

Redes Neuronales:  
del Perceptrón al Transformer



MIGUEL ÁNGEL MORALES RAMÓN

Junio 2024

## Resumen

Este proyecto de fin de grado se centra en el estudio y la aplicación de diversas arquitecturas de redes neuronales: redes neuronales artificiales (ANN), redes neuronales convolucionales (CNN), redes neuronales recurrentes (RNN) y *transformers*. El trabajo cuenta con una fase de desarrollo teórico, donde se exploran los fundamentos matemáticos y arquitectónicos de cada tipo de red, y una fase práctica, en la que se implementan y evalúan modelos específicos utilizando conjuntos de datos estándar. A través de experimentos, se analizan el rendimiento, la eficacia y los desafíos asociados a cada arquitectura. Los resultados obtenidos proporcionan una visión comprensiva de las fortalezas y limitaciones de cada enfoque, ofreciendo directrices claras para su implementación en diferentes aplicaciones del aprendizaje automático y el procesamiento del lenguaje natural. Este estudio explora el conocimiento ya existente en el campo de las redes neuronales, proporcionando tanto una base teórica sólida como orientaciones prácticas detalladas.

**Palabras clave:** Aprendizaje automático, aprendizaje supervisado, aprendizaje no supervisado, aprendizaje profundo, clasificación, regresión, perceptrón, redes neuronales artificiales, redes neuronales convolucionales, redes neuronales recurrentes, transformers, procesamiento natural del lenguaje, grandes modelos de lenguaje, visión por computador, descenso del gradiente, retropropagación, inteligencia artificial.

## Abstract

The following end of degree thesis focuses on the study and application of various neural network architectures: artificial neural networks (ANN), convolutional neural networks (CNN), recurrent neural networks (RNN) and transformers. The work has a theoretical development phase, where the mathematical and architectural foundations of each type of network are explored, and a practical phase, in which specific models are implemented and evaluated using standard data sets. Through experiments, the performance, effectiveness, and challenges associated with each architecture are analyzed. The results obtained provide a comprehensive view of the strengths and limitations of each approach, offering clear guidelines for their implementation in different applications of machine learning and natural language processing. This study contributes significantly to knowledge and practice in the field of neural networks, providing both a solid theoretical foundation and detailed practical guidance.

**Keywords:** machine learning, supervised learning, unsupervised learning, deep learning, regression, classification, perceptron, artificial neural networks, convolutional neural networks, recurrent neural networks, transformers, natural language processing, large language models, computer vision, gradient descent, back-propagation, artificial intelligence.

# Índice general

<b>1. Introducción</b>	<b>4</b>
1.1. Objetivos . . . . .	4
1.2. Metodología . . . . .	5
1.3. Antecedentes . . . . .	6
1.3.1. Aprendizaje automático . . . . .	6
1.3.2. Librerías de Python . . . . .	7
1.3.3. <i>Hardware</i> . . . . .	7
1.4. Estructura de la memoria . . . . .	8
<b>2. Redes neuronales</b>	<b>9</b>
2.1. El perceptrón . . . . .	9
2.2. Redes neuronales . . . . .	11
2.3. Entrenamiento . . . . .	13
2.3.1. Back-propagation y el gradiente descendiente . . . . .	14
2.3.2. Overfitting y underfitting . . . . .	16
2.4. Reconocimiento de dígitos . . . . .	18
2.4.1. Dataset MNIST . . . . .	19
2.4.2. Arquitectura . . . . .	19
2.4.3. Entrenamiento . . . . .	19
2.4.4. Resultados . . . . .	20
<b>3. Redes neuronales convolucionales</b>	<b>21</b>
3.1. Introducción . . . . .	21
3.2. Capa convolucional . . . . .	22
3.3. Capa de agrupamiento . . . . .	26
3.4. Capa densa . . . . .	27
3.5. Reconocimiento de dígitos . . . . .	28
3.5.1. Dataset MNIST . . . . .	28
3.5.2. Arquitectura . . . . .	29
3.5.3. Modelo A . . . . .	29
3.5.4. Modelo B . . . . .	29
3.5.5. Entrenamiento . . . . .	29
3.5.6. Modelo A . . . . .	29
3.5.7. Modelo B . . . . .	30
3.5.8. Resultados del conjunto de prueba . . . . .	30
<b>4. Redes neuronales recurrentes</b>	<b>32</b>
4.1. Arquitectura . . . . .	32
4.2. Diseño de redes neuronales recurrentes . . . . .	33
4.3. Long Short-Term Memory (LSTM) . . . . .	36

4.4.	Gated Recurrent Unit (GRU) . . . . .	38
4.5.	Análisis de reviews de IMDb . . . . .	39
4.5.1.	Dataset . . . . .	39
4.5.2.	Arquitectura . . . . .	39
4.5.3.	Entrenamiento . . . . .	40
4.5.4.	Resultados y conclusiones . . . . .	42
<b>5.</b>	<b><i>Transformers: La clave del procesamiento del lenguaje natural</i></b>	<b>44</b>
5.1.	Arquitectura . . . . .	44
5.1.1.	Embeddings y codificación posicional . . . . .	45
5.1.2.	Producto escalar escalado y multi-head attention . . . . .	47
5.2.	<i>Encoder</i> . . . . .	50
5.3.	<i>Decoder</i> . . . . .	51
5.4.	Entrenamiento . . . . .	52
5.5.	Inferencia . . . . .	54
5.6.	Traductor de inglés a español . . . . .	55
5.6.1.	Dataset . . . . .	55
5.6.2.	Modelos . . . . .	56
<b>6.</b>	<b>Conclusiones</b>	<b>64</b>

# Capítulo 1

## Introducción

En la era digital actual, el vasto volumen de datos generado cada día ha desencadenado una revolución en la forma en que abordamos la resolución de problemas complejos. En este contexto, el **Machine Learning**, o **Aprendizaje Automático**, emerge como una disciplina crucial que ha transformado radicalmente la capacidad de las máquinas para aprender patrones a partir de datos, evolucionando y mejorando sus propias capacidades sin intervención humana directa.

En su esencia, el *Machine Learning* es un campo de la inteligencia artificial que se centra en el desarrollo de algoritmos y modelos capaces de aprender y tomar decisiones basadas en datos. En lugar de depender de programación explícita, las máquinas pueden aprender automáticamente a través de la exposición a conjuntos de datos, adaptándose y mejorando su rendimiento a medida que reciben nuevos datos. De esta forma, se pretende prescindir de técnicas heurísticas para producir algoritmos y que sea la propia máquina la que aprenda a partir de un conjunto de datos con los resultados objetivo.

Este trabajo de fin de grado se propone explorar las bases teóricas y prácticas del *Machine Learning*, más concretamente, del *Deep Learning* aplicado a los procesadores naturales del lenguaje (NLP). Abarcaremos los distintos tipos de redes neuronales, centrándonos en las redes neuronales convolucionales (CNN) y su papel crucial en el procesamiento de imágenes (*computer vision*), textos y audios.

### 1.1. Objetivos

El objetivo de este trabajo es realizar, con ayuda de las matemáticas, un desarrollo teórico de las distintas redes neuronales que existen, empezando por las estructuras más simples y terminando por los últimos avances en la materia. Explicaremos su estructura, funcionamiento, y los problemas específicos que cada una de estas arquitecturas está diseñada para resolver. A través de un análisis teórico y práctico, este trabajo busca desmitificar el funcionamiento interno de estas redes y destacar las ventajas y limitaciones inherentes a cada tipo de red neuronal. Entonces, podemos distinguir los siguientes objetivos:

1. Desarrollo teórico de las redes neuronales:

- Realizar un desarrollo teórico de las distintas redes neuronales existentes.
- Explicar la estructura y funcionamiento de cada tipo de red neuronal.
- Abordar los problemas específicos que cada arquitectura está diseñada para resolver.

2. Análisis práctico:

- Explorar el funcionamiento interno de las redes neuronales.

- Proporcionar una comprensión profunda tanto desde una perspectiva matemática como práctica.
- Destacar las ventajas y limitaciones inherentes a cada tipo de red neuronal.

3. Evaluación y comparación de rendimiento:

- Evaluar el rendimiento de diferentes arquitecturas de redes neuronales mediante experimentos controlados.
- Utilizar conjuntos de datos estándar y aplicar métricas de evaluación pertinentes para cada tipo de tarea.
- Identificar las condiciones bajo las cuales cada tipo de red neuronal sobresale.
- Proporcionar recomendaciones sobre cuándo y cómo utilizar cada tipo de red neuronal.

4. Abordaje de desafíos comunes:

- Identificar y proponer soluciones para el sobreajuste en modelos de redes neuronales.
- Evaluar la eficiencia computacional de diferentes arquitecturas.
- Analizar la escalabilidad de los modelos en contextos prácticos.

Finalmente, el proyecto tiene como objetivo evaluar y comparar el rendimiento de estas diferentes arquitecturas mediante experimentos controlados. Realizamos experimentos con conjuntos de datos estándar, aplicando métricas de evaluación pertinentes para cada tipo de tarea. Estos experimentos permitirán identificar las condiciones bajo las cuales cada tipo de red neuronal sobresale y proporcionar recomendaciones sobre cuándo y cómo utilizar cada una. Además, abordaremos los desafíos comunes como el sobreajuste, la eficiencia computacional y la escalabilidad, proporcionando soluciones y estrategias para superarlos. En conjunto, este trabajo de fin de grado aspira a ser una contribución significativa al conocimiento y la práctica en el campo de las redes neuronales, ofreciendo tanto una base teórica sólida como orientaciones prácticas detalladas.

