (3.72)$$

$$\frac{\partial E_{total}}{\partial a_j^1} = \text{gradiente\_h1}_j \quad (3.73)$$

Una vez más, el proceso de obtención de la fórmula 3.72 es muy parecido al realizado para casos anteriores aunque esta capa sea algo “nuevo” respecto a la sección anterior.

### Pesos capa input-h1

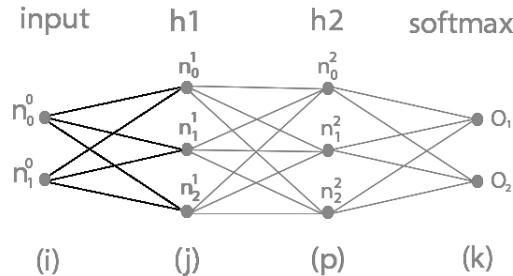


Figura 3.18: Retropropagación respecto a los pesos entre la capa de entrada (input) y la capa oculta h1

$$\frac{\partial a_j^1}{\partial W_{ij}^0} = \frac{\partial([\sum_{c=1}^I z_c^0 * W_{cj}^0] + b_j^1)}{\partial W_{ij}^0} = z_i^0 \quad (3.74)$$

$$\frac{\partial E}{\partial W_{ij}^0} = \frac{\partial E_{total}}{\partial a_j^1} * \frac{\partial a_j^1}{\partial W_{ij}^0} \quad (3.75)$$

$$\frac{\partial E(x)}{\partial W_{ij}^0} = \text{gradiente\_h1}_j * \frac{\partial a_j^1}{\partial W_{ij}^0} = \text{gradiente\_h1}_j * z_i^0 \quad (3.76)$$

Aquí se aprecia como las fórmulas {3.74, 3.75, 3.76 } son iguales a {3.39, 3.40, 3.41 } excepto por el subíndice empleado ( $j \neq p$ ). Aunque los valores

## 6Q. Retropropagación en redes neuronales totalmente conectadas

analíticos no sean los mismos pues las arquitecturas son diferentes, la notación empleada se ha creado con el objetivo de facilitar la visualización y comprensión de la gran capacidad de automatización en capas totalmente conectadas.

### Sesgos capa h1

$$\frac{\partial E}{\partial b_j^1} = \frac{\partial E_{total}}{\partial a_j^1} * \frac{\partial a_j^1}{\partial b_j^1} \quad (3.77)$$

$$\frac{\partial a_j^1}{\partial b_j^1} = \frac{\partial([\sum_{c=1}^I z_c^0 * W_{ij}^0] + b_j^1)}{\partial b_j^1} = 1 \quad (3.78)$$

$$\frac{\partial E}{\partial b_j^1} = \text{gradiente\_h1}_j \quad (3.79)$$

De esta forma, las fórmulas {3.77, 3.78, 3.79 } y {3.42, 3.43, 3.44 } también coincide en todo menos en el subíndice ( $j \neq p$ ).

### Capa input

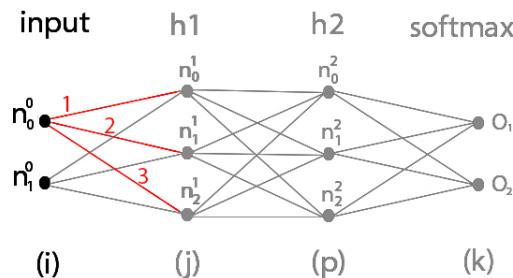


Figura 3.19: 'Caminos' desde la capa oculta h1 hasta  $n_0^0$

$$\frac{\partial E_{total}}{\partial a_i^0} = \sum_{j=1}^J \frac{\partial E_{total}}{\partial a_j^1} * \frac{\partial a_j^1}{\partial z_i^0} * \frac{\partial z_i^0}{\partial a_i^0} \quad (3.80)$$

$$\frac{\partial a_j^1}{\partial z_i^0} = \frac{\partial([\sum_{c=1}^I z_c^0 * W_{ij}^0] + b_j^1)}{\partial z_i^0} = W_{ij}^0 \quad (3.81)$$

Aquí también se aprecia como a diferencia de los subíndices empleados, las fórmulas {3.80, 3.81 } y {3.45, 3.46} son exactamente iguales.

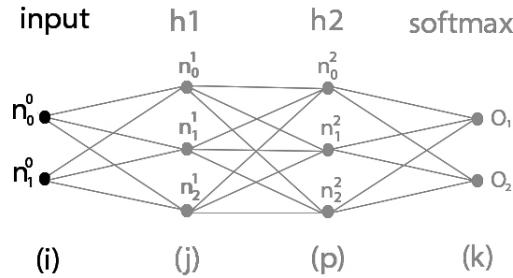


Figura 3.20: Retropopagación en la capa input

Como la capa input no presenta ninguna función de activación asociada,  $z_i^0$  es igual a  $a_i^0$ , (igual que en el caso anterior).

$$\frac{\partial z_i^0}{\partial a_i^0} = 1 \quad (3.82)$$

$$\frac{\partial E_{total}}{\partial a_i^0} = \sum_{p=1}^P \text{gradiente\_} h1_p \quad (3.83)$$

### 3.2.4. Conclusiones

Se definen como capas ocultas “intermedias” todas menos la última de ellas. Tal y como se ha mostrado anteriormente, comparten la mayoría del cálculo en cuanto a retropopagación. De esta forma, se puede dividir una red neuronal totalmente conectada en 4 grupos {capa input, capas ocultas intermedias, última capa oculta, capa de salida o capa softmax}.

A continuación se realiza el cálculo necesario para la retropopagación de una capa de neuronas ‘l’ determinada. Suponemos que la capa  $l+1$  tiene Q neuronas, la capa  $l-1$  tiene K neuronas, y todas las capas ocultas intermedias usan ReLU como función de activación.

## 622. Retropropagación en redes neuronales rotalmente conectadas

### Gradiente respecto a la entrada de la capa

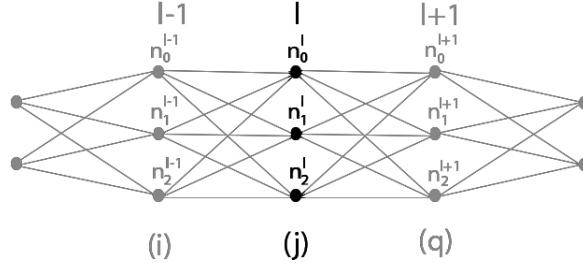


Figura 3.21: Retropropagación en la capa l

$$\frac{\partial E_{total}}{\partial a_j^l} = \sum_{q=1}^Q \frac{\partial E_{total}}{\partial a_q^{l+1}} * \frac{\partial a_q^{l+1}}{\partial z_j^l} * \frac{\partial z_j^l}{\partial a_j^l} \quad (3.84)$$

$$\frac{\partial a_j^{l+1}}{\partial z_j^l} = \frac{\partial([\sum_{c=1}^K z_c^l * W_{ij}^l] + b_j^{l+1})}{\partial z_j^l} = W_{ij}^l \quad (3.85)$$

$$\frac{\partial z_j^l}{\partial a_j^l} = \text{ReLU}'(a_j^l) \quad (3.86)$$

$$\frac{\partial E_{total}}{\partial a_j^l} = \sum_{q=1}^Q \text{gradiente\_} h_{l+1q} * W_{ij}^l * \text{ReLU}'(a_j^l) \quad (3.87)$$

$$\frac{\partial E_{total}}{\partial a_j^l} = \text{gradiente\_} h_{lj} \quad (3.88)$$

Las fórmulas 3.84, 3.85, 3.86, 3.87, y 3.88 muestran el cálculo genérico requerido para obtener el gradiente de la pérdida respecto a la entrada de una capa oculta intermedia ‘l’.

### Gradiente respecto a los pesos

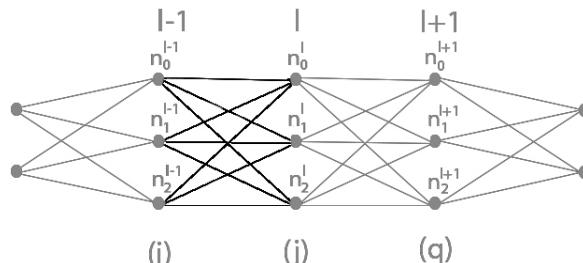


Figura 3.22: Retropropagación respecto a los pesos entre la capa l-1 y l

$$\frac{\partial E}{\partial W_{ij}^{l-1}} = \frac{\partial E_{total}}{\partial a_j^l} * \frac{\partial a_j^l}{W_{ij}^{l-1}} \quad (3.89)$$

$$\frac{\partial a_j^l}{\partial W_{ij}^{l-1}} = \frac{\partial([\sum_{c=1}^K z_c^{l-1} * W_{cj}^{l-1}] + b_j^l)}{\partial W_{ij}^{l-1}} = z_i^{l-1} \quad (3.90)$$

$$\frac{\partial E(x)}{\partial W_{ij}^{l-1}} = \text{gradiente\_} h_{l_j} * \frac{\partial a_j^l}{\partial W_{ij}^{l-1}} = \text{gradiente\_} h_{l_j} * z_i^{l-1} \quad (3.91)$$

Las fórmulas 3.89, 3.90, y 3.91 muestran el cálculo requerido para obtener el gradiente de la pérdida respecto a los pesos entre las capas ocultas genéricas l y l-1.

### Gradiente respecto a sesgos

$$\frac{\partial E}{\partial b_j^l} = \frac{\partial E_{total}}{\partial a_j^l} * \frac{\partial a_j^l}{b_j^l} \quad (3.92)$$

$$\frac{\partial a_j^l}{\partial b_j^l} = \frac{\partial([\sum_{c=1}^K z_c^{l-1} * W_{ij}^{l-1}] + b_j^l)}{\partial b_j^l} = 1 \quad (3.93)$$

$$\frac{\partial E}{\partial b_j^l} = \text{gradiente\_} h_{l_j} \quad (3.94)$$

Las fórmulas 3.92, 3.93, y 3.94 muestran el cálculo requerido para obtener el gradiente de la pérdida respecto a los sesgos de la capa oculta genérica l.

### 3.3. Paralelización mediante OpenMP

#### Tipos de paralelismo

El entrenamiento de una red neuronal convolucional (CNN) se puede paralelizar de distintas formas. Si el modelo se reparte entre varios ordenadores que son entrenados con los mismos datos, se denomina **paralelismo del modelo** (una capa por computador, por ejemplo). Sin embargo, si se distribuyen los datos entre múltiples nodos pero se emplea el mismo modelo para entrenar, se denomina **paralelismo de datos**.

### Paralelismo en SGD

---

**Algorithm 2** Descenso del gradiente estocástico
 

---

Datos de entrenamiento  $D = \{(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)\}$ .

**for** cada trabajador  $t = 0, \dots, T - 1$  en paralelo **do**

**for** época  $p \in \{0, \dots, P - 1\}$  **do**

Desordenar vector de datos D.

**for** cada mini batch  $m = 0, \dots, M - 1$  **do**

Iniciarizar  $\text{gradientes}^t$  a 0.

Reparto de datos del batch m al trabajador t

Realizar propagación hacia delante

Obtener error total con la función de pérdida

Realizar propagación hacia detrás y obtener  $\text{gradientes}^t$  de cada parámetro del modelo.

Acumular gradientes obtenidos por cada trabajador t.

Actualizar parámetros.

**end for**

**end for**

**end for**

---

La naturaleza iterativa del algoritmo del descenso del gradiente estocástico puede parecer un obstáculo ante la paralelización del entrenamiento del modelo, pues la iteración i se basa en el resultado obtenido en la iteración i-1. Sin embargo, tal y como se indica en [72], [73], y [74], existe una forma de aplicar paralelismo en cada iteración.

En cada época se entrena al modelo con M subconjuntos de  $N_m$  datos disjuntos entre ellos de forma que, dados T “trabajadores” o procesos paralelos, se puede dividir cada mini-batch a su vez en T subconjuntos de  $\frac{N_m}{T}$  datos y asignar cada uno a un trabajador distinto.

Siguiendo el mismo razonamiento, se reparten los datos de entrenamiento entre los distintos trabajadores T tanto para realizar la propagación hacia delante como para la posterior retropropagación. En el caso de la retropropagación, se deberá acumular el gradiente de la pérdida respecto a cada parámetro obtenido por cada trabajador, y una vez en posesión de dicho gradiente ‘total’ se procederá a la actualización de los parámetros del modelo.

### 3.4. Retropropagación en redes neuronales convolucionales

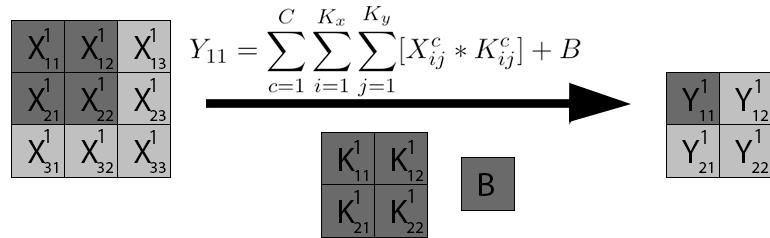


Figura 3.23: Ejemplo de propagación hacia delante en una capa convolucional

La figura 3.23 muestra el ejemplo de propagación hacia delante que se empleará en esta sección. En él,  $C$  indica el número de canales de profundidad del volumen de entrada  $X$ , mientras que  $K_x$  y  $K_y$  hacen referencia al número de filas y columnas del kernel  $K$  empleado, respectivamente.

Siguiendo la notación empleada en secciones anteriores, se denotará como  $A_{ij}^c$  al valor  $X_{ij}^c$  antes de aplicar sobre él su función de activación asociada, y  $Z_{ij}^c$  una vez esta sea aplicada.

#### Sumatoria de gradientes

$$\frac{\partial E}{\partial K_{11}^c} = \sum_{i=1}^I \sum_{j=1}^J \left[ \frac{\partial E}{\partial Y_{ij}^c} * \frac{\partial Y_{ij}^c}{\partial K_{11}^c} \right] \quad (3.95)$$

$$\frac{\partial E}{\partial X_{11}^c} = \sum_{i=1}^I \sum_{j=1}^J \left[ \frac{\partial E}{\partial Y_{ij}^c} * \frac{\partial Y_{ij}^c}{\partial Z_{11}^c} * \frac{\partial Z_{11}^c}{\partial A_{11}^c} \right] \quad (3.96)$$

Para calcular el gradiente de la función de error respecto a cada peso  $K_{xy}$  o entrada  $X_{xy}$ , se deberá realizar una sumatoria del mismo respecto a cada valor de salida de dicha convolución. En el caso de los pesos de un canal de profundidad  $c \in C$ , cada uno se empleó en el cálculo de cada valor  $Y_{ij}^c$ . En el caso de la entrada, hay valores  $x \in X$  que se emplearon en el cálculo de distintos  $\{y_1, y_2\} \in Y$ . Esto se vio anteriormente con detalle en la sección 2.5.1.

**Gradiente de  $Y_{11}^c$** 

$$Y_{11}^c = Z_{11}^c * K_{11}^c + Z_{12}^c * K_{12}^c + Z_{21}^c * K_{21}^c + Z_{22}^c * K_{22}^c \quad (3.97)$$

$$\frac{\partial Y_{11}^c}{\partial K_{xy}^c} = \frac{\partial(Z_{11}^c * K_{11}^c + Z_{12}^c * K_{12}^c + Z_{21}^c * K_{21}^c + Z_{22}^c * K_{22}^c)}{\partial K_{xy}^c} \quad (3.98)$$

$$\frac{\partial Y_{11}^c}{\partial K_{11}^c} = Z_{11}^c, \quad \frac{\partial Y_{11}^c}{\partial K_{12}^c} = Z_{12}^c \quad (3.99)$$

$$\frac{\partial Y_{11}^c}{\partial K_{21}^c} = Z_{21}^c, \quad \frac{\partial Y_{11}^c}{\partial K_{22}^c} = Z_{22}^c \quad (3.100)$$

$$\frac{\partial Y_{11}^c}{\partial Z_{11}^c} = K_{11}^c, \quad \frac{\partial Y_{11}^c}{\partial Z_{12}^c} = K_{12}^c, \quad \frac{\partial Y_{11}^c}{\partial Z_{13}^c} = 0 \quad (3.101)$$

$$\frac{\partial Y_{11}^c}{\partial Z_{21}^c} = K_{21}^c, \quad \frac{\partial Y_{11}^c}{\partial Z_{22}^c} = K_{22}^c, \quad \frac{\partial Y_{11}^c}{\partial Z_{23}^c} = 0 \quad (3.102)$$

$$\frac{\partial Y_{11}^c}{\partial Z_{31}^c} = 0, \quad \frac{\partial Y_{11}^c}{\partial Z_{32}^c} = 0, \quad \frac{\partial Y_{11}^c}{\partial Z_{33}^c} = 0 \quad (3.103)$$

La fórmula 3.97 muestra una descomposición de  $Y_{11}^c$  en términos de  $Z$  y  $K$ . Esto resulta útil para calcular tanto el gradiente respecto a  $Z$  (fórmulas 3.101, 3.102, 3.103) como respecto a  $K$  (fórmulas 3.98, 3.99, 3.100). De esta forma, se calcula el gradiente de  $Y_{11}^c$  respecto a cada parámetro de la capa convolucional.