## 1.2. Metodología

La metodología del proyecto se ha estructurado en dos fases principales en cada capítulo: una fase de desarrollo teórico y una fase de pruebas de distintos modelos. Este enfoque secuencial permite una comprensión profunda y bien fundamentada de las redes neuronales antes de su implementación y evaluación práctica.

En la fase de desarrollo teórico, se llevó a cabo una revisión de la literatura existente sobre redes neuronales artificiales (ANN), redes neuronales convolucionales (CNN), redes neuronales recurrentes (RNN) y *transformers*. Esta revisión incluyó la recopilación y el estudio de artículos académicos, libros de texto y recursos en línea, con el objetivo de entender la evolución histórica, los fundamentos matemáticos, la arquitectura y los algoritmos de entrenamiento de cada tipo de red neuronal. Se elaboraron resúmenes detallados y diagramas explicativos para ilustrar la estructura y el funcionamiento de estas redes, junto con una comparación de sus ventajas y desventajas en diversos contextos. Esta fase teórica sentó las bases para la implementación práctica, proporcionando un marco conceptual sólido y una guía clara para la construcción y el entrenamiento de los modelos.

La segunda fase, centrada en las pruebas de distintos modelos, implicó la implementación práctica de las arquitecturas estudiadas utilizando herramientas y *frameworks* de aprendizaje profundo en *Python* como *scikit-learn* y *PyTorch*. Se seleccionaron varios conjuntos de datos estándar adecuados para evaluar cada tipo de red: imágenes para las CNN, secuencias temporales y de texto para las RNN, y tareas de procesamiento de lenguaje natural para los transformers. Los modelos se entrenaron y evaluaron para asegurar resultados fiables y reproducibles. Se registraron y

analizaron diversas métricas de rendimiento, como la precisión, la pérdida, el tiempo de entrenamiento y la capacidad de generalización. Además, se experimentó con diferentes configuraciones de hiperparámetros y técnicas de regularización para optimizar el rendimiento de los modelos. Esta fase práctica no solo permitió evaluar la eficacia de cada tipo de red en distintas tareas, sino que también ofreció insights valiosos sobre los desafíos y mejores prácticas en la implementación de redes neuronales.

## 1.3. Antecedentes

### 1.3.1. Aprendizaje automático

#### Aprendizaje supervisado

El **aprendizaje supervisado** [4] se basa en aprender de un conjunto de datos con etiquetas para cada uno de los ejemplos. El objetivo es conseguir un *mappeo* de *input* a *output*. Hay dos tipos de aprendizaje supervisado: clasificación y regresión.

- La **clasificación** [4] asigna una entrada a un conjunto fijo de categorías, por ejemplo, clasificando una imagen como un gato o un perro.
- Los problemas de **regresión** [4], por otro lado, asignan a un input un valor de número real. Un ejemplo de esto es intentar predecir los goles esperados de un equipo de fútbol o el precio de la bolsa de valores.

#### Aprendizaje no supervisado

El **aprendizaje no supervisado** [13] determina categorías a partir de datos donde no hay etiquetas presentes. Estos algoritmos pueden ser de dos tipos:

- **Agrupamiento** (Clustering) [13]: Agrupa datos similares sin etiquetas predefinidas. Ejemplos incluyen la segmentación de clientes o la clasificación de noticias por temas.
- **Reducción de Dimensionalidad** [13]: Reduce la cantidad de variables en un conjunto de datos manteniendo la mayor cantidad posible de información relevante.

#### Aprendizaje por refuerzo

El **aprendizaje por refuerzo** [15] se centra en maximizar una recompensa dada una acción o un conjunto de acciones realizadas. Los algoritmos están entrenados para fomentar ciertos comportamientos y desalentar otros. El aprendizaje por refuerzo tiende a funcionar bien en juegos como el ajedrez o el go, donde la recompensa puede ser ganar el juego. En este caso, se deben realizar una serie de acciones antes de alcanzar la recompensa.

#### Deep Learning

El **Deep Learning** [15] es una disciplina del aprendizaje automático que se centra en redes neuronales con múltiples capas (redes neuronales profundas). Los algoritmos de aprendizaje profundo intentan simular la arquitectura del cerebro humano, con capas de nodos (neuronas) interconectados que procesan y transforman los datos de entrada. El término profundo se refiere a la profundidad de la red neuronal, lo que significa que tiene múltiples capas.

La característica distintiva del aprendizaje profundo es su capacidad de aprender automáticamente representaciones jerárquicas de datos. Las redes neuronales profundas destacan en el aprendizaje de características, donde el modelo puede descubrir automáticamente características

relevantes a partir de datos sin procesar, eliminando la necesidad de ingeniería de características manual.

En resumen, el aprendizaje profundo es un subconjunto del aprendizaje automático que se ocupa específicamente de redes neuronales que contienen múltiples capas. Si bien todo aprendizaje profundo es aprendizaje automático, no todo aprendizaje automático es aprendizaje profundo. El aprendizaje automático incluye una gama más amplia de algoritmos y métodos más allá de las redes neuronales y las arquitecturas de aprendizaje profundo.

### 1.3.2. Librerías de Python

La principales librerías de python, utilizadas para el estudio de las distintas redes neuronales, han sido:

- *Pytorch*. Es una librería desarrollada por *Facebook's AI Research Lab* (FAIR), utilizada principalmente para la creación y entrenamiento de modelos de *deep learning*. Las principales razones por las que la he utilizado son:
  - Permite construir redes neuronales de manera rápida y eficiente..
  - Aprovecha el poder de las GPUs para acelerar el entrenamiento de modelos.
  - Facilita la diferenciación automática, lo que es esencial para el entrenamiento de redes neuronales.
  - Ideal para investigación y desarrollo rápido de prototipos debido a su diseño intuitivo y flexible.
- *Scikit-learn*. Es una librería de aprendizaje automático que proporciona herramientas simples y eficientes para el análisis de datos y la modelización predictiva. Sus principales características son:
  - Incluye una amplia variedad de algoritmos de aprendizaje supervisado y no supervisado, como regresión, clasificación, clustering, y reducción de dimensionalidad.
  - Diseñada para ser fácil de usar e integrar con otras librerías de Python como Numpy y Pandas.
  - Ofrece diversas técnicas para preprocesar los datos, como normalización, escalado, y extracción de características.
  - Proporciona herramientas para la validación cruzada, métricas de evaluación y selección de modelos.
- *Datasets*. Librería desarrollada por *Hugging Face* para descargar y trabajar con los *datasets* disponibles en su plataforma.
- *Sacrebleu*. Permite hacer uso de la métrica BLEU, utilizada para evaluar traductores de idiomas y predictores de texto, lo cual será de utilidad en el capítulo 5.
- *Numpy*. Es una librería fundamental para la computación numérica en Python. Proporciona soporte para matrices (*arrays* multidimensionales) y operaciones matemáticas de alto nivel.
- *Pandas*. Es una librería de análisis de datos y manipulación de datos en Python. Es especialmente útil para trabajar con datos estructurados (tabulares).

### 1.3.3. Hardware

- Hasta el capítulo 4, inclusive, el *hardware* utilizado ha sido un ordenador *MacBook* 13-inch, 2017, 2,3 GHz Dual-Core Intel Core i5.
- Para el capítulo 5, he alquilado una GPU *NVIDIA A100* durante 44 horas o el equivalente a 500 *computing units* en *Google Colab*. El coste total ha sido de 55,95€.

## 1.4. Estructura de la memoria

La memoria del proyecto sigue la siguiente estructura:

- Capítulo 2. Aborda los fundamentos y aplicaciones de las redes neuronales. Comienza con una introducción al perceptrón como unidad básica de las redes neuronales (2.1), seguido por una descripción general de las redes neuronales y su estructura (2.2). Se detalla el proceso de entrenamiento, enfocándose en el algoritmo de back-propagation y el método del gradiente descendiente (2.3.1), y se discuten los problemas de overfitting y underfitting (2.3.2). A continuación, se presenta un caso práctico de reconocimiento de dígitos utilizando el dataset MNIST (2.4), describiendo la arquitectura del modelo (2.4.2), su proceso de entrenamiento (2.4.3) y los resultados obtenidos (2.4.4).
- Capítulo 3. Explora las redes neuronales convolucionales (CNN) y sus componentes esenciales. Comienza con una introducción general a las CNN y su relevancia (3.1), seguida por una explicación detallada de la capa convolucional (3.2), la capa de agrupamiento o pooling (3.3), y la capa densa o fully connected (3.4). Luego, se presenta un caso práctico de reconocimiento de dígitos con el dataset MNIST (3.5), incluyendo la descripción de la arquitectura del modelo (3.5.2), el proceso de entrenamiento (3.5.5), y los resultados obtenidos en el conjunto de prueba (3.5.8).
- Capítulo 4. Se centra en las redes neuronales recurrentes (RNN) y su aplicación en el análisis de secuencias temporales. Comienza con una explicación de la arquitectura básica de las RNN (4.1) y el diseño de estas redes (4.2). Se profundiza en dos variantes importantes: las LSTM (Long Short-Term Memory) (4.3) y las GRU (Gated Recurrent Units) (4.4), que son mejoras para manejar dependencias a largo plazo. Finalmente, se presenta un caso práctico de análisis de reviews de IMDb (4.5), detallando el dataset utilizado (4.5.1), la arquitectura del modelo (4.5.2), el proceso de entrenamiento (4.5.3) y los resultados obtenidos (4.5.4).
- Capítulo 5. Este capítulo aborda los transformers y su impacto revolucionario en el procesamiento del lenguaje natural (NLP). Comienza con una descripción detallada de la arquitectura de los transformers (5.1), incluyendo los embeddings y la codificación posicional (5.1.1) y el mecanismo de atención multi-head y el producto escalar escalado (5.1.2). Luego, se explican en profundidad los componentes clave del modelo: el encoder (5.2) y el decoder (5.3). Se detallan los métodos de entrenamiento utilizados para ajustar el modelo (5.4) y se describe el proceso de inferencia para hacer predicciones con transformers entrenados (5.5). Por último, se construye un traductor de inglés a español haciendo uso de esta arquitectura (5.6).
- Capítulo 6. Se sintetizan los hallazgos más importantes del trabajo, destacando las principales contribuciones y aprendizajes obtenidos sobre las diversas arquitecturas de redes neuronales estudiadas.

Además, todo el código utilizado para el desarrollo del proyecto se encuentra en el repositorio de github.<sup>1</sup>

---

<sup>1</sup>Github del proyecto: <https://github.com/miguelangelmoralesramon/tfg.git>

## Capítulo 2

# Redes neuronales

Este apartado comienza con una exploración del perceptrón, la unidad básica de las redes neuronales, abordando sus principios fundamentales y su importancia histórica (2.1). Seguidamente, se ofrece una descripción general de las redes neuronales y su estructura, destacando cómo la interconexión de múltiples perceptrones permite la realización de tareas complejas (2.2). El proceso de entrenamiento de estas redes se analiza en detalle, enfocándose en el algoritmo de back-propagation y el método del gradiente descendente, herramientas cruciales para la optimización de los modelos (2.3.1). También se discuten los desafíos del overfitting y underfitting, problemas comunes que pueden comprometer la capacidad de generalización de los modelos (2.3.2). Para ilustrar estos conceptos, se presenta un caso práctico de reconocimiento de dígitos utilizando el dataset MNIST, describiendo la arquitectura del modelo utilizado (2.4.2), su proceso de entrenamiento (2.4.3) y los resultados obtenidos (2.4.4). Este enfoque proporciona una base sólida para comprender cómo las redes neuronales se pueden aplicar de manera efectiva en diversas tareas de reconocimiento y clasificación.

### 2.1. El perceptrón

Una neurona biológica es una célula especializada del sistema nervioso cuya función principal es recibir, procesar y transmitir información a través de señales químicas y eléctricas. Las neuronas están compuestas por tres partes principales [3]:

- Cuerpo celular. Contiene el núcleo de la célula y es el centro de control de la neurona. También contiene orgánulos que son esenciales para el funcionamiento de la célula.
- Dendritas. Son extensiones cortas y ramificadas que salen del cuerpo celular. Su función es recibir señales de otras neuronas y transmitirlas hacia el cuerpo celular.
- Axón. Es una extensión larga que conduce los impulsos eléctricos (potenciales de acción) desde el cuerpo celular hacia otras neuronas o tejidos. Los axones pueden estar cubiertos por una sustancia llamada mielina, que ayuda a acelerar la transmisión de señales.

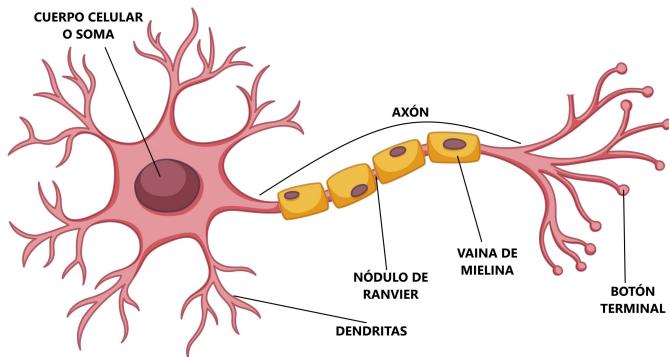


Figura 2.1: Neurona biológica

La comunicación entre neuronas se realiza en las sinapsis, que son pequeñas brechas entre el extremo de un axón de una neurona y las dendritas o el cuerpo celular de otra. Las señales eléctricas que llegan al final del axón provocan la liberación de neurotransmisores, que cruzan la sinapsis y se unen a los receptores en la neurona receptora, generando una nueva señal eléctrica.

Inspirado por esta idea, nace el concepto del perceptrón [2], que intenta simular el comportamiento una neurona del cerebro humano. En el perceptrón, cada dato de entrada (dendritas) representa una característica del experimento y está asociada con un peso que determina la importancia de esa característica. Las entradas se multiplican por sus respectivos pesos y se suman, dando como resultado el valor de la neurona (cuerpo celular). Luego, este cálculo pasa a través de una función de activación que decide si el perceptrón se activa o no, produciendo una salida (axón). Esto es análogo a cómo una neurona biológica suma las señales entrantes y, si superan un cierto umbral, dispara un potencial de acción. En la figura 2.2 se muestra un esquema de la arquitectura del perceptrón.

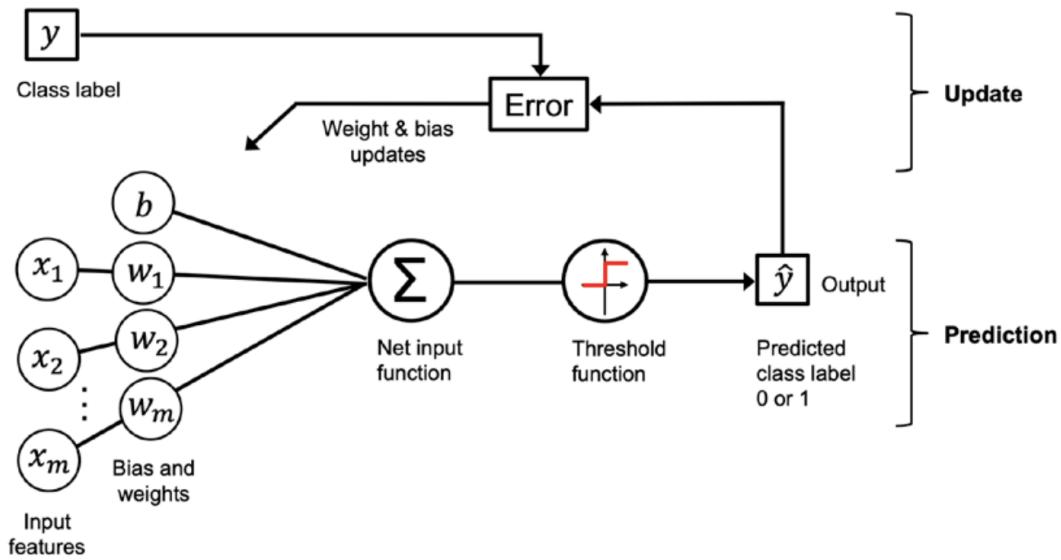


Figura 2.2: Perceptrón

## 2.2. Redes neuronales

Las **redes neuronales artificiales** (ANN) [2] o *multi-layer perceptron* (MLP) son una evolución del algoritmo del perceptrón. De la misma forma que el cerebro está compuesto por una red de neuronas biológicas, una ANN intenta imitar esta arquitectura, a través de un conjunto de perceptrones que están conectados entre sí y se transmiten información entre ellos.

Continuando con los conceptos mencionados en la sección anterior, podemos disponer  $n$  perceptrones en paralelo, es decir, sin que estén conectados entre sí. Construimos así una capa neuronal, que nos es más que un conjunto de neuronas  $a := \{a_i\}_{i=1}^n$  que no intercambian información entre sí, pero comparten conexiones con otras neuronas.

Si conectamos varias capas neuronales entre sí, obtenemos una red neuronal artificial, que podemos definir como un sistema  $S = \{a_i^j, \{z_i^j, w_i^j, b^j\}, f\}_{i,j}$ , cuyos componentes son:

- El índice  $j$  determina la capa de la red neuronal, por lo que  $1 \leq j \leq L$ , donde  $L$  es el número total de capas de la red. El índice  $i$  indica la posición dentro de cada capa, es decir, si la capa  $j$  tiene  $n_j$  neuronas entonces  $1 \leq i \leq n_j$ .
- Elementos  $a_i^j \in \mathbb{R}$  llamados **neuronas**, son la unidad básica de la red. La neurona  $a_i^j$  corresponde a la posición  $i$  en la capa  $j$  de la red.
- Denotamos a la capa neuronal  $j$  como  $a^j := \{a_1^j, a_2^j, \dots, a_{n_j}^j\} = \{a_i^j\}_{i=1}^{n_j}$ , donde  $n_j$  es el número total de neuronas de la capa  $j$
- Las interacciones  $f$ , de tipo **función de pérdida** ( $f_l$ ) y **funciones de activación** ( $f_a$ ). Un requerimiento esencial es que ambas sean derivables ya que al optimizar los parámetros de la red, necesitaremos calcular sus derivadas parciales.
- Tuplas de estados  $\{z_i^j, w_i^j, \}$ , donde  $z_i^j$  son los **valores de activación**,  $w_i^j$  los **pesos** de la red. Los valores de activación son el resultado de aplicar la función de activación a cada neurona. Los pesos determinan el valor de las neuronas de cada capa, en función de los valores de entrada, que proceden de la capa anterior. Los pesos de la red son vectores que determinan los cálculos entre y capa y capa. El valor de las neuronas de cada capa, es el resultado de computar los pesos con los valores de entrada, que proceden de la capa anterior. De esta forma, el vector de pesos  $w_i^j$  sería:

$w_i^j = (w_{i,1}^j, \dots, w_{i,k}^j, \dots, w_{i,n_{j-1}}^j)$ ,  $1 \leq j \leq L$ ,  $1 \leq i \leq n_j$  donde  $L$  es el número de capas de la red y  $n_{j-1}$ ,  $n_j$  es el número de neuronas en la capa  $j-1$  y  $j$ , respectivamente.