Cabe destacar que para realmente calcular el gradiente respecto a cada valor del volumen de entrada ( $X$ ), se debería calcular también la derivada de la función de activación asociada a dicha capa. Es decir,  $\frac{\partial Z}{\partial A}$ . Sin embargo, como este proceso ya se ha visto varias veces en secciones anteriores, se omitirá (por ahora) junto con el cálculo del gradiente respecto al sesgo de cada capa. El objetivo de ello reside en la eliminación de cálculos redundantes y centrar la atención en los aspectos importantes y novedosos. Aun así, todos los cálculos mostrados en esta documentación (y más) se encuentran en el código correspondiente, pues se recuerda que todo este conocimiento se ha llevado a la práctica y por ello comprobado su correcto funcionamiento.

Además, como todo se ha hecho a mano por la misma persona, la mayoría de las variables y/o índices coinciden a la perfección o son muy parecidas en la documentación y en el código, por lo que cualquier lector con conocimientos básicos de programación podría entender gran parte de las implementaciones desarrolladas en este proyecto.

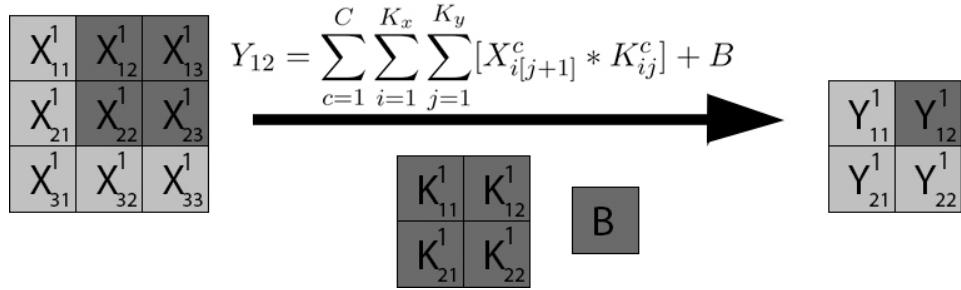
Gradiente de  $Y_{12}^c$ 

Figura 3.24: Cálculo de  $Y_{12}^c$  mediante propagación hacia delante en una capa convolucional

Del mismo modo, se calcula el gradiente de  $Y_{12}^c$  respecto a cada peso (3.106 y 3.107)

$$Y_{12}^c = Z_{12}^c * K_{11}^c + Z_{13}^c * K_{12}^c + Z_{22}^c * K_{21}^c + Z_{23}^c * K_{22}^c \quad (3.104)$$

$$\frac{\partial Y_{12}^c}{\partial K_{xy}^c} = \frac{\partial (Z_{11}^c * K_{11}^c + Z_{12}^c * K_{12}^c + Z_{21}^c * K_{21}^c + Z_{22}^c * K_{22}^c)}{\partial K_{xy}^c} \quad (3.105)$$

$$\frac{\partial Y_{12}^c}{\partial K_{11}^c} = Z_{12}^c, \quad \frac{\partial Y_{12}^c}{\partial K_{12}^c} = Z_{13}^c \quad (3.106)$$

$$\frac{\partial Y_{12}^c}{\partial K_{21}^c} = Z_{22}^c, \quad \frac{\partial Y_{12}^c}{\partial K_{22}^c} = Z_{23}^c \quad (3.107)$$

y valor de entrada (3.108, 3.109 y 3.110).

$$\frac{\partial Y_{12}^c}{\partial Z_{11}^c} = 0, \quad \frac{\partial Y_{12}^c}{\partial Z_{12}^c} = K_{11}^c, \quad \frac{\partial Y_{12}^c}{\partial Z_{13}^c} = K_{12}^c \quad (3.108)$$

$$\frac{\partial Y_{12}^c}{\partial Z_{21}^c} = 0, \quad \frac{\partial Y_{12}^c}{\partial Z_{22}^c} = K_{21}^c, \quad \frac{\partial Y_{12}^c}{\partial Z_{23}^c} = K_{22}^c \quad (3.109)$$

$$\frac{\partial Y_{12}^c}{\partial Z_{31}^c} = 0, \quad \frac{\partial Y_{12}^c}{\partial Z_{32}^c} = 0, \quad \frac{\partial Y_{12}^c}{\partial Z_{33}^c} = 0 \quad (3.110)$$

Gradiente de  $Y_{21}^c$

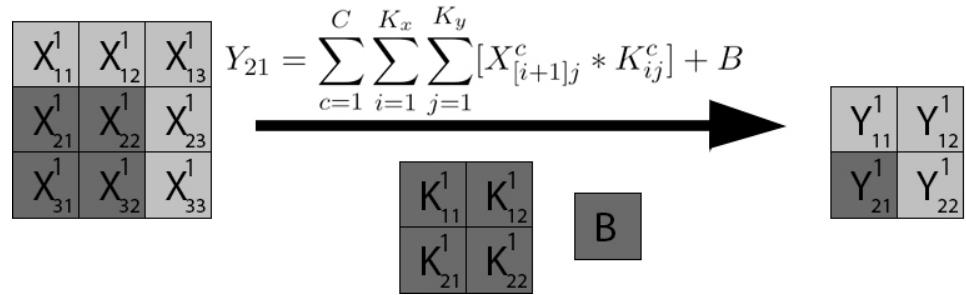


Figura 3.25: Cálculo de  $Y_{21}^c$  mediante propagación hacia delante en una capa convolucional

Se calcula el gradiente de  $Y_{21}^c$  respecto a cada peso (3.113 y 3.114)

$$Y_{21}^c = Z_{21}^c * K_{11}^c + Z_{22}^c * K_{12}^c + Z_{31}^c * K_{21}^c + Z_{32}^c * K_{22}^c \quad (3.111)$$

$$\frac{\partial Y_{21}^c}{\partial K_{xy}^c} = \frac{\partial (Z_{21}^c * K_{11}^c + Z_{22}^c * K_{12}^c + Z_{31}^c * K_{21}^c + Z_{32}^c * K_{22}^c)}{\partial K_{xy}^c} \quad (3.112)$$

$$\frac{\partial Y_{21}^c}{\partial K_{11}^c} = Z_{21}^c, \quad \frac{\partial Y_{21}^c}{\partial K_{12}^c} = Z_{22}^c \quad (3.113)$$

$$\frac{\partial Y_{21}^c}{\partial K_{21}^c} = Z_{31}^c, \quad \frac{\partial Y_{21}^c}{\partial K_{22}^c} = Z_{32}^c \quad (3.114)$$

y valor de entrada (3.115, 3.116 y 3.117).

$$\frac{\partial Y_{21}^c}{\partial Z_{11}^c} = 0, \quad \frac{\partial Y_{21}^c}{\partial Z_{12}^c} = 0, \quad \frac{\partial Y_{21}^c}{\partial Z_{13}^c} = 0 \quad (3.115)$$

$$\frac{\partial Y_{21}^c}{\partial Z_{21}^c} = K_{11}^c, \quad \frac{\partial Y_{21}^c}{\partial Z_{22}^c} = K_{12}^c, \quad \frac{\partial Y_{21}^c}{\partial Z_{23}^c} = 0 \quad (3.116)$$

$$\frac{\partial Y_{21}^c}{\partial Z_{31}^c} = K_{21}^c, \quad \frac{\partial Y_{21}^c}{\partial Z_{32}^c} = K_{22}^c, \quad \frac{\partial Y_{21}^c}{\partial Z_{33}^c} = 0 \quad (3.117)$$

### Gradiente de $Y_{22}^c$

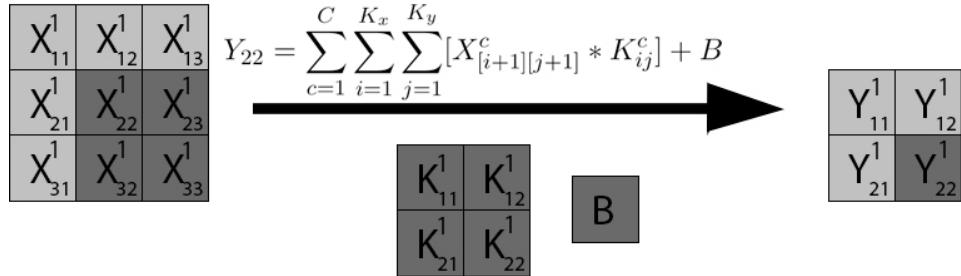


Figura 3.26: Cálculo de  $Y_{22}^c$  mediante propagación hacia delante en una capa convolucional

Se calcula el gradiente de  $Y_{22}^c$  respecto a cada peso (3.120, 3.121).

$$Y_{22}^c = Z_{22}^c * K_{11}^c + Z_{23}^c * K_{12}^c + Z_{32}^c * K_{21}^c + Z_{33}^c * K_{22}^c \quad (3.118)$$

$$\frac{\partial Y_{22}^c}{\partial K_{xy}^c} = \frac{\partial (Z_{11}^c * K_{11}^c + Z_{12}^c * K_{12}^c + Z_{21}^c * K_{21}^c + Z_{22}^c * K_{22}^c)}{\partial K_{xy}^c} \quad (3.119)$$

$$\frac{\partial Y_{22}^c}{\partial K_{11}^c} = Z_{22}^c, \quad \frac{\partial Y_{22}^c}{\partial K_{12}^c} = Z_{23}^c \quad (3.120)$$

$$\frac{\partial Y_{22}^c}{\partial K_{21}^c} = Z_{32}^c, \quad \frac{\partial Y_{22}^c}{\partial K_{22}^c} = Z_{33}^c \quad (3.121)$$

Se calcula el gradiente de  $Y_{22}^c$  respecto a cada valor del volumen de entrada (3.122, 3.123, y 3.124).

$$\frac{\partial Y_{22}^c}{\partial Z_{11}^c} = 0, \quad \frac{\partial Y_{22}^c}{\partial Z_{12}^c} = 0, \quad \frac{\partial Y_{22}^c}{\partial Z_{13}^c} = 0 \quad (3.122)$$

$$\frac{\partial Y_{22}^c}{\partial Z_{21}^c} = 0, \quad \frac{\partial Y_{22}^c}{\partial Z_{22}^c} = K_{11}^c, \quad \frac{\partial Y_{22}^c}{\partial Z_{23}^c} = K_{12}^c \quad (3.123)$$

$$\frac{\partial Y_{22}^c}{\partial Z_{31}^c} = 0, \quad \frac{\partial Y_{22}^c}{\partial Z_{32}^c} = K_{21}^c, \quad \frac{\partial Y_{22}^c}{\partial Z_{33}^c} = K_{22}^c \quad (3.124)$$

### Gradiente respecto a pesos como convolución

Finalmente, se calcula la sumatoria total de gradientes respecto a cada peso de la capa (3.125, 3.126, 3.127, 3.128) y se observa un claro patrón.

$$\frac{\partial E}{\partial K_{11}^c} = \frac{\partial E}{\partial Y_{11}^c} * Z_{11}^c + \frac{\partial E}{\partial Y_{12}^c} * Z_{12}^c + \frac{\partial E}{\partial Y_{21}^c} * Z_{21}^c + \frac{\partial E}{\partial Y_{22}^c} * Z_{22}^c \quad (3.125)$$

$$\frac{\partial E}{\partial K_{12}^c} = \frac{\partial E}{\partial Y_{11}^c} * Z_{12}^c + \frac{\partial E}{\partial Y_{12}^c} * Z_{13}^c + \frac{\partial E}{\partial Y_{21}^c} * Z_{22}^c + \frac{\partial E}{\partial Y_{22}^c} * Z_{23}^c \quad (3.126)$$

$$\frac{\partial E}{\partial K_{21}^c} = \frac{\partial E}{\partial Y_{11}^c} * Z_{21}^c + \frac{\partial E}{\partial Y_{12}^c} * Z_{22}^c + \frac{\partial E}{\partial Y_{31}^c} * Z_{21}^c + \frac{\partial E}{\partial Y_{22}^c} * Z_{32}^c \quad (3.127)$$

$$\frac{\partial E}{\partial K_{22}^c} = \frac{\partial E}{\partial Y_{11}^c} * Z_{22}^c + \frac{\partial E}{\partial Y_{12}^c} * Z_{23}^c + \frac{\partial E}{\partial Y_{31}^c} * Z_{32}^c + \frac{\partial E}{\partial Y_{22}^c} * Z_{33}^c \quad (3.128)$$

Tal y como se observa en los cálculos obtenidos, estos coinciden con una convolución entre la entrada X y el gradiente respecto a la capa de salida Y. Esto se ve con detalle en la figura 3.27 [75].

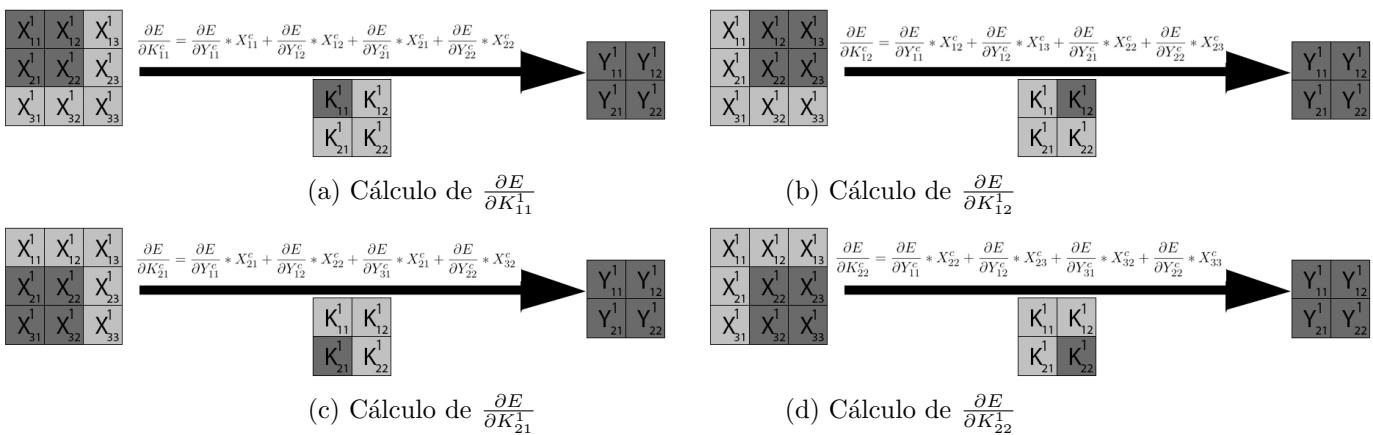


Figura 3.27: Cálculo del gradiente de la pérdida respecto a cada filtro como convolución entre X e Y

En la figura 3.27, cada subfigura  $\{(a), (b), (c), (d)\}$  corresponde al cálculo del gradiente respecto a un peso distinto. Aunque a simple vista parezca algo diferente, los cálculos son exactamente los mismos que los obtenidos anteriormente, solo cambia la visualización de los mismos.