Es decir, el peso  $w_{i,k}^j$  representa la conexión entre la neurona  $a_k^{j-1}$  (perteneciente a la capa  $j-1$ ) y la neurona  $a_i^j$ .

Podemos representar esta construcción usando la teoría de grafos como se muestra en la figura 2.3.

Supongamos que tenemos para un *input* de dimensión dos, una red neuronal como la de la figura 2.3, con 3 capas, de tamaño 4 la primera, 3 la segunda y la última que tiene una única neurona. El funcionamiento básico de la red será de la siguiente forma:

1. Dado un *input*  $x = \{x_1, x_2\}$ , este se identificará con la capa 0 de la red, es decir,  $a^0 := x$ .
2. Los valores de la primera capa se obtendrán haciendo los siguientes cálculos:  $a_i^1 = f_a(z_i^1)$ , donde  $z_i^1 = w_i^1 \cdot a^0 + b^1$  y  $f_a$  es la función de activación. El resultado de la primera capa será:  $a^1 = (a_1^1, a_2^1, a_3^1, a_4^1)$
3. Repetimos el proceso con la segunda capa:  $a_i^2 = f_a(w_i^2 \cdot a^1 + b^2)$
4. Al llegar a la capa final  $a^3 := f_a(\sum_{i=1}^3 w_i^3 \cdot a_i^2 + b^3)$ , que será el *output* de nuestra red.

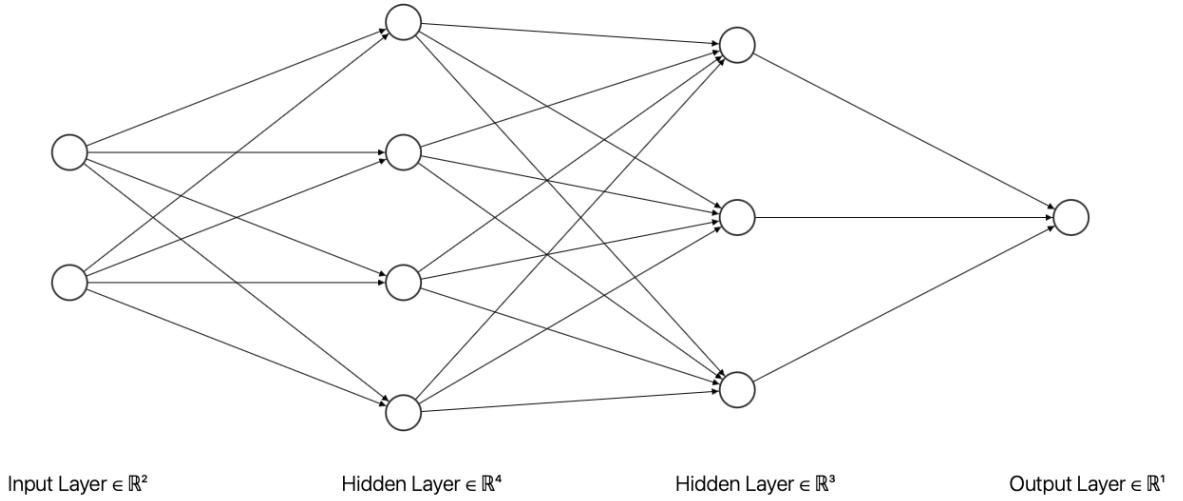


Figura 2.3: Red neuronal de 3 capas

Como podemos observar, nos resultará más cómodo representar los pesos de la capa  $j$  en forma de matriz de tal forma que cada fila  $i$  corresponda a un vector  $w_i^j : W^j = (w_1^j, \dots, w_N^j)^\top$ . Así, podremos escribir los cálculos de una forma más sencilla:  $a^j = f_a(W^j \cdot a^{j-1} + b^j)$

En cuanto a las interacciones,  $f$ , sus componentes están muy relacionadas entre sí, ya que la elección de la función de activación condicionará la función de pérdida que utilizemos. Para la parte práctica de este trabajo, el único requisito indispensable que exigiremos será:  $f_l, f_a \in C^\infty$ . A continuación, se muestran las funciones de activación y pérdida más utilizadas:

■ Funciones de activación:

1. Step. Es utilizada principalmente en perceptrones y redes neuronales muy simples. Es una función binaria que devuelve 0 si la entrada es menor que un umbral predefinido y 1 si es mayor o igual a ese umbral. Es fácil de implementar y comprender, pero no es diferenciable en todos los puntos, lo cual dificulta su aplicación en técnicas de optimización basadas en gradientes, como el algoritmo de *backpropagation*:

$$f(z) = \begin{cases} 0 & z < 0,5 \\ 1 & 0,5 \leq z \end{cases}$$

2. Sigmoid. Transforma cualquier valor de entrada en un rango entre 0 y 1, siguiendo una curva en forma de "S". Es comúnmente utilizada en la capa de salida de modelos de clasificación binaria, donde la salida puede interpretarse directamente como una probabilidad. Sin embargo, puede enfrentar el problema del gradiente desvanecido para valores extremos de entrada, lo que puede complicar el entrenamiento de redes neuronales profundas:

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

3. Tangente hiperbólica. Similar a la sigmoid, la función *tanh* transforma cualquier valor de entrada en un rango entre -1 y 1. Es preferible utilizarla en redes neuronales profundas

por su salida centrada en torno a cero, lo que puede ayudar en el entrenamiento al proporcionar gradientes más fuertes comparado con la sigmoide. Sin embargo, también puede sufrir del problema del gradiente desvanecido para valores extremos de entrada, aunque en menor medida que la sigmoid:

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

4. ReLu. Es ampliamente utilizada en las capas ocultas de redes neuronales profundas debido a su simplicidad y eficiencia computacional. Esta función devuelve el valor de entrada si es positivo y 0 si es negativo, lo que ayuda a mitigar el problema del gradiente desvanecido al mantener activas las neuronas durante el entrenamiento. Sin embargo, puede provocar el problema de "neuronas moribundas" donde algunas neuronas pueden quedar inactivas si reciben siempre entradas negativas:

$$R(z) = \max(0, z)$$

5. Softmax. Es utilizada comúnmente en la capa de salida de modelos de clasificación multiclase. Transforma un vector de valores en un vector de probabilidades, donde cada valor es transformado de manera que la suma de todas las probabilidades es igual a 1. Esto permite interpretar la salida como una distribución de probabilidad sobre las diferentes clases. Sin embargo, puede ser computacionalmente intensiva y sensible a valores atípicos en los datos de entrada, lo que requiere cuidado en su aplicación práctica:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{i=0}^n e^{x_i}}$$

- Funciones de pérdida, donde  $y$  es la variable objetivo y  $\hat{y}$  la predicción del modelo:

1. MSE. Mide la media de los cuadrados de los errores, es decir, las diferencias entre los valores reales y los valores predichos. Se utiliza comúnmente en problemas de regresión, donde se predice un valor continuo.

$$MSE(y, \hat{y}) = \frac{1}{n} \sum_0^n (y_i - \hat{y}_i)^2$$

2. *Binary Cross Entropy Loss*. Mide la disimilitud entre las distribuciones de las etiquetas reales y las predichas, penalizando más fuertemente las predicciones incorrectas cuando el modelo está muy seguro de su predicción (es decir, cuando  $\hat{y}_i$  es cercano a 0 o 1). Se utiliza en problemas de clasificación binaria, donde la tarea es predecir una de dos clases:

$$L(y, \hat{y}) = -\frac{1}{n} \sum_0^n y_i \log \hat{y}_i + (1 - y_i) \log (1 - \hat{y}_i)$$

3. *Cross Entropy Loss*. Es similar a la *binary cross-entropy loss*, pero extendida a múltiples clases:

$$L(y, \hat{y}) = - \sum y_i \log \hat{y}_i$$

### 2.3. Entrenamiento

Este algoritmo requiere de un conjunto de entrenamiento o dataset, D:

$D = \{(x_i, y_i) \mid x_i \in X, y_i \in Y\}$ , donde X es el conjunto de datos de entrada e Y el conjunto de etiquetas, que le corresponde a cada data de entrada. Este conjunto de datos lo dividiremos en tres partes o *splits*:

- Conjunto de entrenamiento, que utilizaremos para entrenar al modelo y ajustar los parámetros.
- Conjunto de validación, que utilizaremos para ajustar los hiperparámetros, es decir, para determinar la tasa de aprendizaje, el tamaño del lote y la arquitectura de la red.
- Conjunto de test, que servirá para proporcionar una evaluación final del modelo.

Nuestro objetivo será encontrar los valores óptimos de los pesos de la red, para que produzca predicciones lo más cercanas posibles a la realidad. Para ello, determinaremos un el número de *epochs* ( $n\_epochs$ ), que indica el número de veces que iteraremos sobre el *dataset* completo, y tamaño de lote (*batch\_size*). En cada *epoch*, realizaremos  $\frac{\text{len}(\text{dataset})}{\text{batch\_size}}$  iteraciones en las que calcularemos las predicciones de nuestra red sobre un conjunto de datos de tamaño igual a *batch\_size*. Estas predicciones las mediremos contra los valores reales y computaremos el error en base a la función de pérdida  $f_l$ . A continuación, actualizaremos los pesos siguiendo el algoritmo de *back-propagation*.

### 2.3.1. Back-propagation y el gradiente descendiente

Cuando entrenamos una red neuronal, la información fluye de la capa de entrada a la capa de salida y produce un coste escalar, determinado por la función de pérdida  $f_l$ . Nuestro objetivo al entrenar la red es disminuir el valor de la función de pérdida, modificando los pesos de cada capa. Para ello, se aplica el algoritmo de descenso del gradiente (o *gradient descent*) a  $f_l$ , en función a la matriz de pesos  $W$ . Este algoritmo requiere que  $f_l$  sea diferenciable y convexa. Consta de los siguientes pasos:

1. Calcular el punto de salida. En este caso será el valor actual de la función  $f_l$  para el lote de entrenamiento correspondiente.
2. Establecer la **tasa de aprendizaje** o *learning rate*,  $\eta \in \mathbb{R}$ , que determina la velocidad a la que se ajustan los pesos del modelo. Una tasa de aprendizaje demasiado grande puede provocar que el algoritmo no converga y una tasa insignificante, hará que el algoritmo tarde demasiado tiempo en converger, por lo que la elección de esta es un aspecto crucial para obtener un entrenamiento eficiente.
3. Calcular el gradiente de la función:  $\nabla f_l(W)$ , donde  $W$  es la matriz de pesos de la red neuronal.
4. Actualizar los pesos de la red a  $W'$ , donde  $W' = W - \eta \nabla f_l(W)$

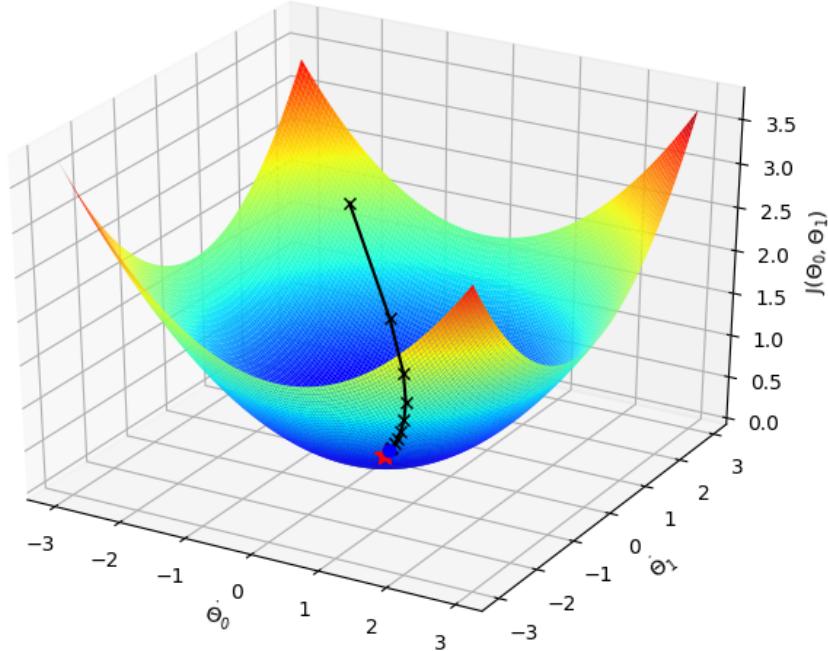


Figura 2.4: Resultado de aplicar el algoritmo de *gradient descent* a la superficie  $z = x^2 + y^2$ .

La desventaja de aplicar directamente este algoritmo es que calcular el gradiente de la función en redes neuronales, es muy costoso computacionalmente por la gran cantidad de pesos de los que dependen las derivadas parciales del gradiente. El algoritmo de *back-propagation* soluciona este problema y nos permite calcular el gradiente más eficientemente. Se basa en la idea de que valor de  $f_l$  fluya de la última capa hasta la primera para calcular el gradiente y actualizar los pesos de la red, de tal forma que el coste se reduzca y nuestro modelo sea más preciso.

Primero, debemos definir la función de coste general [5], que debe de ser la media de  $n = batch\_size$  errores de muestras individuales  $x_i$ :  $L(w) = \frac{1}{n} \sum_{i=1}^n f_l(x_i)$ . Al igual que en el algoritmo tradicional del descenso del gradiente, es necesario que  $f_l$  sea diferenciable y convexa.

El error,  $\delta_i^j$ , de la neurona  $i$  en la capa  $j$ :  $\delta_i^j := \frac{\partial L}{\partial z_i^j}$ . Identificaremos con  $\delta^j := (\delta_1^j, \dots, \delta_{d_l}^j)$ ,  $d_l = dim(a^l)$  el vector de errores asociado a la capa  $j$ . La retropropagación se basa en 4 ecuaciones fundamentales [5], que permiten calcular  $\delta^j$  y el gradiente de la función de coste:

1. Error en la capa de salida:  $\delta_i^N = \frac{\partial L}{\partial a_i^N} f'_a(z_i^N)$
2. Error de la capa  $j$ , en función de la capa  $j+1$ :  $\delta_i^j = \frac{\partial L}{\partial z_i^j} = \sum_k \frac{\partial L}{\partial z_k^{j+1}} \frac{\partial z_k^{j+1}}{\partial z_i^j} = \sum_k \frac{\partial z_k^{j+1}}{\partial z_i^j} \delta_k^{j+1}$ .  
Teniendo en cuenta que  $z_k^{j+1} = \sum_i w_{k,i}^{j+1} a_i^j + b^{j+1}$   
 $= \sum_i w_{k,i}^{j+1} f_a(z_i^j) + b^{j+1}$ , diferenciando obtenemos:  $\frac{\partial z_k^{j+1}}{\partial z_i^j} = w_{k,i}^{j+1} f'_a(z_i^j)$ . Finalmente sustituyendo, obtenemos la segunda ecuación fundamental:  $\delta_i^j = \sum_k w_{k,i}^{j+1} \delta_k^{j+1} f'_a(z_i^j)$
3. La derivada parcial del coste respecto de cualquier peso:  $\frac{\partial L}{\partial w_{i,k}^j} = a_k^{j-1} \delta_i^j$ .
4. La derivada parcial del coste respecto del sesgo:  $\frac{\partial L}{\partial b_i^j} = \delta_i^j$ .

Finalmente, debemos de determinar una tasa de aprendizaje,  $\eta$ , que como se ha indicado anteriormente, determinará el tamaño de los pasos al actualizar los pesos y sesgos en el algoritmo de descenso del gradiente. Con esto, ya tenemos todos los ingredientes necesarios para construir el algoritmo de retropropagación:

---

#### Algorithm 1 Algoritmo de *back-propagation* o retropropagación

---

```

Require:  $batch := [(x_1, y_1), \dots, (x_n, y_n)]$       ▷ Calculamos los errores para cada neurona y cada muestra
for  $x_i, -$  in  $batch$  do
     $a^{x_i,1} := a^1(x_i)$ 
    Feedforward :  $\forall l = 2, \dots, L$     $a^{x_i,l} := f_a(z^{x_i,l})$ 
    Calcular  $\delta^{x_i,L}$ 
    retropropagar el error :  $\forall j = N - 1, \dots, 2$    calcular    $\delta^{x_i,l}$ 
end for
 $l \leftarrow N$ 
while  $l \geq 0$  do          ▷ Aplicamos el algoritmo de descenso de gradiente para actualizar los parámetros
     $w^l \leftarrow w^l - \eta \frac{1}{n} \sum_{i=1}^n \delta^{x_i,l} (a^{x_i,l-1})^T$ 
     $b^l \leftarrow b^l - \eta \frac{1}{n} \sum_{i=1}^n \delta^{x_i,l}$ 
end while

```

---

#### Optimizadores

En la práctica, no realiza una implementación literal de estos algoritmos de optimización, sino que existen librerías de python de aprendizaje automático que cuentan con funciones listas para implementar en los modelos. Estas herramientas se denominan **optimizadores** y permiten ajustar

los pesos de los modelos durante la fase de entrenamiento de forma eficiente. En *pytorch*, que es la librería para diseñar y entrenar modelos que utilizaré principalmente, los optimizadores más utilizados son:

- Descenso del gradiente estocástico (SGD). Actualiza los parámetros en la dirección del gradiente negativo de la función de pérdida, siguiendo los algoritmos clásicos enunciados anteriormente. Se puede implementar en *pytorch* a través de la función `torch.optim.SGD`.
- *Adagrad* [12]. Es un algoritmo adaptativo, lo que significa que varía la tasa de aprendizaje inicial para optimizar el proceso de ajuste de pesos. *Adagrad* (en *pytorch*, `torch.optim.Adagrad`) adapta la tasa de aprendizaje en función de la frecuencia de actualizaciones de cada parámetro. Los parámetros con actualizaciones más frecuentes obtienen tasas de aprendizaje más pequeñas, mientras que aquellos con actualizaciones poco frecuentes obtienen tasas de aprendizaje mayores.
- *RMSProp* [12]. También adapta la tasa de aprendizaje en función de la frecuencia de las actualizaciones de parámetros, pero introduce una media decreciente de los gradientes cuadrados anteriores. Utiliza un factor de caída para controlar cuánto afectan los gradientes pasados al gradiente actual. Esto mantiene una tasa de aprendizaje más consistente, lo que resulta en un entrenamiento más estable. En *pytorch*, se puede utilizar con la función `torch.optim.RMSprop`.
- *Adam*. El optimizador *Adaptive Moment Estimation* [6] (en *pytorch*, `torch.optim.Adam`) es también un algoritmo adaptativo y combina las ventajas de *AdaGrad* y *RMSProp*. Ajusta los parámetros de la red neuronal durante el entrenamiento basándose en estimaciones adaptativas de momentos de primer y segundo orden, que son la media y la varianza de los gradientes. Este será el optimizador que utilizaré por lo general a lo largo del proyecto.