### Gradiente respecto a entrada como convolución

Por simplicidad y seguir tanto la opinión de expertos como a mi experimentación personal, en capas convolucionales se usará ReLU como función de activación. Por tanto, ya se conoce su derivada pues se calculó previamente (3.70).

$$\frac{\partial E}{\partial A_{11}^c} = \frac{\partial E}{\partial Y_{11}^c} * K_{11}^c * \text{ReLU}'(A_{11}^c) \quad (3.129)$$

$$\frac{\partial E}{\partial A_{12}^c} = (\frac{\partial E}{\partial Y_{11}^c} * K_{12}^c + \frac{\partial E}{\partial Y_{12}^c} * K_{11}^c) * \text{ReLU}'(A_{12}^c) \quad (3.130)$$

$$\frac{\partial E}{\partial A_{13}^c} = \frac{\partial E}{\partial Y_{12}^c} * K_{12}^c * \text{ReLU}'(A_{13}^c) \quad (3.131)$$

(3.132)

$$\frac{\partial E}{\partial A_{21}^c} = (\frac{\partial E}{\partial Y_{11}^c} * K_{21}^c + \frac{\partial E}{\partial Y_{21}^c} * K_{11}^c) * \text{ReLU}'(A_{21}^c) \quad (3.133)$$

$$\frac{\partial E}{\partial A_{22}^c} = (\frac{\partial E}{\partial Y_{11}^c} * K_{22}^c + \frac{\partial E}{\partial Y_{12}^c} * K_{21}^c + \frac{\partial E}{\partial Y_{21}^c} * K_{12}^c + \frac{\partial E}{\partial Y_{22}^c} * K_{11}^c) * \text{ReLU}'(A_{22}^c) \quad (3.134)$$

$$\frac{\partial E}{\partial A_{23}^c} = (\frac{\partial E}{\partial Y_{12}^c} * K_{22}^c + \frac{\partial E}{\partial Y_{22}^c} * K_{12}^c) * \text{ReLU}'(A_{22}^c) \quad (3.135)$$

(3.136)

$$\frac{\partial E}{\partial A_{31}^c} = \frac{\partial E}{\partial Y_{21}^c} * K_{21}^c * \text{ReLU}'(A_{31}^c) \quad (3.137)$$

$$\frac{\partial E}{\partial A_{32}^c} = (\frac{\partial E}{\partial Y_{21}^c} * K_{22}^c + \frac{\partial E}{\partial Y_{22}^c} * K_{21}^c) * \text{ReLU}'(A_{32}^c) \quad (3.138)$$

$$\frac{\partial E}{\partial A_{33}^c} = \frac{\partial E}{\partial Y_{22}^c} * K_{22}^c * \text{ReLU}'(A_{33}^c) \quad (3.139)$$

Tal y como se observa en los cálculos obtenidos, estos coinciden con una convolución tipo completa o “full” entre el gradiente respecto a la capa de salida Y y los pesos K invertidos tanto horizontal como verticalmente. El cálculo del gradiente respecto a cada valor  $x \in X$  se ve con detalle en la figura 3.29, mientras que la forma de invertir los pesos se muestra en la figura 3.28 [75].

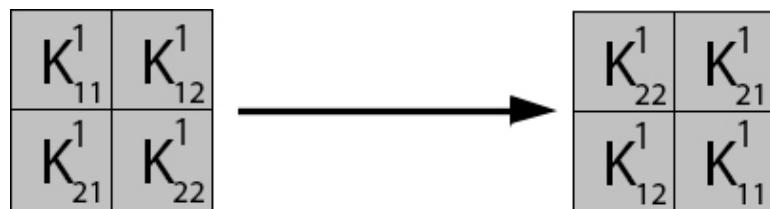


Figura 3.28: Invertir pesos en K tanto horizontal como verticalmente

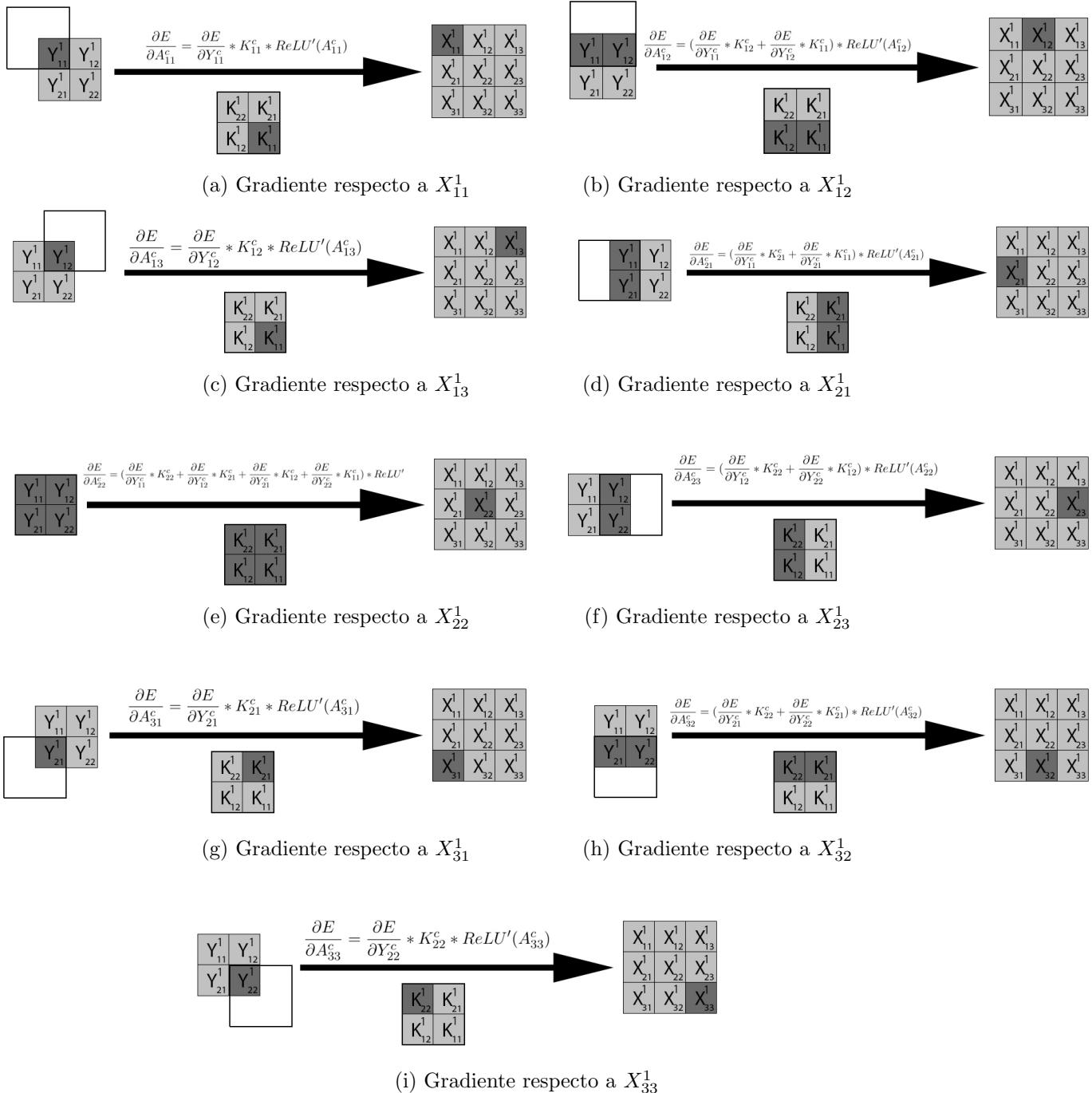


Figura 3.29: Cálculo del gradiente de la pérdida respecto a cada valor de entrada como convolución entre K e Y

Una vez más, los cálculos mostrados en la Figura 3.29 coinciden perfectamente con los obtenidos anteriormente, pues son los mismos. Lo único que cambia es la visualización de los mismos para conseguir una mejor comprensión a escala general y permitir una obtención automática de estos, permitiendo con ello una cómoda implementación en código.

### 3.4.1. Retropropagación con relleno

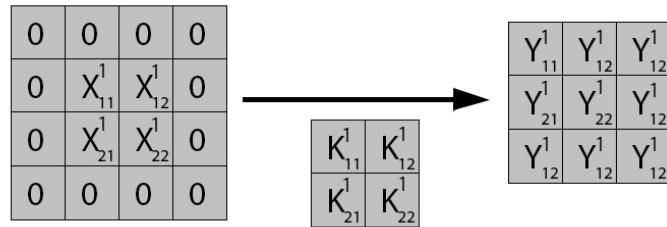


Figura 3.30: Ejemplo de retropropagación en una capa convolucional con relleno

Al igual que en el apartado anterior, se realizará y mostrará la retropropagación respecto a una capa convolucional. La principal diferencia en este caso consiste en la presencia de relleno en la capa, tal y como se muestra en el ejemplo a emplear dado por la Figura 3.30. Por tanto, a continuación se calculará el gradiente de la función de error respecto a los pesos y los valores de entrada de la capa.

#### Gradiente de $Y_{11}^c$

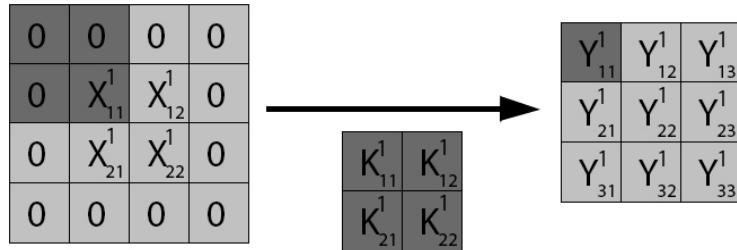


Figura 3.31: Retropropagación de  $Y_{11}^c$

Para facilitar la compresión del lector, se mantendrá la misma estructura y notación que en el apartado anterior.

Así, se calcula el gradiente de  $Y_{11}^c$  respecto a cada peso mediante las fórmulas 3.142 y 3.142,

$$Y_{11}^c = Z_{11}^c * K_{22}^c \quad (3.140)$$

$$\frac{\partial Y_{11}^c}{\partial K_{xy}^c} = \frac{\partial (Z_{11}^c * K_{22}^c)}{\partial K_{xy}^c} \quad (3.141)$$

$$\frac{\partial Y_{11}^c}{\partial K_{11}^c} = 0, \quad \frac{\partial Y_{11}^c}{\partial K_{12}^c} = 0 \quad (3.142)$$

$$\frac{\partial Y_{11}^c}{\partial K_{21}^c} = 0, \quad \frac{\partial Y_{11}^c}{\partial K_{22}^c} = Z_{11}^c \quad (3.143)$$

así como el gradiente de  $Y_{11}^c$  respecto a cada valor del volumen de entrada mediante 3.144 y 3.145.

$$\frac{\partial Y_{11}^c}{\partial Z_{11}^c} = K_{22}^c, \quad \frac{\partial Y_{11}^c}{\partial Z_{12}^c} = 0 \quad (3.144)$$

$$\frac{\partial Y_{11}^c}{\partial Z_{21}^c} = 0, \quad \frac{\partial Y_{11}^c}{\partial Z_{22}^c} = 0 \quad (3.145)$$

#### Gradiente de $Y_{12}^c$

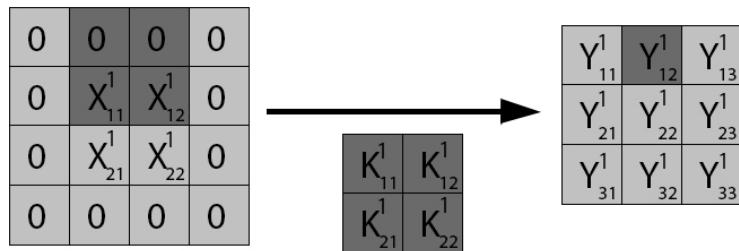


Figura 3.32: Retropropagación de  $Y_{12}^c$

Se calcula el gradiente de  $Y_{12}^c$  respecto a cada peso mediante 3.148 y 3.149,

$$Y_{12}^c = Z_{11}^c * K_{21}^c + Z_{12}^c * K_{22}^c \quad (3.146)$$

$$\frac{\partial Y_{12}^c}{\partial K_{xy}^c} = \frac{\partial (Z_{11}^c * K_{21}^c + Z_{12}^c * K_{22}^c)}{\partial K_{xy}^c} \quad (3.147)$$

$$\frac{\partial Y_{12}^c}{\partial K_{11}^c} = 0, \quad \frac{\partial Y_{12}^c}{\partial K_{12}^c} = 0 \quad (3.148)$$

$$\frac{\partial Y_{12}^c}{\partial K_{21}^c} = Z_{11}^c, \quad \frac{\partial Y_{12}^c}{\partial K_{22}^c} = Z_{12}^c \quad (3.149)$$

así como el gradiente de  $Y_{12}^c$  respecto a cada valor del volumen de entrada (3.150, 3.151).

$$\frac{\partial Y_{12}^c}{\partial Z_{11}^c} = K_{21}^c, \quad \frac{\partial Y_{12}^c}{\partial Z_{12}^c} = K_{22}^c \quad (3.150)$$

$$\frac{\partial Y_{12}^c}{\partial Z_{21}^c} = 0, \quad \frac{\partial Y_{12}^c}{\partial Z_{22}^c} = 0 \quad (3.151)$$

### Gradiente de $Y_{13}^c$

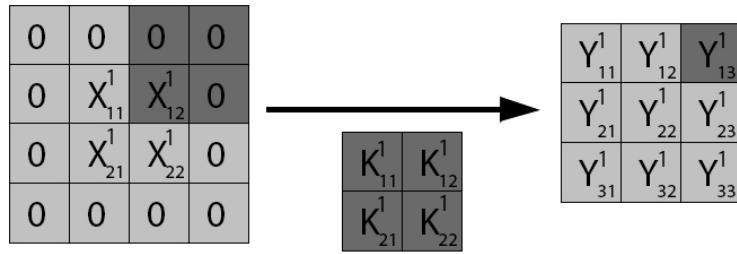


Figura 3.33: Retropropagación de  $Y_{13}^c$

Se calcula el gradiente de  $Y_{13}^c$  respecto a cada peso mediante 3.154 y 3.4.1,

$$Y_{13}^c = Z_{12}^c * K_{21}^c \quad (3.152)$$

$$\frac{\partial Y_{13}^c}{\partial K_{xy}^c} = \frac{\partial(Z_{12}^c * K_{21}^c)}{\partial K_{xy}^c} \quad (3.153)$$

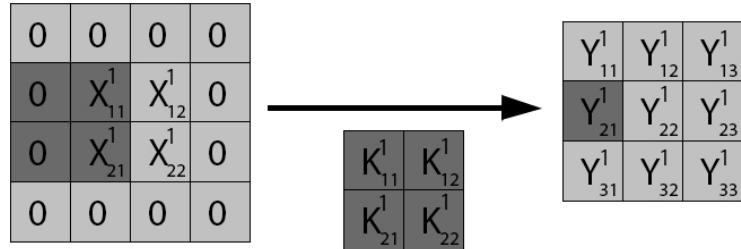
$$\frac{\partial Y_{13}^c}{\partial K_{11}^c} = 0, \quad \frac{\partial Y_{13}^c}{\partial K_{12}^c} = 0 \quad (3.154)$$

$$\frac{\partial Y_{13}^c}{\partial K_{21}^c} = Z_{12}^c, \quad \frac{\partial Y_{13}^c}{\partial K_{22}^c} = 0 \quad (3.155)$$

así como el gradiente de  $Y_{13}^c$  respecto a cada valor del volumen de entrada (3.156, 3.157).

$$\frac{\partial Y_{13}^c}{\partial Z_{11}^c} = 0, \quad \frac{\partial Y_{13}^c}{\partial Z_{12}^c} = K_{21}^c \quad (3.156)$$

$$\frac{\partial Y_{13}^c}{\partial Z_{21}^c} = 0, \quad \frac{\partial Y_{13}^c}{\partial Z_{22}^c} = 0 \quad (3.157)$$

Gradiente de  $Y_{21}^c$ Figura 3.34: Retropropagación de  $Y_{21}^c$ 

Se calcula el gradiente de  $Y_{21}^c$  respecto a cada peso mediante 3.160 y 3.161,

$$Y_{21}^c = Z_{11}^c * K_{12}^c + Z_{21}^c * K_{22}^c \quad (3.158)$$

$$\frac{\partial Y_{21}^c}{\partial K_{xy}^c} = \frac{\partial(Z_{11}^c * K_{12}^c + Z_{21}^c * K_{22}^c)}{\partial K_{xy}^c} \quad (3.159)$$