Existe una variación, llamada *AdamW*, que aplica de forma distinta la regularización  $L_2$ .

### 2.3.2. Overfitting y underfitting

Un problema común en machine learning es el sobre-ajuste (o *over-fitting*) e infra-ajuste (o *under-fitting*) del modelo a los datos. El primero tiene lugar cuando el modelo se ajusta demasiado al conjunto de entrenamiento, produciendo muy malos resultados en los tests y dando lugar a un modelo preciso pero inexacto. El segundo, ocurre cuando el modelo es demasiado simple para los datos con los que se está trabajando y, como resultado, tiene un rendimiento deficiente tanto en los datos de entrenamiento como en los datos del test. En la figura 2.5, se muestra el resultado de 3 modelos de clasificación (línea roja) sobre unos datos que tienen únicamente 2 clases. Se observa cómo el modelo de la izquierda tiene *under-fitting*, el de la derecha *over-fitting* y el del centro es un modelo preciso y exacto.

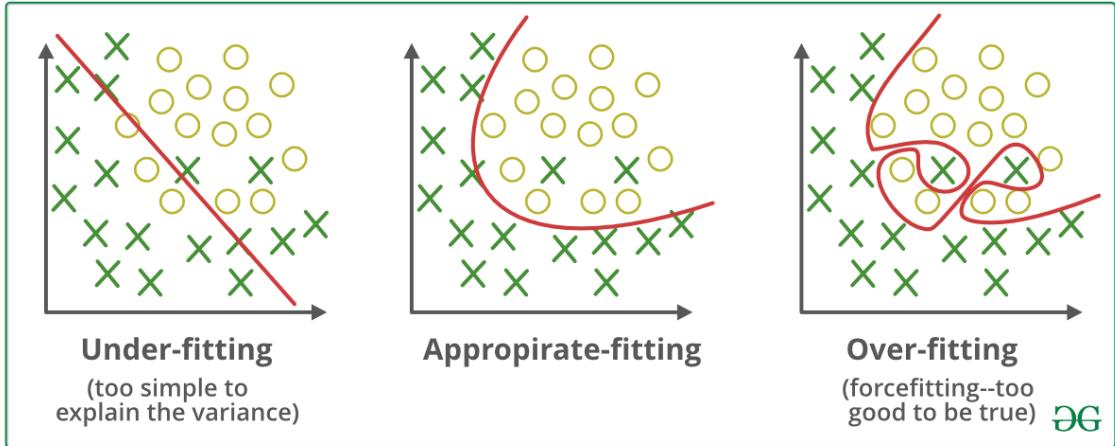


Figura 2.5: *Under-fitting* (izquierda), *fitting* correcto (centro) y *over-fitting* (derecha) en un problema de clasificación.

Para combatir estos problemas, las técnicas más utilizadas son:

- **Regularización:** se define como cualquier modificación que introducimos a un algoritmo de aprendizaje con la intención de reducir su error de generalización, es decir, de reducir su sobre-ajuste. Estas modificaciones se añaden en la función de pérdida,  $f_l$ , a través de un sumando adicional.

1. Tipo L1: modificamos la función de pérdida añadiendo el sumatorio de los valores absolutos de los pesos  $f_l^{reg} = f_l + \frac{\lambda}{n} \sum |w|$

$$\text{Derivando, obtenemos } \frac{\partial f_l^{reg}}{\partial w} = \frac{\partial f_l}{\partial w} + \frac{\lambda}{n} signo(w) \text{ y } \frac{\partial f_l^{reg}}{\partial b} = \frac{\partial f_l}{\partial b}$$

Entonces, introducir este parámetro no afecta a la modificación de los sesgos durante el entrenamiento, pero sí, a la actualización de los pesos en el algoritmo de retropropagación:

$$w^l \leftarrow (1 - \frac{\eta \lambda}{n} signo(w^l))w^l - \eta \frac{1}{n} \sum_{i=1}^n \delta^{x_i, l} (a^{x_i, l-1})^T$$

2. Tipo L2: añadimos el sumatorio de los pesos al cuadrado.

$$f_l^{reg} = f_l + \frac{\lambda}{2n} \sum w^2$$

$$\text{Derivando, obtenemos } \frac{\partial f_l^{reg}}{\partial w} = \frac{\partial f_l}{\partial w} + \frac{\lambda}{n} w \text{ y } \frac{\partial f_l^{reg}}{\partial b} = \frac{\partial f_l}{\partial b}$$

Por tanto, deberemos de añadir un coeficiente al actualizar los pesos en el algoritmo de *back-propagation*:

$$w^l \leftarrow (1 - \frac{\eta \lambda}{n})w^l - \eta \frac{1}{n} \sum_{i=1}^n \delta^{x_i, l} (a^{x_i, l-1})^T$$

- **Normalización por lotes:** además de ser una técnica de regularización eficaz, es un gran antídoto contra el desplazamiento de las covarianzas internas. Este problema se da cuando la distribución de probabilidad de los datos de entrada cambia al pasar de una capa a otra, lo que hace el entrenamiento inestable y lento. Para solucionarlo, se normalizan los valores de activación de cada capa. Supongamos que el resultado de aplicar una capa neuronal a un lote de  $b$  muestras con  $k$  características cada una es  $X \in Mat_{b \times k}$ . Es decir, la primera muestra sería  $x_1 = (X_{1,1}, \dots, X_{1,k})$  y la primera característica sería  $y_1 = (X_{1,1}, \dots, X_{b,1})$ . Obtenemos el lote normalizado  $\hat{X} = (\hat{y}_1, \dots, \hat{y}_k) \in Mat_{b \times k}$ , realizando los siguientes cálculos:

$$\hat{y}_i = \frac{y_i - \nu_i}{\sqrt{\sigma_i + \epsilon}}$$

donde  $\nu_i = mean(y_i)$ ,  $\sigma_i = std(y_i)$  y  $\epsilon$  es un valor mayor que cero despreciable, para evitar que el denominador sea nulo.

- **Normalización por capas:** consiste en normalizar las características de cada muestra del lote de forma independiente. Supongamos que el resultado de aplicar una capa neuronal a un lote de  $b$  muestras con  $k$  características cada una es  $X \in Mat_{b \times k}$ . La primera muestra sería  $x_1 = (X_{1,1}, \dots, X_{1,k})$  y la primera característica sería  $y_1 = (X_{1,1}, \dots, X_{b,1})$ . Obtenemos el lote normalizado  $\hat{X} = (\hat{x}_1, \dots, \hat{x}_b) \in Mat_{b \times k}$ , realizando los siguientes cálculos:

$$\hat{x}_i = \frac{x_i - \nu_i}{\sqrt{\sigma_i + \epsilon}}$$

donde  $\nu_i = mean(x_i)$ ,  $\sigma_i = std(x_i)$  y  $\epsilon$  es un valor mayor que cero despreciable, para evitar que el denominador sea nulo.

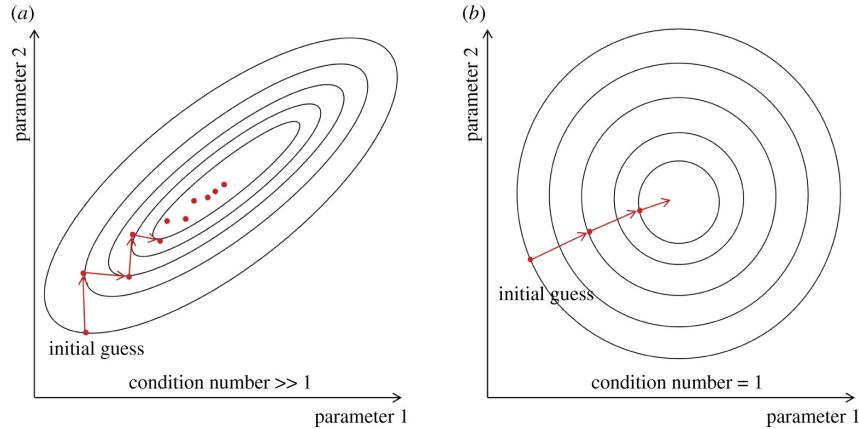


Figura 2.6: Descenso del gradiente sin aplicar normalización (a) vs con normalización (b)

- **Dropout:** A diferencia de las técnicas de regularización, en lugar de modificar la función de pérdida, el *dropout* se centra en modificar la estructura de la red neuronal. Esto lo haremos especificando una probabilidad  $p^l \in [0, 1]$  en cierta capa  $l$ , que indicará la probabilidad de que cada neurona de esa capa se desactive. Así, cuando entrenemos la red y propaguemos hacia adelante la información, ciertas neuronas de las capas en las que esté activo el *dropout* se apagarán en función de  $p^l$ . La idea detrás de esta técnica, es que cada vez que apagamos ciertas neuronas, nuestra red cambia, lo que hace que la red resultante sea la *media* de distintas redes y como consecuencia, un mejor modelo. Así, evitamos que se desarrollen patrones en nuestra red, que produzcan un sobre-ajuste.
- **Data Augmentation:** Entrenar un modelo con pocos datos produce ambos problemas. Las técnicas de data augmentation nos permiten realizar transformaciones en el *dataset* para, a partir de los datos existentes, producir nuevas muestras. Esta técnica es particularmente útil en imágenes, sobre las que podemos aplicar rotaciones, inversiones, reflexiones, interpolaciones y correcciones, resultando en una gran cantidad de nuevas muestras de calidad para nuestro modelo.