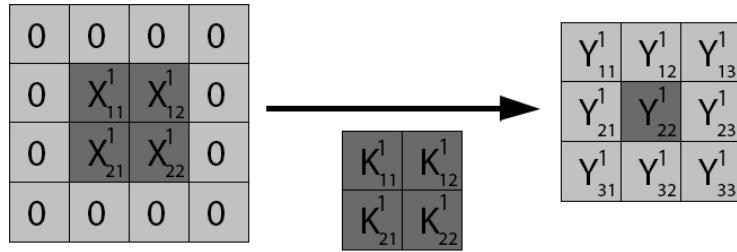
$$\frac{\partial Y_{21}^c}{\partial K_{11}^c} = 0, \quad \frac{\partial Y_{21}^c}{\partial K_{12}^c} = Z_{11}^c \quad (3.160)$$

$$\frac{\partial Y_{21}^c}{\partial K_{21}^c} = 0, \quad \frac{\partial Y_{21}^c}{\partial K_{22}^c} = Z_{21}^c \quad (3.161)$$

así como el gradiente de  $Y_{21}^c$  respecto a cada valor del volumen de entrada (3.162, 3.4.1).

$$\frac{\partial Y_{21}^c}{\partial Z_{11}^c} = K_{12}^c, \quad \frac{\partial Y_{21}^c}{\partial Z_{12}^c} = 0 \quad (3.162)$$

$$\frac{\partial Y_{21}^c}{\partial Z_{21}^c} = K_{22}^c, \quad \frac{\partial Y_{21}^c}{\partial Z_{22}^c} = 0 \quad (3.163)$$

**Gradiente de  $Y_{22}^c$** Figura 3.35: Retropropagación de  $Y_{22}^c$ 

Se calcula el gradiente de  $Y_{22}^c$  respecto a cada peso mediante 3.166 y 3.167,

$$Y_{22}^c = Z_{11}^c * K_{11}^c + Z_{12}^c * K_{12}^c + Z_{21}^c * K_{21}^c + Z_{22}^c * K_{22}^c \quad (3.164)$$

$$\frac{\partial Y_{22}^c}{\partial K_{xy}^c} = \frac{\partial (Z_{11}^c * K_{11}^c + Z_{12}^c * K_{12}^c + Z_{21}^c * K_{21}^c + Z_{22}^c * K_{22}^c)}{\partial K_{xy}^c} \quad (3.165)$$

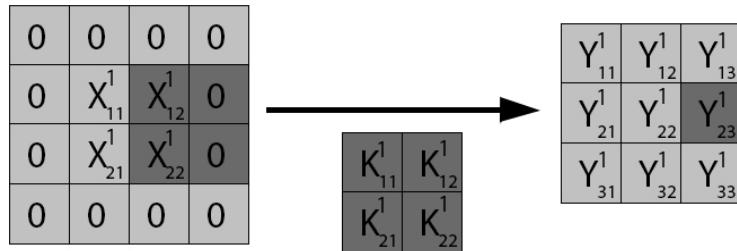
$$\frac{\partial Y_{22}^c}{\partial K_{11}^c} = Z_{11}^c, \quad \frac{\partial Y_{22}^c}{\partial K_{12}^c} = Z_{12}^c \quad (3.166)$$

$$\frac{\partial Y_{22}^c}{\partial K_{21}^c} = Z_{21}^c, \quad \frac{\partial Y_{22}^c}{\partial K_{22}^c} = Z_{22}^c \quad (3.167)$$

así como el gradiente de  $Y_{22}^c$  respecto a cada valor del volumen de entrada (3.168, 3.169).

$$\frac{\partial Y_{22}^c}{\partial Z_{11}^c} = K_{11}^c, \quad \frac{\partial Y_{22}^c}{\partial Z_{12}^c} = K_{12}^c \quad (3.168)$$

$$\frac{\partial Y_{22}^c}{\partial Z_{21}^c} = K_{21}^c, \quad \frac{\partial Y_{22}^c}{\partial Z_{22}^c} = K_{22}^c. \quad (3.169)$$

**Gradiente de  $Y_{23}^c$** Figura 3.36: Retropropagación de  $Y_{23}^c$

Se calcula el gradiente de  $Y_{23}^c$  respecto a cada peso mediante 3.172 y 3.173,

$$Y_{23}^c = Z_{12}^c * K_{11}^c + Z_{22}^c * K_{21}^c \quad (3.170)$$

$$\frac{\partial Y_{23}^c}{\partial K_{xy}^c} = \frac{\partial(Z_{12}^c * K_{11}^c + Z_{22}^c * K_{21}^c)}{\partial K_{xy}^c} \quad (3.171)$$

$$\frac{\partial Y_{23}^c}{\partial K_{11}^c} = Z_{12}^c, \quad \frac{\partial Y_{23}^c}{\partial K_{12}^c} = 0 \quad (3.172)$$

$$\frac{\partial Y_{23}^c}{\partial K_{21}^c} = Z_{22}^c, \quad \frac{\partial Y_{23}^c}{\partial K_{22}^c} = 0 \quad (3.173)$$

así como el gradiente de  $Y_{23}^c$  respecto a cada valor del volumen de entrada (3.168, 3.169).

$$\frac{\partial Y_{23}^c}{\partial Z_{11}^c} = 0, \quad \frac{\partial Y_{23}^c}{\partial Z_{12}^c} = K_{11}^c \quad (3.174)$$

$$\frac{\partial Y_{23}^c}{\partial Z_{21}^c} = 0, \quad \frac{\partial Y_{23}^c}{\partial Z_{22}^c} = K_{21}^c \quad (3.175)$$

#### Gradiente de $Y_{31}^c$

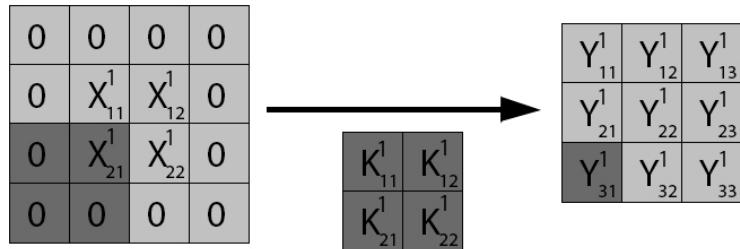


Figura 3.37: Retropropagación de  $Y_{31}^c$

Se calcula el gradiente de  $Y_{31}^c$  respecto a cada peso mediante 3.178 y 3.179,

$$Y_{31}^c = Z_{21}^c * K_{12}^c \quad (3.176)$$

$$\frac{\partial Y_{31}^c}{\partial K_{xy}^c} = \frac{\partial(Z_{21}^c * K_{12}^c)}{\partial K_{xy}^c} \quad (3.177)$$

$$\frac{\partial Y_{31}^c}{\partial K_{11}^c} = 0, \quad \frac{\partial Y_{31}^c}{\partial K_{12}^c} = Z_{21}^c \quad (3.178)$$

$$\frac{\partial Y_{31}^c}{\partial K_{21}^c} = 0, \quad \frac{\partial Y_{31}^c}{\partial K_{22}^c} = 0 \quad (3.179)$$

así como el gradiente de  $Y_{31}^c$  respecto a cada valor del volumen de entrada (3.180, 3.181).

$$\frac{\partial Y_{31}^c}{\partial Z_{11}^c} = 0, \quad \frac{\partial Y_{31}^c}{\partial Z_{12}^c} = 0 \quad (3.180)$$

$$\frac{\partial Y_{31}^c}{\partial Z_{21}^c} = K_{12}^c, \quad \frac{\partial Y_{31}^c}{\partial Z_{22}^c} = 0 \quad (3.181)$$

### Gradiente de $Y_{32}^c$

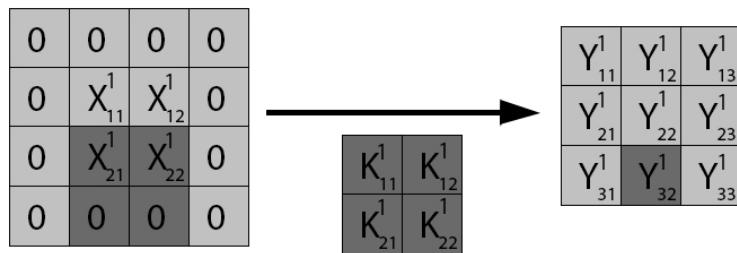


Figura 3.38: Retropropagación de  $Y_{32}^c$

Se calcula el gradiente de  $Y_{32}^c$  respecto a cada peso mediante 3.184 y 3.185,

$$Y_{32}^c = Z_{21}^c * K_{11}^c + Z_{22}^c * K_{12}^c \quad (3.182)$$

$$\frac{\partial Y_{32}^c}{\partial K_{xy}^c} = \frac{\partial(Z_{21}^c * K_{11}^c + Z_{22}^c * K_{12}^c)}{\partial K_{xy}^c} \quad (3.183)$$

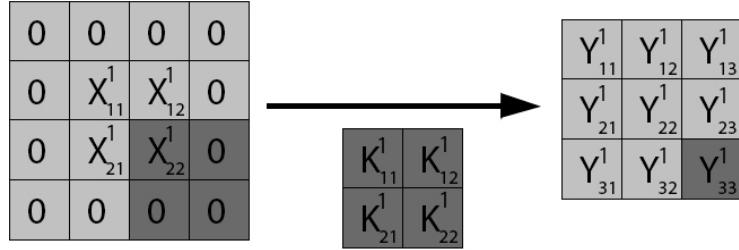
$$\frac{\partial Y_{32}^c}{\partial K_{11}^c} = Z_{21}^c, \quad \frac{\partial Y_{32}^c}{\partial K_{12}^c} = Z_{22}^c \quad (3.184)$$

$$\frac{\partial Y_{32}^c}{\partial K_{21}^c} = 0, \quad \frac{\partial Y_{32}^c}{\partial K_{22}^c} = 0 \quad (3.185)$$

así como el gradiente de  $Y_{32}^c$  respecto a cada valor del volumen de entrada (3.186, 3.187).

$$\frac{\partial Y_{32}^c}{\partial Z_{11}^c} = 0, \quad \frac{\partial Y_{32}^c}{\partial Z_{12}^c} = 0 \quad (3.186)$$

$$\frac{\partial Y_{32}^c}{\partial Z_{21}^c} = K_{11}^c, \quad \frac{\partial Y_{32}^c}{\partial Z_{22}^c} = K_{12}^c \quad (3.187)$$

**Gradiente de  $Y_{33}^c$** Figura 3.39: Retropropagación de  $Y_{33}^c$ 

Se calcula el gradiente de  $Y_{33}^c$  respecto a cada peso mediante 3.190 y 3.191,

$$Y_{33}^c = Z_{22}^c * K_{11}^c \quad (3.188)$$

$$\frac{\partial Y_{33}^c}{\partial K_{xy}^c} = \frac{\partial (Z_{22}^c * K_{11}^c)}{\partial K_{xy}^c} \quad (3.189)$$

$$\frac{\partial Y_{33}^c}{\partial K_{11}^c} = Z_{22}^c, \quad \frac{\partial Y_{33}^c}{\partial K_{12}^c} = 0 \quad (3.190)$$

$$\frac{\partial Y_{33}^c}{\partial K_{21}^c} = 0, \quad \frac{\partial Y_{33}^c}{\partial K_{22}^c} = 0 \quad (3.191)$$

así como el gradiente de  $Y_{33}^c$  respecto a cada valor del volumen de entrada (3.192, 3.193).

$$\frac{\partial Y_{33}^c}{\partial Z_{11}^c} = 0, \quad \frac{\partial Y_{33}^c}{\partial Z_{12}^c} = 0 \quad (3.192)$$

$$\frac{\partial Y_{33}^c}{\partial Z_{21}^c} = 0, \quad \frac{\partial Y_{33}^c}{\partial Z_{22}^c} = K_{11}^c \quad (3.193)$$

**Gradiente respecto a pesos como convolución**

Finalmente, al igual que en el caso anterior, se calcula la sumatoria y con ello el gradiente de la función de pérdida respecto a cada peso  $K_{xy}$ , y una vez más se observa un claro patrón en los resultados obtenidos.



$$\frac{\partial E}{\partial A_{11}^c} = \left( \frac{\partial E}{\partial Y_{11}^c} * K_{22}^c + \frac{\partial E}{\partial Y_{12}^c} * K_{21}^c + \frac{\partial E}{\partial Y_{21}^c} * K_{12}^c + \frac{\partial E}{\partial Y_{22}^c} * K_{11}^c \right) * \text{ReLU}'(A_{11}^c) \quad (3.198)$$

$$\frac{\partial E}{\partial A_{12}^c} = \left( \frac{\partial E}{\partial Y_{12}^c} * K_{22}^c + \frac{\partial E}{\partial Y_{13}^c} * K_{21}^c + \frac{\partial E}{\partial Y_{22}^c} * K_{12}^c + \frac{\partial E}{\partial Y_{23}^c} * K_{11}^c \right) * \text{ReLU}'(A_{12}^c) \quad (3.199)$$

$$\frac{\partial E}{\partial A_{21}^c} = \left( \frac{\partial E}{\partial Y_{21}^c} * K_{22}^c + \frac{\partial E}{\partial Y_{22}^c} * K_{21}^c + \frac{\partial E}{\partial Y_{31}^c} * K_{12}^c + \frac{\partial E}{\partial Y_{32}^c} * K_{11}^c \right) * \text{ReLU}'(A_{21}^c) \quad (3.200)$$

$$\frac{\partial E}{\partial A_{22}^c} = \left( \frac{\partial E}{\partial Y_{22}^c} * K_{22}^c + \frac{\partial E}{\partial Y_{23}^c} * K_{21}^c + \frac{\partial E}{\partial Y_{32}^c} * K_{12}^c + \frac{\partial E}{\partial Y_{33}^c} * K_{11}^c \right) * \text{ReLU}'(A_{22}^c) \quad (3.201)$$

Tal y como se observa en los cálculos obtenidos, estos coinciden con una convolución entre el gradiente respecto a la capa de salida  $Y$  y los pesos  $K$  invertidos tanto horizontal como verticalmente. Esto se ilustra en detalle en la figura 3.41.

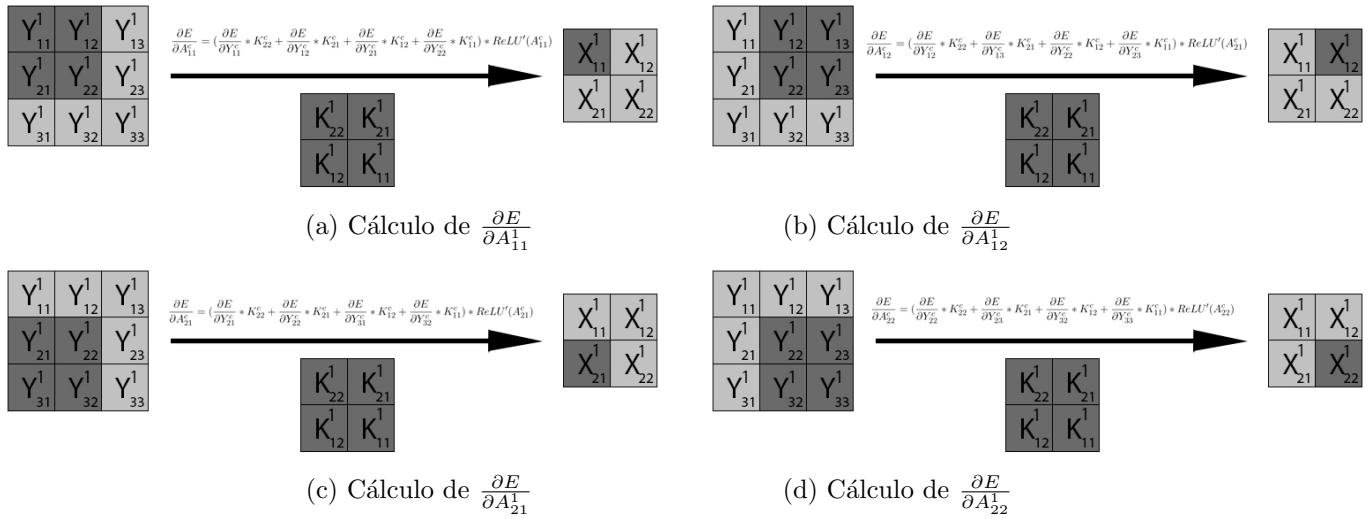
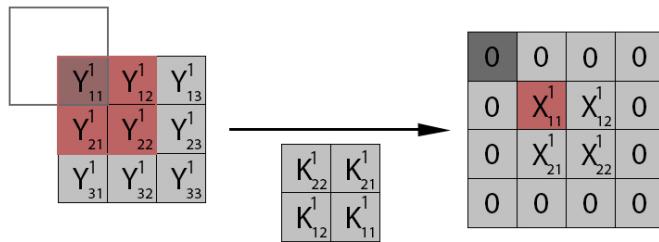
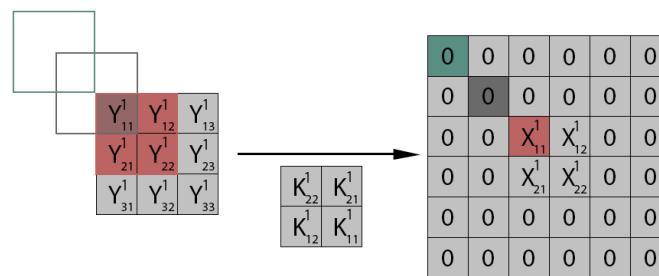


Figura 3.41: Cálculo del gradiente de la pérdida respecto de la entrada como convolución

Una vez desarrolladas ambas retropropagaciones en una capa convolucional (con y sin relleno), se pueden comparar las figuras 3.29 y 3.41 y ver que, aunque ambas calculan el gradiente de la pérdida respecto al volumen de entrada, en una figura se emplea una convolución completa mientras que en la otra no.



(a) Retropropagación con un nivel de relleno



(b) Retropropagación con dos niveles de relleno

Figura 3.42: Cálculo del gradiente de la pérdida respecto a la entrada X con uno o dos niveles de relleno

El motivo de ello se presenta en detalle en la figura 3.42. Al realizar una convolución total, se empiezan a calcular los gradientes comenzando por la esquina superior izquierda de X. Sin embargo, sabiendo que X tiene relleno, no es necesario realizar el cálculo de los gradientes respecto a las posiciones de ‘relleno’ de la entrada pues no se emplean en ningún cálculo posterior.



# Capítulo 4

## Adaptación GPU

Una vez comprendidos los cálculos empleados por una CNN, en esta sección se tratará de introducir como se realizarán los mismos desde una perspectiva heterogénea. Esto es, mediante el uso de GPU para multiplicación matricial.

### 4.1. GEMM

Se trata de un enfoque ampliamente conocido en el mundo de deep learning, empleándolo gran cantidad de librerías del sector como Caffe, Torch-cunn, Theano-CorrMM, o incluso CuDNN [76].

El enfoque GEMM (General Matrix Multiply o Multiplicación General de Matrices) en redes neuronales convolucionales permite reducir el tiempo de cómputo requerido en distintas capas a cambio de aumentar el espacio necesario para la misma.

### 4.2. Convolución como GEMM

En el caso de capas convolucionales, seguir un enfoque GEMM consiste en “desenrollar” tanto el volumen de entrada X como la matriz de filtros K, además de realizar una serie de duplicaciones en X, de tal forma que cada columna de esta nueva matriz 2D X\_unroll contenga todos los elementos de X implicados en el cálculo de una posición distinta del volumen de salida Y. Como los resultados de cada convolución se suman a lo largo de cada canal de profundidad, estos se pueden concatenar en una matriz de gran tamaño. De esta forma, cada kernel de pesos se transforma en una fila de una gran matriz de pesos, tal y como se muestra en la figura 4.1 con los kernels K y G (colores verde y marrón).

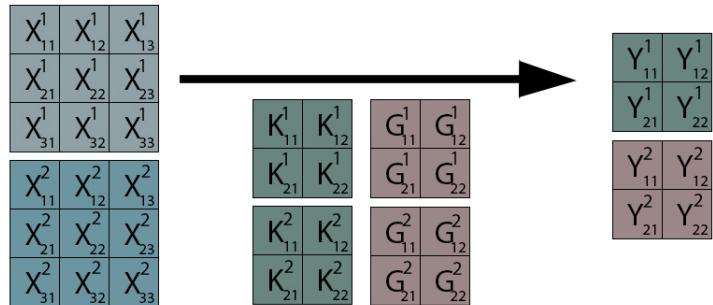
Mientras que el método estándar requiere de varias iteraciones para calcular cada valor del volumen de salida Y (figura 2.10), en la parte inferior de la figura 4.1 se observa como la alternativa GEMM permite que cada valor de

Y sea el resultado de multiplicar una fila de pesos (K o G en la figura 4.1) por una columna de X\_unroll (azul) [11].

De esta forma, con N kernels de tamaño KxK, un volumen de entrada con C canales de profundidad, y un volumen de salida de dimensiones  $N \times H_{out} \times W_{out}$ , una multiplicación matricial entre la matriz de pesos  $M_1$  con N filas y  $K^2 * C$  columnas, y X\_unroll con  $K^2 * C$  filas y  $H_{out} * W_{out}$  columnas, produce el mismo volumen de salida Y que una convolución ordinaria con N kernels distintos.

Por último, aunque se haya omitido para simplificar la comprensión del método planteado, tras realizar dicha multiplicación matricial se debe sumar el sesgo y aplicar la función de activación correspondiente a cada elemento del volumen de salida Y.

## ESTÁNDAR



## GEMM

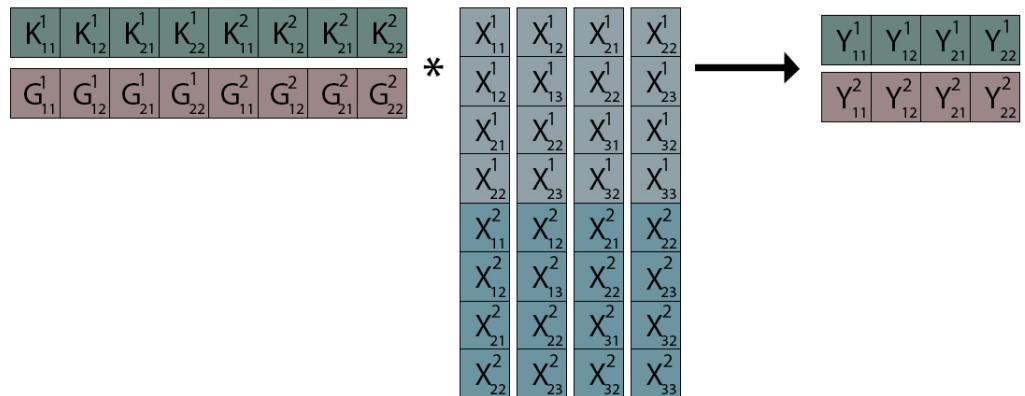


Figura 4.1: Imagen de una convolución estándar frente a una convolución empleando GEMM

#### 4.2.1. Memoria requerida al emplear GEMM

Al realizar una convolución, un mismo filtro de pesos puede iterar más de una vez por varios valores de X. Esto implica la duplicación dichos valores de X en la matriz X\_unroll tantas veces como se requiera el acceso a los mismos. En el caso de una entrada X(3x3) y un kernel de pesos K(2x2) (ejemplo de la figura 4.1), el valor central de X deberá duplicarse 4 veces, mientras que el valor central de cada lateral se duplicará 2 veces y los valores de cada esquina solo requieren 1 acceso por lo que no se duplican. Así, una matriz inicial X(3x3) con 9 valores, se transforma en X\_unroll con  $4^*1 + 2^*4 + 1^*4 = 16$  valores, contando con un ratio de expansión de  $16/9 = 1.8$ .

Como cada valor de salida  $Y_{ij}^m$  es producto de una convolución de  $K^*K$  pesos a lo largo de X sobre C canales de profundidad, el número de columnas de X\_unroll se define por  $C^*K^*K$ . De la misma forma, como el resultado de cada fila multiplicada por cada columna de las matrices “desenrolladas” aportan un valor  $Y_{ij}^m$ , X\_unrolled tiene tantas columnas como Y elementos ( $M * H_{out} * W_{out}$ ).

El ratio de expansión se define mediante  $\frac{C^*K^*K^*H_{out}^*W_{out}}{C^*H^*W}$ , donde H y W hacen referencia a las filas y columnas de X, y  $H_{out}$  y  $W_{out}$  a las filas y columnas de la salida Y respectivamente. En general, si la entrada X y la salida Y poseen unas dimensiones mucho más grandes que el filtro de pesos K, el ratio de expansión será de  $K^*K$ .

Dado que cada kernel de pesos presenta unas dimensiones de  $K^*K$  y se representa como una fila en la matriz total de pesos, esta tendrá  $K^*K$  filas y M columnas, siendo M el número de filtros de pesos a aplicar sobre X.

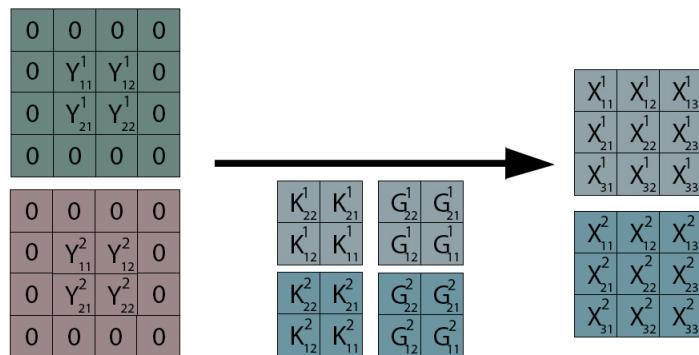
Realizar varias convoluciones sobre una misma entrada X (una con cada filtro de pesos distinto) implica compartir una única matriz X\_unrolled. Sin embargo, para valores comunes de filtros de pesos (5 o superior), una matriz de entrada con dimensiones expandidas con un ratio de  $K^*K$  puede ser excesivamente grande. Como el consumo de memoria por almacenar todas las matrices de entrada expandidas en un minibatch puede suponer un problema, simplemente se reservará un espacio de  $C * K * K * H_{out} * W_{out}$  (una instancia de X\_unrolled), de forma que se reutilice para cada dato del minibatch [11].

### 4.3. Retropropagación GEMM en capa convolucional

En la figura 3.41 se vio como la retropropagación respecto a la entrada en una capa convolucional consistía en una convolución entre los kernels de pesos y el volumen de salida. Del mismo modo, en la figura 3.40 también se mostró como el gradiente respecto a los pesos consistía en una convolución entre el volumen de entrada y el de salida.

Como ambos gradientes se pueden implementar como convoluciones, y en la sección 4.2 se mostró como implementar convoluciones con el enfoque GEMM, tiene sentido asumir que la retropropagación en una capa convolucional se puede implementar mediante el enfoque GEMM.

#### ESTÁNDAR



#### GEMM

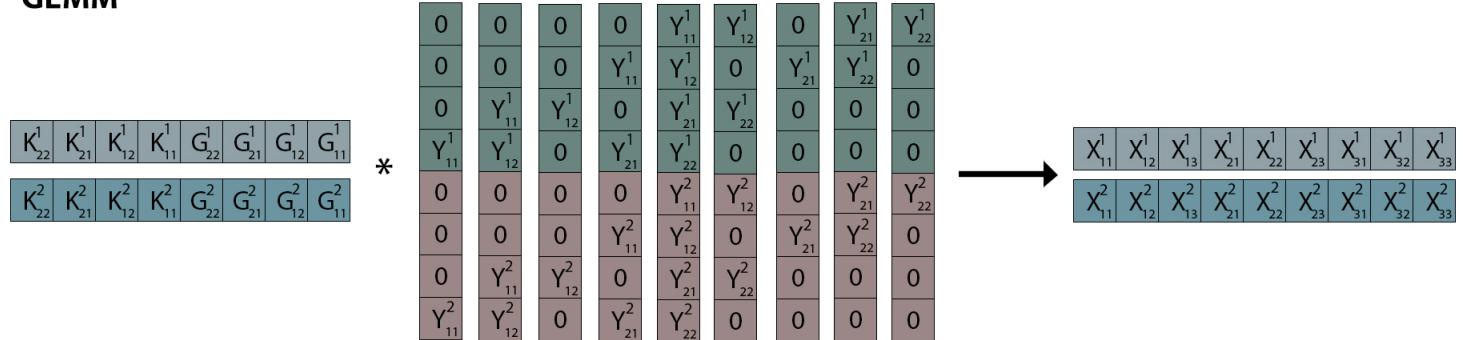
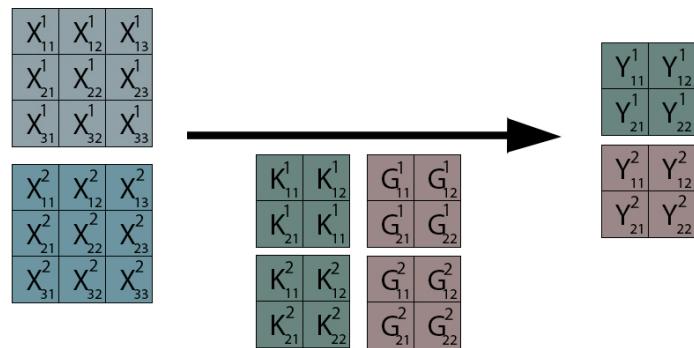


Figura 4.2: Retropropagación en una capa convolucional de forma estándar frente a GEMM respecto a la entrada

En la Figura 4.2 cabe destacar que, como cada canal de profundidad  $c \in C$  de cada kernel solo influye sobre dicho canal  $c$  sobre el volumen de entrada  $X$ , en la retropropagación sólo recibirá el gradiente proveniente de dicho canal. De esta forma, como cada kernel se empleó para producir un canal  $m \in M$  distinto sobre el volumen de salida  $Y$ , se pueden concatenar los  $M$  distintos kernels para un mismo canal de profundidad  $c$ , de forma que cada canal  $c$  de cada kernel  $m$  se multiplique por su subconjunto correspondiente en el canal  $m$  de  $Y$  e influya sobre el canal  $c$  de  $X$  (Figura 4.2).

## ESTÁNDAR



## GEMM

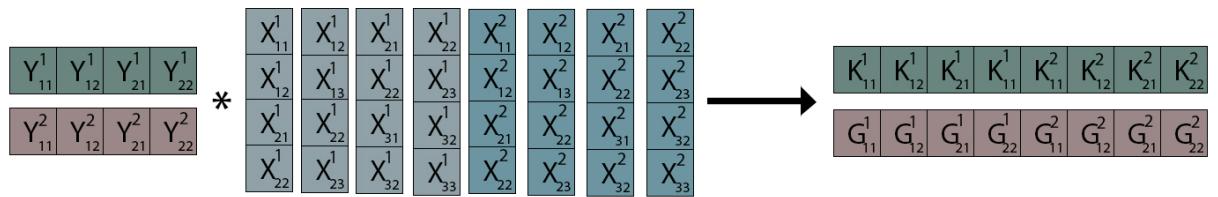


Figura 4.3: Retropropagación en una capa convolucional de forma estándar frente a GEMM respecto a los pesos

En el caso del cálculo del gradiente de la pérdida respecto a los pesos de una capa convolucional, se recuerda que se podía implementar como una convolución entre el volumen de salida y el de entrada. Mediante el enfoque GEMM, se puede observar en la Figura 4.3 como para ello es necesario ‘desenrollar’ tanto el volumen de entrada  $X$  como el de salida  $Y$ . De esta forma, multiplicando cada fila de  $Y$  (desenrollada) por cada columna de  $X$  (desenrollada), se estarán multiplicando todos los elementos del volumen de salida  $Y$  y del volumen de entrada  $X$  que fueron multiplicados anteriormente (en la propagación hacia delante) por un peso distinto, calculando así el gradiente de dicho peso.