## 2.4. Reconocimiento de dígitos

Esta sección tiene como objetivo implementar y entrenar una red neuronal artificial para el reconocimiento de dígitos escritos a mano, utilizando el conjunto de datos MNIST.

El código utilizado se encuentra en el archivo `ann_MNIST.ipynb` del repositorio del proyecto<sup>1</sup>.

<sup>1</sup>Github del proyecto: <https://github.com/miguelangelmoralesramon/tfg.git>

#### 2.4.1. Dataset MNIST

El conjunto de datos MNIST (Modified National Institute of Standards and Technology) es uno de los conjuntos de datos más emblemáticos en el campo del aprendizaje automático. Sus características principales son:

- Contenido: MNIST contiene imágenes en blanco y negro de dígitos escritos a mano, del 0 al 9, como se muestra en la figura 2.7.
- Tamaño de las imágenes: Cada imagen tiene un tamaño de 28x28 píxeles, y cada píxel tiene un valor de intensidad (en escala de grises) que varía de 0 a 255. Para este caso, cargaremos las imágenes directamente como vectores de dimensión  $28 \times 28 = 784$ .
- Datos: se dividen en un conjunto de entrenamiento, compuestos por 60000 imágenes y un conjunto de prueba de 10000 imágenes.

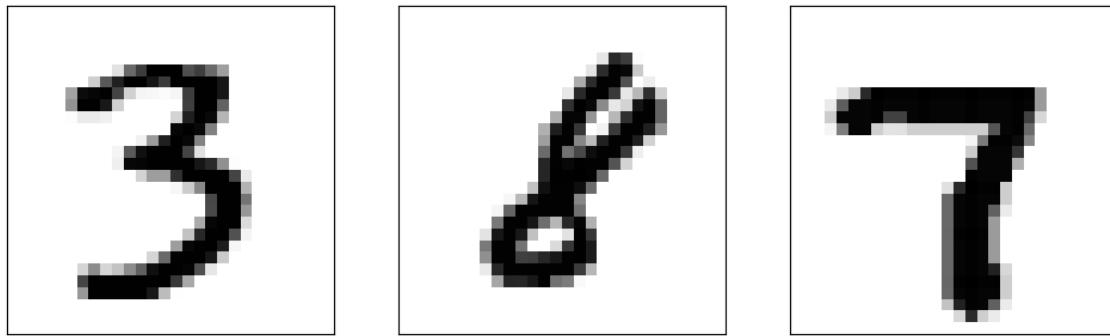


Figura 2.7: Primeras 3 muestras del conjunto de entrenamiento del dataset MNIST.

#### 2.4.2. Arquitectura

1. Capa 1: tamaño de entrada 784 y tamaño de salida 128, con función de activación ReLU.
2. Capa 2: tamaño de entrada 128 y tamaño de salida 64, con función de activación ReLU.
3. Capa 3: tamaño de entrada 64 y tamaño de salida 10, con función de activación softmax.

#### 2.4.3. Entrenamiento

Para el entrenamiento he dividido los conjuntos de entrenamiento y validación en *batch sizes* de 100 muestras y han sido necesarias 50 *epochs*. Para calcular la pérdida he utilizado la función de *Cross Entropy Loss* ya que es la más representativa en problemas de clasificación múltiple. He probado varias tasas de aprendizaje,  $\eta \in [10^{-4}, 10^{-1}]$ , obteniendo un mejor resultado con  $\eta \approx 10^{-3}$ . El tiempo de entrenamiento con esta arquitectura y parámetros ha sido de **160,77 segundos**. En la figura 2.8 se muestran los resultados de entrenamiento con estos parámetros.

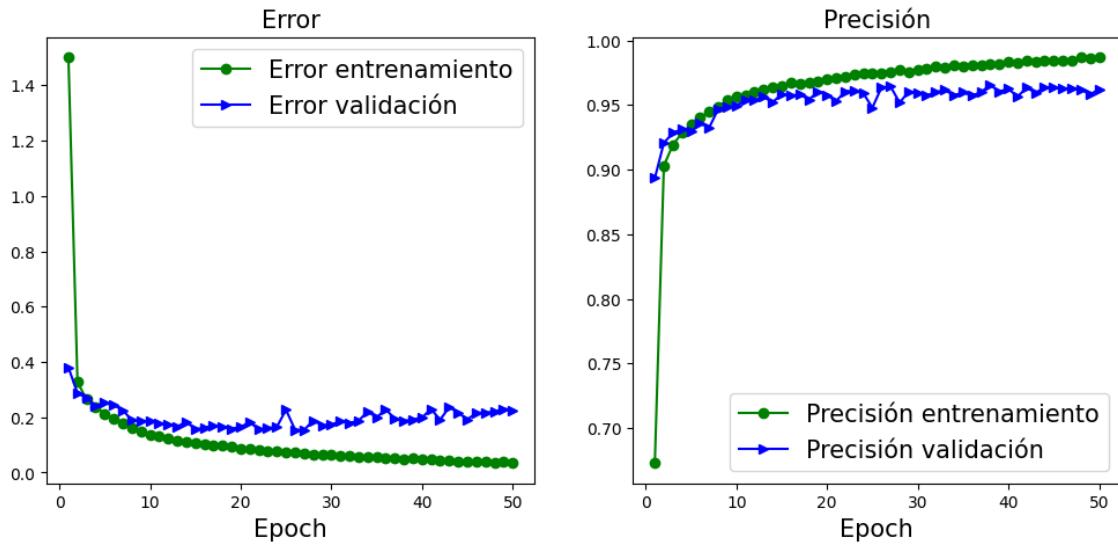


Figura 2.8: Resultados de entrenamiento con  $\eta = 0,001$ ,  $batch\_size = 100$  y  $n\_epochs = 50$ .

#### 2.4.4. Resultados

Finalmente, he medido la pérdida del modelo enfrentándolo al conjunto de prueba, obteniendo una precisión final del 95,73 %. En la figura 2.9 se muestran algunas predicciones del modelo sobre 12 muestras del conjunto de prueba.

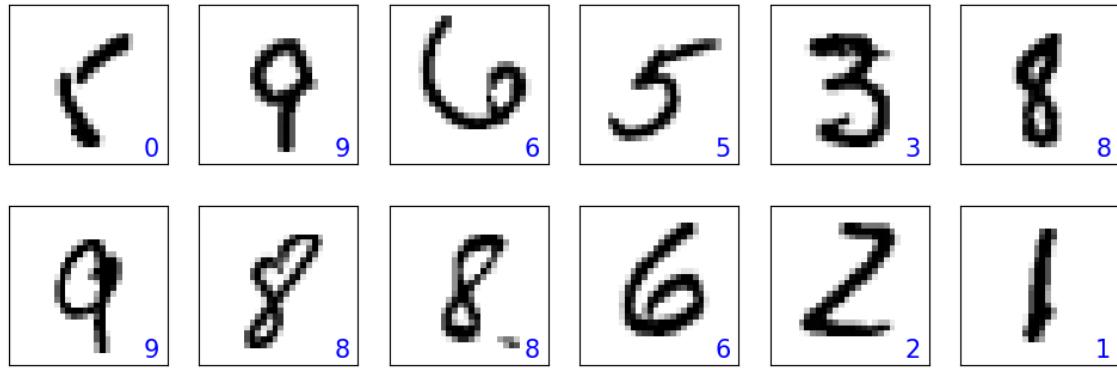


Figura 2.9: Resultados del modelo contra una muestra de 12 imágenes del conjunto de prueba. En azul se muestra la predicción del modelo.

# Capítulo 3

## Redes neuronales convolucionales

En este capítulo, se exploran las redes neuronales convolucionales (CNN), una arquitectura avanzada que ha revolucionado el campo del procesamiento de imágenes y el reconocimiento de patrones. Se inicia con una introducción general a las CNN y su importancia en la actualidad (3.1). A continuación, se detalla el funcionamiento de la capa convolucional (3.2), que es fundamental para la extracción de características locales de las imágenes, seguida por la capa de agrupamiento o *pooling* (3.3), que reduce dimensionalidad y mejora la eficiencia computacional. La capa densa o *fully connected* (3.4) también se analiza, describiendo cómo esta contribuye a la toma de decisiones finales del modelo. Posteriormente, se presenta un caso práctico de reconocimiento de dígitos utilizando el dataset MNIST (3.5), donde se describe minuciosamente la arquitectura del modelo empleado (3.5.2), su proceso de entrenamiento (3.5.5), y los resultados obtenidos en el conjunto de prueba (3.5.8). Este capítulo ofrece una comprensión profunda de las CNN y su aplicación práctica en el reconocimiento de imágenes, destacando su efectividad y relevancia en el campo de la inteligencia artificial.