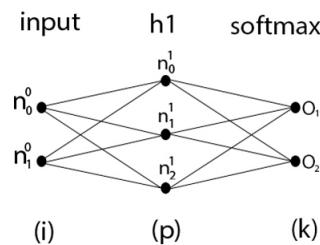
## 4.4. Capa totalmente conectada como GEMM [5]

Al igual que las capas convolucionales, las capas totalmente conectadas también se pueden implementar mediante un enfoque GEMM. Para ello, analizaremos por separado cada uno de los 3 casos que nos encontraremos:

- Propagación hacia delante
- Cálculo del gradiente respecto a la entrada
- Cálculo del gradiente respecto a los pesos

### 4.4.1. Propagación hacia delante

#### ESTÁNDAR



$$a_j^1 = \sum_{i=0}^{i=1} [z_i^0 * w_{i,j}^0] + b_j^1$$

#### GEMM

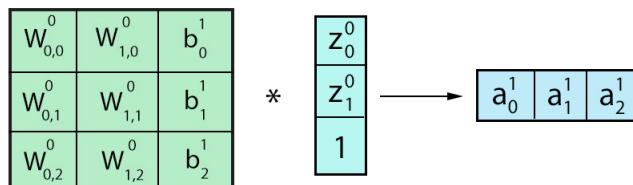


Figura 4.4: Propagación GEMM hacia delante en una capa totalmente conectada

En la Figura 4.4 se muestran 2 formas distintas de implementar la propagación hacia delante en una capa totalmente conectada. El método **ESTÁNDAR** hace referencia al método empleado en secciones anteriores. Sin embargo, el método **GEMM** aporta un nuevo ángulo desde el cual afrontarlo. Este último consiste en, para una capa i, agrupar por un lado pesos y sesgos de dicha capa en una matriz, y por otra parte tener otra matriz de una sola columna con todas las neuronas de dicha capa i más un elemento con valor igual a 1 (para sumar el sesgo).

Siguiendo la estructura planteada en la Figura 4.4, cada multiplicación de

fila `matriz_pesos_sesgos` por columna de `matriz_neuronas` produce como resultado una neurona de la capa  $i+1$  (para completar la propagación hacia delante de dicha capa habría que aplicar la función de activación sobre los resultados obtenidos).

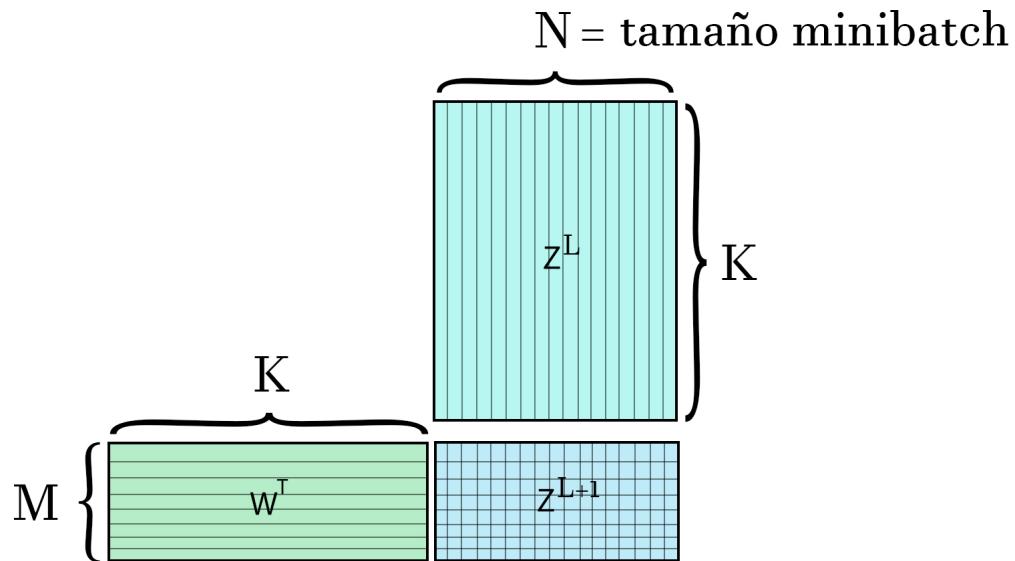


Figura 4.5: Propagación GEMM de un minibatch entero hacia delante en una capa totalmente conectada

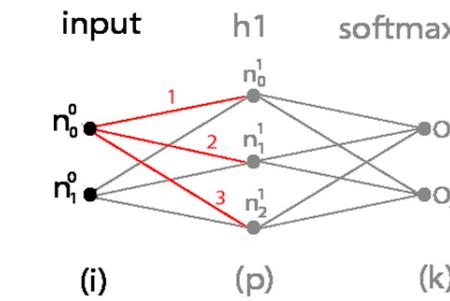
Si tenemos en cuenta que los pesos y sesgos durante el entrenamiento permanecen constantes durante todo un minibatch, para un minibatch de tamaño  $N$ , podemos expandir la matriz `matriz_neuronas` definida anteriormente para que tenga  $N$  columnas, una por cada elemento del minibatch. De esta forma, mediante una simple multiplicación matricial (enfoque GEMM) se puede realizar la propagación hacia delante de un minibatch entero para una capa totalmente conectada, tal y como se muestra en la Figura 4.5 [5].

#### 4.4.2. Retropropagación

Al igual que la propagación hacia delante, la retropropagación en una capa totalmente conectada también puede ser calculada mediante un enfoque de multiplicación matricial como es GEMM.

**Gradiente respecto a la entrada**

#### ESTÁNDAR



$$\frac{\partial E_{total}}{\partial a_i^0} = \sum_{p=1}^2 \left[ \frac{\partial E_{total}}{\partial a_p^1} * W_{ip}^0 \right] * \frac{\partial z_i^0}{\partial a_i^0} = \frac{\partial E_{total}}{\partial z_1^0} * \frac{\partial z_i^0}{\partial a_i^0}$$

#### GEMM

$W_{0,0}^0$	$W_{0,1}^0$	$W_{0,2}^0$
$W_{1,0}^0$	$W_{1,1}^0$	$W_{1,2}^0$

$$\begin{matrix} \frac{\partial E_{total}}{\partial a_0^1} \\ \frac{\partial E_{total}}{\partial a_2^1} \\ \frac{\partial E_{total}}{\partial a_2^1} \end{matrix} *$$

$$\begin{matrix} \frac{\partial E_{total}}{\partial z_0^0} \\ \frac{\partial E_{total}}{\partial z_1^0} \end{matrix}$$

Figura 4.6: Cálculo del gradiente respecto a la entrada en una capa totalmente conectada

Suponiendo que nos encontramos en la capa  $i$  y ya tenemos el gradiente de la pérdida respecto a la entrada de la capa  $i+1$ , podemos calcular dicho gradiente respecto a la entrada de la capa  $i$  tal y como se muestra en la Figura 4.6.

Esto es, teniendo por un lado una matriz `matriz_pesos` donde cada fila  $j$  tiene los pesos que conectan a la neurona  $j$  de dicha capa  $i$  y a cualquier otra neurona de la capa  $i+1$ . En este caso, la matriz `matriz_neuronas` vuelve a tener una sola columna y contiene las neuronas de la capa  $i+1$ . Sin embargo, esta vez dichas neuronas contendrán el gradiente de la pérdida hasta dicha

capa. Así, cada multiplicación de una fila de matriz\_pesos por una columna de matriz\_neuronas producirá el cálculo del gradiente de la pérdida respecto a una neurona de entrada distinta.

En la Figura 4.6 se muestra tanto una representación visual de ambas matrices como la fórmula que se emplea en el método estándar, facilitando así una mejor comprensión de que estos dos métodos son simplemente caminos que llevan al mismo destino y por tanto producen los mismos resultados.

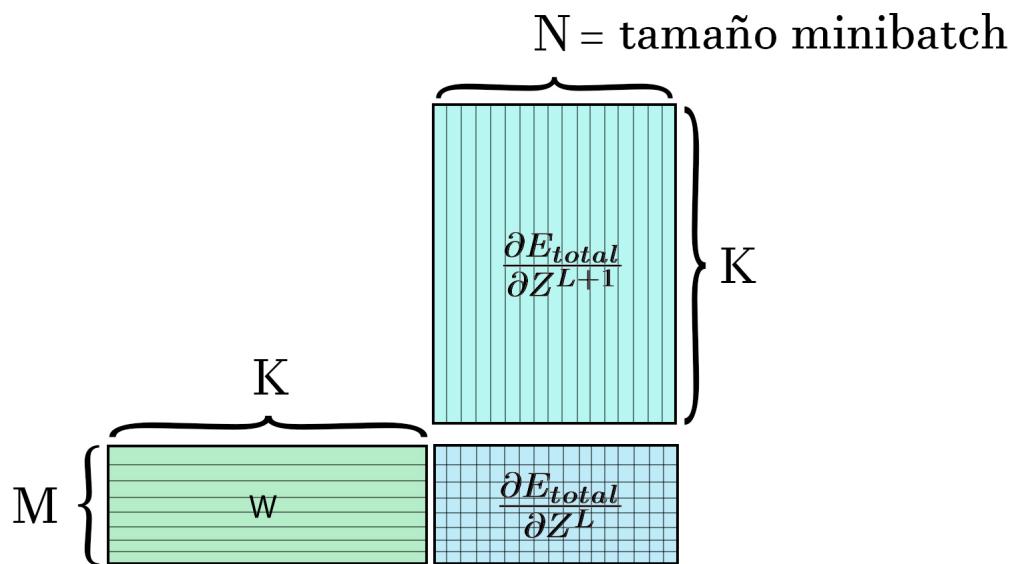
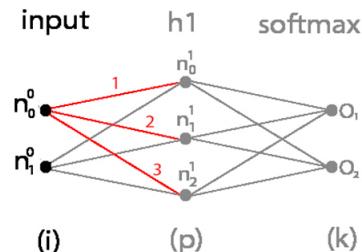


Figura 4.7: Cálculo del gradiente respecto a la entrada de todo un minibatch en una capa totalmente conectada

De forma similar al apartado anterior, para un minibatch de  $N$  elementos, se puede expandir la matriz matriz\_neuronas para que contenga  $N$  columnas, permitiendo así el cálculo del gradiente de la pérdida respecto a la entrada para una capa  $i$  para todo un minibatch mediante una simple multiplicación matricial, tal y como se muestra en la Figura 4.7.

### Gradiente respecto a los pesos

#### ESTÁNDAR



$$\frac{\partial E_{total}}{\partial W_{ip}^0} = \sum_{p=0}^2 \left[ \frac{\partial E_{total}}{\partial a_p^1} * z_i^0 \right]$$

#### GEMM

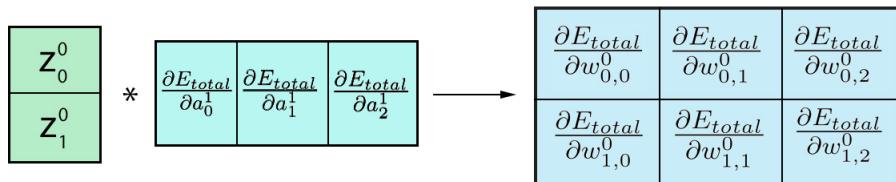


Figura 4.8: Cálculo del gradiente respecto a los pesos en una capa totalmente conectada

En este caso, para una capa  $i$ , tendremos 2 matrices a las que denominaremos matriz\_entrada y matriz\_salida. La primera de ellas se caracterizará por tener una sola columna con las neuronas de entrada de la capa  $i$ , y la segunda contendrá una sola fila con el gradiente de la pérdida respecto a cada neurona de salida de la capa  $i$ . De esta forma, cada multiplicación de fila matriz\_entrada por columna de matriz\_salida producirá el cálculo del gradiente de la pérdida respecto a un peso que une una neurona de la capa  $i$  con otra de la capa  $i+1$ . Es decir, se calcula el gradiente de la pérdida respecto a los pesos de la capa  $i$ , tal y como se muestra en la Figura 4.8.

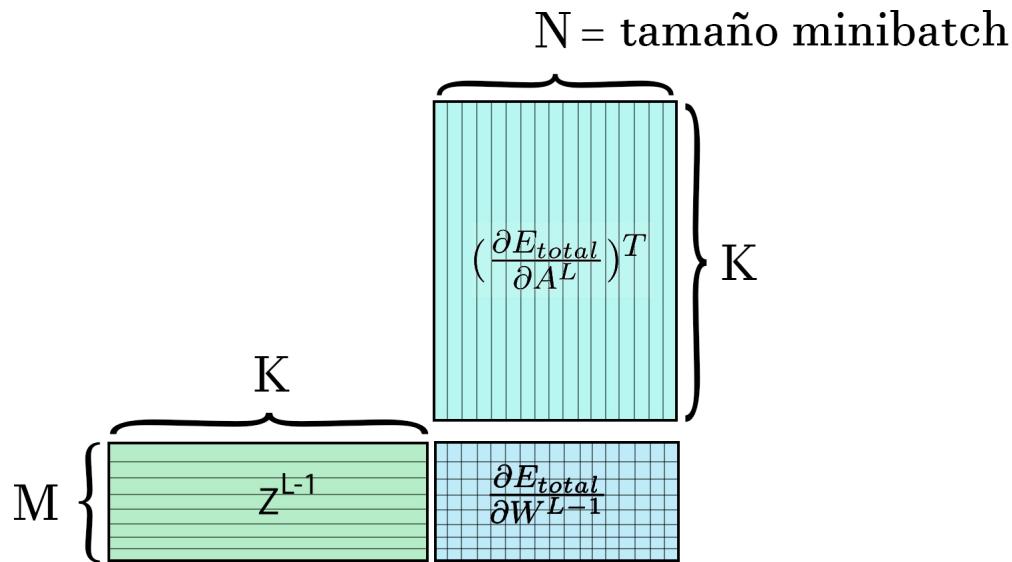


Figura 4.9: Cálculo del gradiente respecto a los pesos de todo un minibatch en una capa totalmente conectada

Para un minibatch de tamaño  $N$ , se pueden expandir ambas matrices de forma que estas tengan  $N$  filas o columnas (según se indica en la Figura 4.9) y calcular el gradiente de la pérdida respecto a los pesos para una capa  $i$  en un minibatch entero mediante una multiplicación matricial.

#### 4.4.3. Multiplicación de matrices en CUDA

Dadas dos matrices  $A(M \times K)$  y  $B(K \times N)$ , se obtiene  $C(K \times N)$  como producto de la multiplicación  $AB$ . Basándome en mi propia experiencia, una primera aproximación de cara a realizar dicha operación en CUDA consiste en crear un bloque  $\text{Block}(K \times N)$ , de forma que cada hebra calcule una posición de la matriz resultado  $C$ . Dada la naturaleza extremadamente simple de tal implementación, el desarrollador que la llevó a cabo optará por buscar formas de reducir el tiempo de cómputo requerido por la misma, optando en la mayoría de los casos por emplear memoria compartida de bloque. Tras desarrollar esta “segunda versión”, el siguiente paso del desarrollador consiste en comparar ambas implementaciones para verificar si efectivamente se obtiene una ganancia notable sobre la primera. En el proceso se observa como para matrices de tamaño reducido ambas implementaciones parecen funcionar. Sin embargo, a medida que aumenta el tamaño de las mismas se ve un claro “tope” pues ambas comparten un mismo defecto, el tamaño de bloque. Por ejemplo, para calcular  $C(50 \times 50) = A(50 \times 10) \times B(10 \times 50)$  ambas implementaciones requerirían un bloque de  $50 \times 50 = 2500$  hebras, lo cual es imposible pues excede el tamaño límite (CUDA solo permite 1024 hebras por bloque).

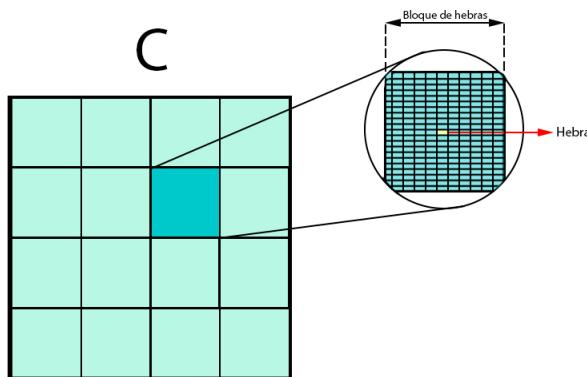


Figura 4.10: Tercera implementación de multiplicación matricial con CUDA

Por tanto, resulta evidente la existencia de una “tercera” implementación carente de este gran defecto. Esta se caracteriza por la división la matriz C en submatrices o tiles, de forma que cada tile sea un bloque cuda. Partiendo de las implementaciones anteriores, el cambio resulta inmediato y los resultados descomunales pues aporta la capacidad de multiplicación matricial con independencia del tamaño de A y B, además de mejoras en cuanto a rendimiento [77].

Sin embargo, esta última implementación carece de los beneficios de la memoria compartida nivel de bloque, por lo que una vez más indica la muy posible existencia de una mejora sobre la misma.

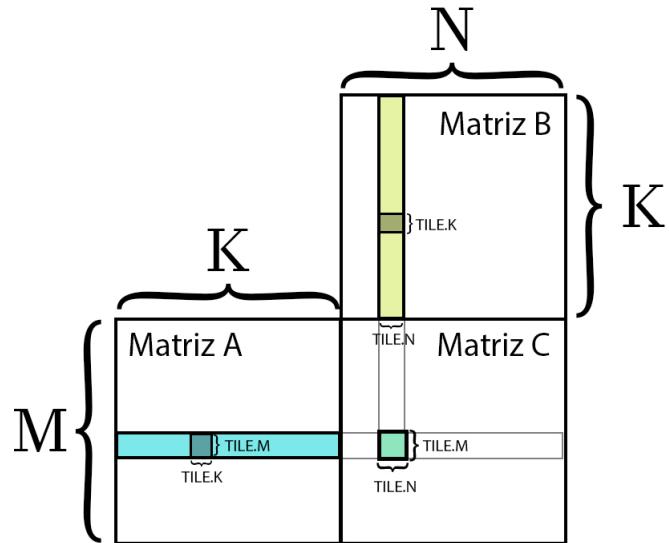


Figura 4.11: Cuarta implementación de multiplicación matricial con CUDA

Para realizar el cálculo de cada valor de C es necesario multiplicar una fila de A por una columna de B. Es decir, multiplicar dos vectores de K elementos. Una idea inicial consiste en cargar 2K elementos en memoria compartida para posteriormente usarlos. Sin embargo, si cada hebra de cada bloque requiere 2K elementos, esto no es viable pues la memoria compartida es limitada y escasa. En su lugar, un mejor enfoque consiste en, dado un bloque 2D de dimensiones  $\text{TILE} \times \text{TILE}$ , almacenar  $2 * \text{TILE} * \text{TILE}$ . Su objetivo es dividir el cálculo de cada valor de C en iteraciones, y en cada una de ellas, cada hebra multiplica  $\text{TILE}$  elementos de A por  $\text{TILE}$  de B, además de acumular y sumar los resultados obtenidos en cada iteración [78].



## Capítulo 5

# Comparación entre distintas implementaciones

### 5.0.1. Modelos a emplear

A continuación se muestra la arquitectura general de algunos de los modelos que se emplearán para tomar medidas de rendimiento y así poder comparar las distintas implementaciones de CNN desarrolladas a lo largo de este proyecto.

#### 1. Modelo 0

- Tamaño imágenes de entrada: 3x32x32
- Capas convolucionales
  - 16 kernels de tamaño 3x3, padding=1
  - 32 kernels de tamaño 3x3, padding=1
- Capas de Agrupación Máxima
  - Kernel de tamaño 2x2
  - Kernel de tamaño 2x2
- Capas totalmente conectadas
  - n\_neuronas\_tras\_flatten
  - 100 neuronas
  - 10 neuronas
- learning rate = 0.001
- 1000 imágenes de entrenamiento
- Tamaño de mini batch = 32

**2. Modelo 1**

- Tamaño imágenes de entrada: 3x40x40
- Capas convolucionales
  - 16 kernels de tamaño 3x3, padding=1
  - 32 kernels de tamaño 3x3, padding=1
- Capas de Agrupación Máxima
  - Kernel de tamaño 2x2
  - Kernel de tamaño 2x2
- Capas totalmente conectadas
  - n\_neuronas\_tras\_flatten
  - 100 neuronas
  - 10 neuronas
- learning rate = 0.001
- 1000 imágenes de entrenamiento
- Tamaño de mini batch = 32

**3. Modelo 2**

- Tamaño imágenes de entrada: 3x40x40
- Capas convolucionales
  - 16 kernels de tamaño 3x3, padding=1
  - 32 kernels de tamaño 3x3, padding=1
  - 32 kernels de tamaño 3x3, padding=1
- Capas de Agrupación Máxima
  - Kernel de tamaño 2x2
  - Kernel de tamaño 2x2
  - Kernel de tamaño 2x2
- Capas totalmente conectadas
  - n\_neuronas\_tras\_flatten
  - 128 neuronas
  - 50 neuronas
  - 10 neuronas
- learning rate = 0.00001
- 2000 imágenes de entrenamiento
- Tamaño de mini batch = 32

### 5.0.2. Comparación de rendimiento

En esta sección se compararán las prestaciones de cada implementación. Como todas y cada una de ellas producen los mismos resultados (cuando se eliminan factores aleatorios como inicialización de pesos, entre otros), dicha comparación se centrará en el tiempo de cómputo requerido por cada una. Se comienza analizando aquellas implementaciones que requieren más tiempo de cómputo. Es decir, las que emplean exclusivamente CPU, (Secuencial y OpenMP). Tras ello, el análisis se posiciona sobre el comportamiento en aquellos sistemas heterogéneos que emplean tanto CPU como GPU (CUDA y cuDNN).

Además, se proporcionan figuras que ayudan a comprender mejor dichos análisis de rendimiento, así como una tabla con los resultados de tiempo exactos empleados por cada capa de una misma arquitectura desarrollada en distintas implementaciones.

#### Implementaciones CPU

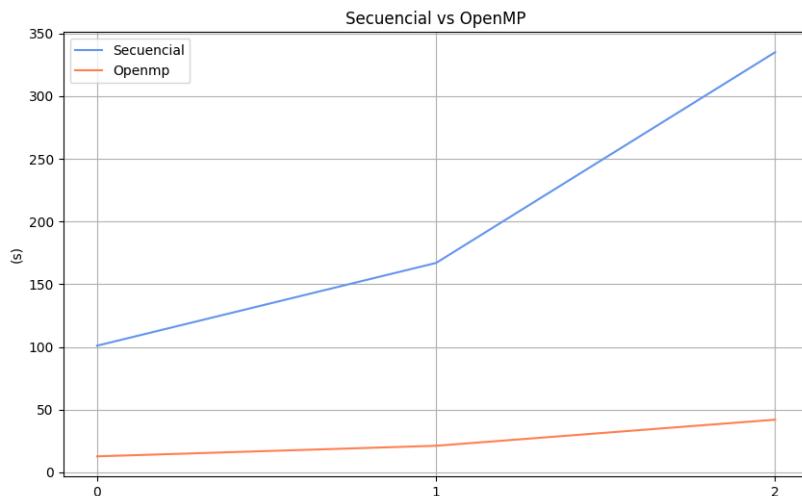


Figura 5.1: Secuencial vs OpenMP

En la Figura 5.1 se muestra una comparación de rendimiento entre la implementación secuencial y la implementación en OpenMP. En el eje Y se muestra el tiempo requerido en el entrenamiento de una época, mientras que en el eje X se muestra el modelo empleado para realizar dicho entrenamiento. De esta forma, el punto  $p1(100, 0)$  indica que el modelo 0 emplea 101 segundos en el entrenamiento de una época según la implementación secuencial, pero 12.76 segundos en la implementación con OpenMP.

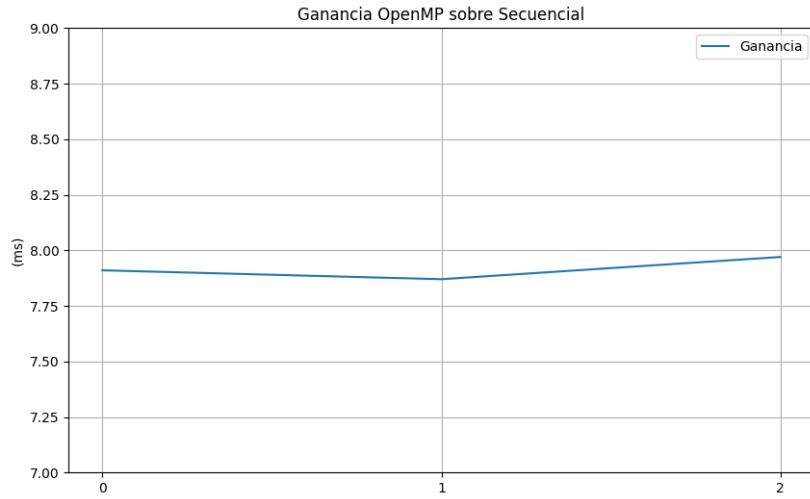


Figura 5.2: Ganancia de OpenMP respecto a secuencial

La implementación con OpenMP se caracteriza por un paralelismo a nivel de CPU mediante 8 hebras. Por tanto, lo esperado es una ganancia cercana a 8 respecto a la implementación secuencial, tal y como se muestra en la Figura 5.2.

### Implementaciones GPU

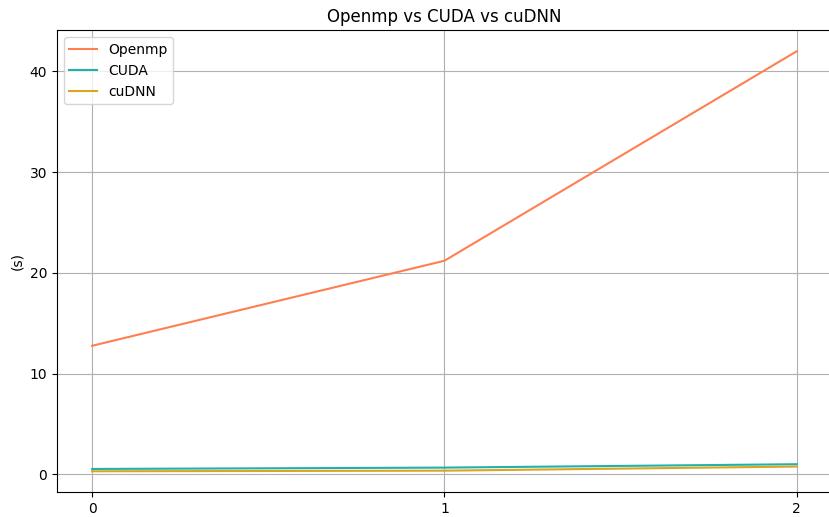


Figura 5.3: OpenMP vs CUDA vs CUDNN

En la Figura 5.3 se emplean los mismos modelos que en las Figuras 5.1 y 5.2. Al igual que la implementación en OpenMP muestra una diferencia notable respecto a la implementación secuencial, esto también es esperable entre implementaciones heterogéneas con CPU-GPU e implementaciones caracterizadas por un uso exclusivo de CPU como es OpenMP en este caso. Esto se observa en la Figura 5.3, donde en este caso las diferencias de rendimiento son incluso mayores que las observadas en el apartado anterior.

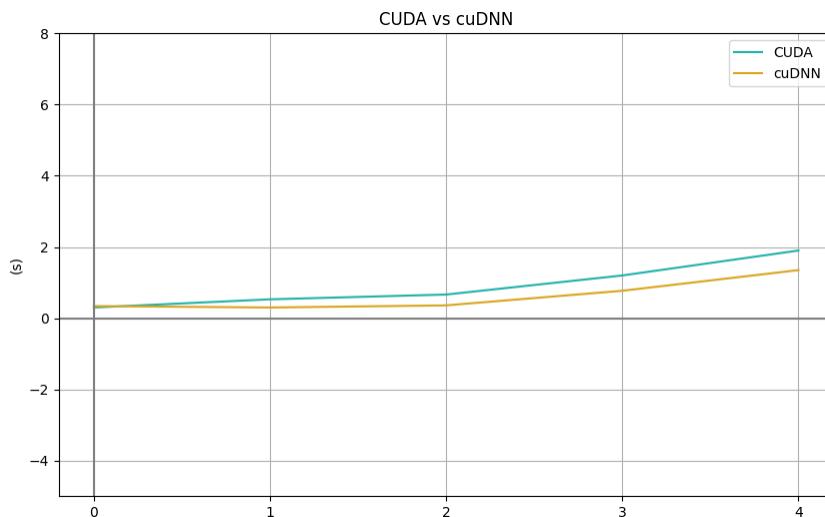


Figura 5.4: CUDA vs CUDNN

Para comparar las implementaciones en GPU se añaden 2 modelos, uno menos y otro más complejo que los anteriores. Al igual que en el experimento anterior, dichos modelos están ordenados en las figuras de izquierda a derecha, de menor a mayor complejidad. El objetivo de ello es facilitar la comprensión entre el rendimiento de la implementación CUDA y la implementación cuDNN.

Tal y como se muestra en la Figura 5.4, la implementación CUDA obtiene mejores resultados en modelos menos complejos, pero a medida que incrementa la complejidad de los mismos, el modelo cuDNN va tomando la ventaja dada la naturaleza de una librería altamente optimizada como se trata de cuDNN. Gracias a esta importante característica se ha empleado en una gran cantidad de librerías de alto nivel y gran prestigio como Caffe2, MATLAB, PyTorch, o TensorFlow, entre otras.

---

Operación	CuDNN (ms)	CUDA (ms)
Conv_fwd_0	0.005	0.009
Conv_back_0	0.032	0.044
Pool_fwd_0	0.003	0.005
Pool_back_0	0.01	0.023
<hr/>		
Conv_fwd_1	0.02	0.022
Conv_back_1	0.065	0.16
Pool_fwd_1	0.0029	0.0039
Pool_back_1	0.014	0.025
<hr/>		
Conv_fwd_2	0.023	0.018
Conv_back_2	0.047	0.018
Pool_fwd_2	0.032	0.023
Pool_back_2	0.0057	0.023

Cuadro 5.1: Comparación rendimiento CuDNN vs CUDA

Además, empleando el modelo con la mayor complejidad de la Figura 5.4, se crea una la Tabla 5.1, mostrando el tiempo requerido en cada capa para realizar tanto la propagación hacia delante como la retropropagación de la misma. En dicha tabla, se observa como a medida que los datos se van procesando por las distintas capas de la CNN, se va reduciendo la ventaja que cuDNN tenía sobre CUDA al inicio de la misma. Esto tiene sentido con lo comentado anteriormente, pues en cada capa se reduce el coste computacional (no siempre es así, pero en este caso concreto sí).

## Capítulo 6

# Conclusiones y trabajo futuro

Tras la comparación de las distintas implementaciones sobre redes neuronales convolucionales realizadas a lo largo de este proyecto, se llega a la conclusión de que la mejor de todas ellas es la que usa cuDNN. Aunque todas las implementaciones producen los mismos resultados y logran aprender tal y como se esperaba, cuDNN logra obtener unos tiempos de cómputo menores al resto. Por tanto, tiene sentido su uso en gran cantidad de librerías de más alto nivel como TensorFlow o Caffe2, entre otras. Además, se han cumplido los objetivos propuestos tal y como se esperaba.

Con respecto al trabajo futuro que se podría realizar, destaca la realización de ciertas mejoras en varios ámbitos:

- Mejorar la implementación en CUDA para intentar reducir la diferencia en el tiempo requerido respecto a cuDNN.
- Realizar distintas implementaciones sobre redes neuronales convolucionales con librerías de alto nivel y comparar los resultados con las implementaciones ya existentes en este proyecto.
- Investigar el funcionamiento de cuDNN a bajo nivel y tratar de mejorarla en algún caso concreto.
- Comparar el rendimiento obtenido con cuDNN respecto al ofrecido por librerías de alto nivel que usen cuDNN por debajo.
- Emplear librerías de bajo nivel como cuDNN para desarrollar sistemas heterogéneos optimizados sobre algoritmos de deep learning más complejos que redes neuronales convolucionales.



# Bibliografía

- [1] mehran@mldawn.com. Back-propagation through cross-entropy softmax, 2021. <https://www.mldawn.com/back-propagation-with-cross-entropy-and-softmax/> [Accessed:29/02/2024].
- [2] mehran@mldawn.com. The derivative of softmax function wrt z, 2021. <https://www.mldawn.com/the-derivative-of-softmaxz-function-w-r-t-z/> [Accessed:05/03/2024].
- [3] Prakash Jay. Back-propagation is very simple. who made it complicated ?, 2017. <https://medium.com/@14prakash/back-propagation-is-very-simple-who-made-it-complicated-97b794c97e5c> [Accessed:06/03/2024].
- [4] Chamanth mvs. No more confusion on back-propagation, 2022. <https://pub.aimind.so/no-more-confusion-on-backpropagation-7adfc271539f> [Accessed:07/03/2024].
- [5] NVIDIA. Linear/fully-connected layers user's guide, 2024. <https://docs.nvidia.com/deeplearning/performance/dl-performance-fully-connected/index.html> [Accessed:11/05/2024].
- [6] Amazon. ¿qué es una unidad central de procesamiento (cpu)?, 2024. <https://aws.amazon.com/es/what-is/cpu/> [Accessed:15/08/2024].
- [7] Amazon. ¿qué es una gpu?, 2024. <https://aws.amazon.com/es/what-is/gpu/> [Accessed:15/08/2024].
- [8] Patricio Bulić. Heterogeneous systems, 2024. <https://doc.sling.si/en/workshops/programming-gpu-cuda/01-intro/02-hetsys/#:~:text=A%20homogeneous%20system%20uses%20one,%2Ds suited%2C%20yielding%20performance%20improvement.> [Accessed:15/08/2024].

- [9] Ty McKercher John Cheng, Max Grossman. *Professional Cuda C Programming*. John Wiley and Sons, Inc., 10475 Crosspoint Boulevard, 1 edition, 2014.
- [10] IBM. What are convolutional neural networks?, 2024. <https://www.ibm.com/topics/convolutional-neural-networks> [Accessed:15/08/2024].
- [11] Izzat El Hajj Wen-emi W.Hwu, David B.kirk. *Programming Massively Parallel Processors*. Morgan Kaufmann, 50 Hampshire Street, 5th Floor, Cambridge, MA 02139, United States, 4 edition, 2022.
- [12] IBM. ¿qué es el machine learning (ml)?, 2024. <https://www.ibm.com/es-es/topics/machine-learning> [Accessed:15/08/2024].
- [13] IBM. ¿qué es el deep learning?, 2024. <https://www.ibm.com/es-es/topics/deep-learning> [Accessed:15/08/2024].
- [14] Mohdsanadzakirizvi. Image classification using cnn, 2024. <https://www.analyticsvidhya.com/blog/2020/02/learn-image-classification-cnn-convolutional-neural-networks-3-datasets/> [Accessed:15/08/2024].
- [15] Microsoft. ¿qué es la visión artificial?, 2024. <https://azure.microsoft.com/es-es/resources/cloud-computing-dictionary/what-is-computer-vision#clasificaci%C3%B3n-de-objetos> [Accessed:15/08/2024].
- [16] Yi Wang Jusong Ren. Overview of object detection algorithms using convolutional neural networks, 2024. <https://www.scirp.org/journal/paperinformation?paperid=115011> [Accessed:15/08/2024].
- [17] Shameerayaseen. U-net: Advancing image segmentation with convolutional neural networks, 2023. <https://medium.com/@shameerayaseen21/u-net-advancing-image-segmentation-with-convolutional-neural-networks-1fd81> [Accessed:15/08/2024].
- [18] Ying Nian Wu Jifeng Dai, Yang Lu. Generative modeling of convolutional neural networks, 2023. <http://www.stat.ucla.edu/~ywu/generativeCNN/main.html> [Accessed:15/08/2024].
- [19] Intel. Convolutional neural networks (cnns), deep learning, and computer vision, 2024. <https://www.intel.com/content/www/us/en/internet-of-things/computer-vision/convolutional-neural-networks.html> [Accessed:15/08/2024].

- [20] Taha Binhuraib. Nlp with cnns, 2020. <https://towardsdatascience.com/nlp-with-cnns-a6aa743bdc1e> [Accessed:15/08/2024].
- [21] Augmented AI. Convolutional neural networks (cnn) in self-driving cars, 2024. <https://www.augmentedstartups.com/blog/convolutional-neural-networks-cnn-in-self-driving-cars#:~:text=They%20have%20the%20potential%20to,decisions%20based%20on%20the%20environment>. [Accessed:15/08/2024].
- [22] OpeMP. Openmp, 2024. <https://www.openmp.org/> [Accessed:15/08/2024].
- [23] NVIDIA. Cuda toolkit, 2024. <https://developer.nvidia.com/cuda-toolkit> [Accessed:15/08/2024].
- [24] NVIDIA. Nvidia cudnn, 2024. <https://developer.nvidia.com/cudnn> [Accessed:02/08/2024].
- [25] Caffe2. What is caffe2?, 2024. <https://caffe2.ai/docs/caffe-migration.html> [Accessed:15/08/2024].
- [26] Keras. Keras, 2024. <https://keras.io/> [Accessed:15/08/2024].
- [27] Mathworks. Matlab, 2024. <https://la.mathworks.com/products/matlab.html> [Accessed:15/08/2024].
- [28] Pytorch. Pytorch, 2024. <https://pytorch.org/> [Accessed:15/08/2024].
- [29] Tensorflow. Tensorflow, 2024. <https://www.tensorflow.org/?hl=es-419> [Accessed:15/08/2024].
- [30] NVIDIA. Nvidia cudnn, 2024. <https://docs.nvidia.com/deeplearning/cudnn/latest/developer/overview.html#dev-overview> [Accessed:02/08/2024].
- [31] Hsuan-Tien Lin Yaser S. Abu-Mostafa, Malik Magdon-Ismail. *Learning From Data*. California Institute of Technology Pasadena, CA 91125, USA, 1 edition, 2012.
- [32] IBM. ¿qué es el aprendizaje supervisado?, 2024. <https://www.ibm.com/es-es/topics/supervised-learning> [Accessed:22/08/2024].
- [33] Interactive Chaos. ¿qué es el aprendizaje supervisado?, 2024. <https://interactivechaos.com/es/manual/tutorial-de-machine-learning/etiquetas> [Accessed:22/08/2024].

- [34] IBM. ¿qué son las redes neuronales?, 2024. <https://www.ibm.com/es-es/topics/neural-networks#:~:text=%C2%BFQu%C3%A9%20son%20las%20redes%20neuronales,opciones%20y%20llegar%20a%20conclusiones>. [Accessed:10/08/2024].
- [35] Sakshi Tiwari. Funciones de activación en redes neuronales, 2024. <https://www.geeksforgeeks.org/activation-functions-neural-networks/> [Accessed:10/08/2024].
- [36] Douglas Karr. Unidad lineal rectificada, 2024. <https://es.martech.zone/acronym/relu/> [Accessed:25/02/2024].
- [37] Jorge Calvo Martin. La importancia de las funciones de activación en una red neuronal, 2022. <https://www.linkedin.com/pulse/la-importancia-de-las-funciones-activaci%C3%B3n-B3n-en-una-red-calvo-martin/> [Accessed:10/06/2024].
- [38] Miguel Sotaquirá. La función de activación, 2018. <https://www.codificandobits.com/blog/funcion-de-activacion/> [Accessed:10/06/2024].
- [39] Javi. La función sigmoide: Una herramienta clave en redes neuronales, 2023. <https://jacar.es/la-funcion-sigmoide-una-herramienta-clave-en-redes-neuronales/> [Accessed:25/02/2024].
- [40] Jason Brownlee. Softmax activation function with python, 2020. <https://machinelearningmastery.com/softmax-activation-function-with-python/> [Accessed:24/02/2024].
- [41] Interactive Chaos. One hot encoding, 2024. <https://interactivechaos.com/es/manual/tutorial-de-machine-learning/one-hot-encoding> [Accessed:22/08/2024].
- [42] Saurav Maheshkar. What is cross entropy loss? a tutorial with code, 2023. <https://wandb.ai/sauravmaheshkar/cross-entropy/reports/What-Is-Cross-Entropy-Loss-A-Tutorial-With-Code--VmlldzoxMDA5NTMx#:~:text=Cross%20entropy%20loss%20is%20a,close%20to%200%20as%20possible.> [Accessed:29/02/2024].
- [43] Descenso del gradiente, 2024. [https://es.wikipedia.org/wiki/Descenso\\_del\\_gradiente#:~:text=El%20descenso%20del%20gradiente%20o,en%20direcci%C3%B3n%20contraria%20al%20gradiente.](https://es.wikipedia.org/wiki/Descenso_del_gradiente#:~:text=El%20descenso%20del%20gradiente%20o,en%20direcci%C3%B3n%20contraria%20al%20gradiente.) [Accessed:26/02/2024].

- [44] Gradiente, 2024. <https://es.wikipedia.org/wiki/Gradiente> [Accessed:26/02/2024].
- [45] Robert Kwiatkowski. Gradient descent algorithm — a deep dive, 2021. [https://towardsdatascience.com/gradient-descent-algorithm-a-deep-dive-cf04e8115f21#:~:text=Gradient%20descent%20\(GD\)%20is%20an,e.g.%20in%20a%20linear%20regression\).](https://towardsdatascience.com/gradient-descent-algorithm-a-deep-dive-cf04e8115f21#:~:text=Gradient%20descent%20(GD)%20is%20an,e.g.%20in%20a%20linear%20regression).) [Accessed:26/02/2024].
- [46] Jason Brownlee. Weight initialization for deep learning neural networks, 2021. <https://machinelearningmastery.com/weight-initialization-for-deep-learning-neural-networks/> [Accessed:14/03/2024].
- [47] Sandeep Jain. Kaiming initialization in deep learning, 2023. <https://www.geeksforgeeks.org/kaiming-initialization-in-deep-learning/> [Accessed:14/03/2024].
- [48] Adrian Rosebrock. Understanding weight initialization for neural networks, 2021. <https://pyimagesearch.com/2021/05/06/understanding-weight-initialization-for-neural-networks/> [Accessed:14/03/2024].
- [49] Yahia Zakaria. Initial bias values for a neural network, 2017. <https://stackoverflow.com/questions/44883861/initial-bias-values-for-a-neural-network> [Accessed:14/03/2024].
- [50] Glen Meyerowitz. Bias initialization in a neural network, 2018. <https://medium.com/@glenmeyerowitz/bias-initialization-in-a-neural-network-2e5d26fed0f0> [Accessed:14/03/2024].
- [51] ml4a. How neural networks are trained, 2020. [https://ml4a.github.io/ml4a/how\\_neural\\_networks\\_are\\_trained/](https://ml4a.github.io/ml4a/how_neural_networks_are_trained/) [Accessed:27/02/2024].
- [52] Sebastian Raschka. How is stochastic gradient descent implemented in the context of machine learning and deep learning?, 2024. <https://sebastianraschka.com/faq/docs/sgd-methods.html> [Accessed:27/03/2024].
- [53] Wikipedia. Stochastic gradient descent, 2024. [https://en.wikipedia.org/wiki/Stochastic\\_gradient\\_descent](https://en.wikipedia.org/wiki/Stochastic_gradient_descent) [Accessed:22/03/2024].

- [54] Aishwarya V Srinivasan. Stochastic gradient descent — clearly explained !!, 2019. <https://towardsdatascience.com/stochastic-gradient-descent-clearly-explained-53d239905d31> [Accessed:22/03/2024].
- [55] Afshine Amidi y Shervine Amidi. Convolutional neural networks cheatsheet, 2018. <https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks> [Accessed:19/03/2024].
- [56] Stanford. Convolutional neural networks (cnns / convnets), 2020. <https://cs231n.github.io/convolutional-networks/> [Accessed:19/03/2024].
- [57] DeepAI. Understanding padding in machine learning, 2024. <https://deepai.org/machine-learning-glossary-and-terms/padding> [Accessed:23/08/2024].
- [58] savyakhosla. Cnn — introduction to padding, 2024. <https://www.geeksforgeeks.org/cnn-introduction-to-padding/> [Accessed:02/04/2024].
- [59] Avinash Kumar. Full padding, 2020. [https://www.researchgate.net/figure/Full-padding-and-same-padding-10\\_fig5\\_337287600](https://www.researchgate.net/figure/Full-padding-and-same-padding-10_fig5_337287600) [Accessed:23/08/2024].
- [60] Avinash Kumar, Sobhangi Sarkar, and Chittaranjan Pradhan. *Malaria Disease Detection Using CNN Technique with SGD, RMSprop and ADAM Optimizers*, pages 211–230. 01 2020.
- [61] Archana David. Backprop through max-pooling layers?, 2007. <https://datascience.stackexchange.com/questions/11699/backprop-through-max-pooling-layers> [Accessed:24/03/2024].
- [62] Muhammad Baqir. How to backpropagate through max-pooling layers, 2024. <https://www.educative.io/answers/how-to-backpropagate-through-max-pooling-layers> [Accessed:24/03/2024].
- [63] Piotr Skalski. Let's code convolutional neural network in plain numpy, 2020. <https://towardsdatascience.com/lets-code-convolutional-neural-network-in-plain-numpy-ce48e732f5d5> [Accessed:24/03/2024].
- [64] Arnab Chakraborty. What is openmp?, 2019. <https://www.tutorialspoint.com/what-is-openmp> [Accessed:24/08/2024].

- [65] Microsoft. Directivas de openmp, 2024. <https://learn.microsoft.com/es-es/cpp/parallel/openmp/reference/openmp-directives?view=msvc-170> [Accessed:24/08/2024].
- [66] NVIDIA. Cuda c++ programming guide, 2024. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html> [Accessed:24/08/2024].
- [67] NVIDIA. Nvidia cudnn, 2024. <https://docs.nvidia.com/deeplearning/cudnn/latest/developer/core-concepts.html> [Accessed:02/08/2024].
- [68] NVIDIA. Nvidia cudnn, 2024. <https://docs.nvidia.com/deeplearning/cudnn/latest/api/cudnn-cnn-library.html#cudnnconvolutionforward> [Accessed:03/08/2024].
- [69] NVIDIA. Nvidia cudnn, 2024. <https://docs.nvidia.com/deeplearning/cudnn/archives/cudnn-897/api/index.html#cudnnPoolingForward> [Accessed:03/08/2024].
- [70] NVIDIA. Nvidia cudnn, 2024. <https://docs.nvidia.com/deeplearning/cudnn/latest/api/cudnn-cnn-library.html#cudnnconvolutionbackwardfilter> [Accessed:03/08/2024].
- [71] NVIDIA. Nvidia cudnn, 2024. <https://docs.nvidia.com/deeplearning/cudnn/latest/api/cudnn-cnn-library.html#cudnnconvolutionbackwarddata> [Accessed:03/08/2024].
- [72] Vishakh Hegde and Sheema Usmani. Parallel and distributed deep learning. 2016.
- [73] Sunwoo Lee, Dipendra Jha, Ankit Agrawal, Alok Choudhary, and Weikeng Liao. Parallel deep convolutional neural network training by exploiting the overlapping of computation and communication. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, pages 183–192, 2017.
- [74] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *ArXiv*, abs/1404.5997, 2014.
- [75] Hide Inada. Calculate cnn backprop with padding and stride set to 2, 2024. [https://hideyukiinada.github.io/cnn\\_backprop\\_strides2.html](https://hideyukiinada.github.io/cnn_backprop_strides2.html) [Accessed:01/04/2024].
- [76] H. Howie Huang¶ Zhufan Wang†‡ Weimin Zheng†‡ †Department of Computer Science Xiaqing Li†‡§, Guangyan Zhang†‡§, Tsinghua University ‡Tsinghua National Laboratory for Information Science Technology, Technology §State Key Lab of Mathematical Engineering, China

- ¶Department of Electrical Advanced Computing, Wuxi, and George Washington University Computer Engineering. Performance analysis of gpu-based convolutional neural networks. *International Conference on Parallel Processing*, 2016.
- [77] Mary Thomas. Comp 605: Introduction to parallel computing lecture : Cuda matrix-matrix multiplication, 2017. <https://edoras.sdsu.edu/~mthomas/sp17.605/lectures/CUDA-Mat-Mat-Mult.pdf> [Accessed:10/05/2024].
  - [78] NVIDIA. Matrix multiplication background, user's guide — nvidia docs, 2023. <https://docs.nvidia.com/deeplearning/performance/pdf/Matrix-Multiplication-Background-User-Guide.pdf> [Accessed:10/05/2024].