### 3.1. Introducción

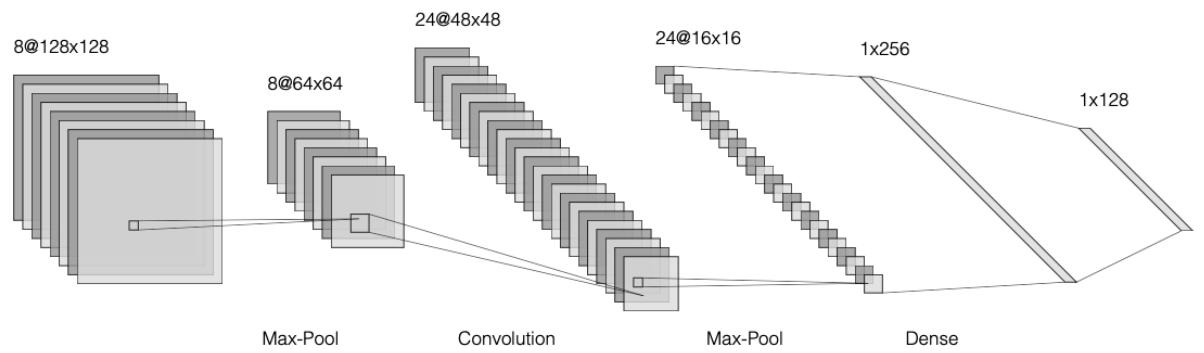


Figura 3.1: Red neuronal convolucional

Las **redes neuronales convolucionales** (CNN) [5] son un tipo especial de red neuronal para procesar datos con una topología similar a una malla, como series temporales (1D) o imágenes (2D y 3D), que se caracterizan por tener datos altamente estructurados y correlacionados espacialmente. La arquitectura de una CNN se inspira en el proceso visual biológico del campo visual humano [1] y han demostrado ser extremadamente eficientes en tareas de *computer vision*, como reconocimiento de imágenes, clasificación de imágenes y detección de objetos, entre otros. Este será nuestro objetivo en este capítulo. En la figura 3.1, se muestra la arquitectura de una CNN, que detallamos a continuación:

1. **Capas convolucionales:** Estas capas son el núcleo de las CNN. Realizan una operación matemática llamada convolución. La convolución utiliza un conjunto de **filtros** que escanean la imagen de entrada (o la salida de otra capa) y generan **mapas de activación** (o mapas de características). Cada filtro es entrenable y se adapta durante el proceso de aprendizaje para resaltar características específicas en los datos, como bordes, texturas o patrones más complejos. La convolución consiste en la multiplicación elemento por elemento de los valores del filtro con los valores originales de los datos en una región local, seguida de la suma de estos productos.
2. **Funciones de activación:** Después de cada capa convolucional, se suele aplicar una función de activación no lineal, como la función ReLU (Rectified Linear Unit) o variantes de esta. La función de activación introduce no linealidades en el modelo, permitiendo a la red aprender patrones complejos y no lineales en los datos.
3. **Capas de agrupamiento (*pooling*):** Las capas de agrupamiento reducen la dimensionalidad espacial de los mapas de activación obtenidos después de las capas convolucionales y de activación. Esto se hace para disminuir el número de parámetros, lo que ayuda a reducir el riesgo de sobreajuste y mejora la eficiencia computacional. El agrupamiento más común es el agrupamiento máximo (max pooling), que selecciona el valor máximo de una región de la entrada.
4. **Capa densa (*fully-connected layer*):** Después de varias capas convolucionales y de agrupamiento, la red suele incluir una o más capas completamente conectadas, donde cada entrada de la capa está conectada a cada salida de la capa anterior. Estas capas se utilizan para realizar la clasificación o regresión basada en las características extraídas por las capas convolucionales y de agrupamiento. La última capa completamente conectada suele tener un número de salidas igual al número de clases en una tarea de clasificación y utiliza una función de activación softmax para obtener las probabilidades de cada clase.

## 3.2. Capa convolucional

La intuición básica tras esta operación es la obtención de las características principales de una imagen. Dada una imagen  $I \in Mat_{h \times w}$ , donde  $h$  y  $w$  son la altura y anchura de la misma, le aplicaremos un **filtro** o *kernel*, que son matrices de menor dimensión y entrenables que la red utiliza para extraer características de los datos de entrada. Aplicar un filtro sobre la entrada, nos permite identificar patrones específicos como bordes, texturas, colores, o incluso características más complejas en niveles más profundos de la red.

**Definición 1** (Convolución 2D). *Dadas una matriz  $I \in Mat_{h \times w}$  y un filtro  $\omega \in Mat_{a \times b}$ , con  $0 \leq a \leq h$  y  $0 \leq b \leq w$ , el resultado de aplicar la **convolución** del filtro  $\omega$  sobre  $I$  es la matriz  $F \in Mat_{(h-a+1),(w-b+1)}$ . Esta operación se denota con el símbolo  $*$ :*

$$F(i,j) = (I * \omega)(i,j) = \sum_x \sum_y I(i+x, j+y) \cdot \omega(x,y)$$

Para poder aplicar un filtro a una matriz, necesitamos definir dos parámetros adicionales:

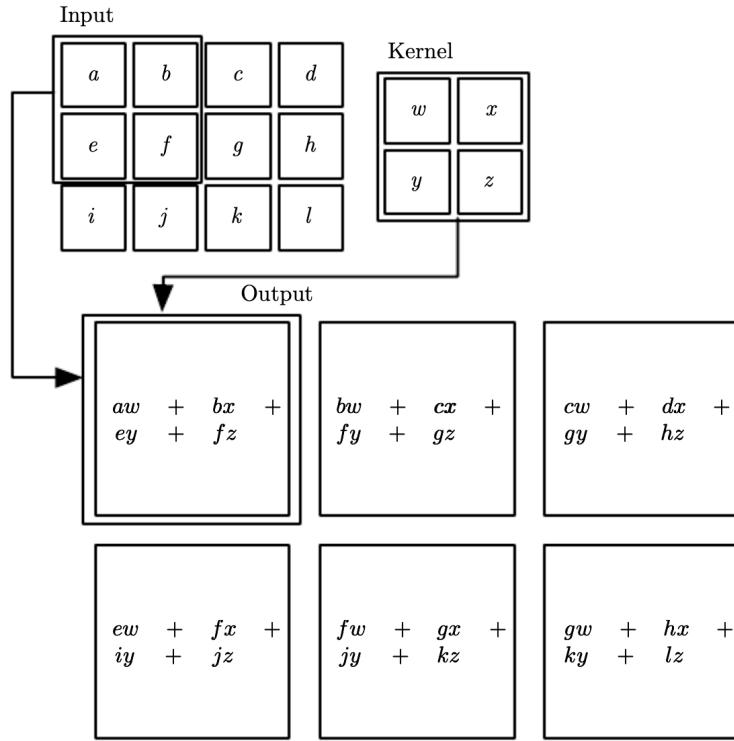


Figura 3.2: Convolución 2D [5]

- **Padding.** El *padding*,  $p \in \mathbb{N}$ , representará el número de filas y columnas nulas que añadiremos a los bordes de la matriz de la imagen sobre la que aplicaremos la convolución.

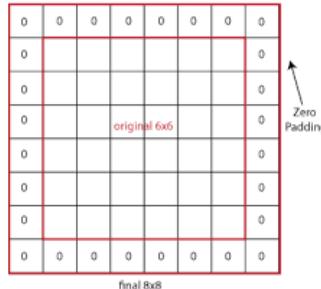


Figura 3.3: Ejemplo de *padding*  $p = 1$ .

- **Stride.** El *stride*,  $s \in \mathbb{N}$ , representa el salto entre dos posiciones consecutivas del filtro.

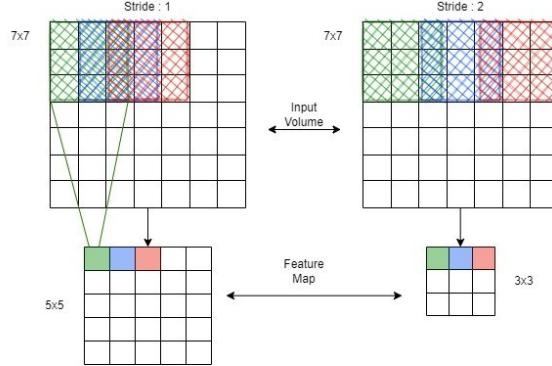


Figura 3.4: Ejemplos de *stride* 1 y 2.

Además cuando tratemos con imágenes, normalmente estas estarán dadas en formato RGB, por lo que tendrán dimensión  $3 \times h \times w$ , donde  $h$  y  $w$  son el ancho y alto, respectivamente. Para poder aplicar una convolución, el filtro también deberá tener 3 dimensiones. Por esta razón, hemos de definir la operación de convolución 3D.

**Definición 2** (Convolución 3D). *Dada una matrix,  $I \in Mat_{d \times h \times w}$ , donde  $d$ ,  $h$  y  $w$  son la profundidad, altura y anchura, y un filtro,  $\omega \in Mat_{d \times a \times b}$ , podemos escribir ambas como:  $I = (I_1, \dots, I_d)$  y  $\omega = (\omega_1, \dots, \omega_d)$ , con  $I_i \in Mat_{w \times h}$ ,  $\omega_i \in Mat_{w \times h}$ . Así, el resultado de aplicar  $\omega$  sobre  $I$  será:*

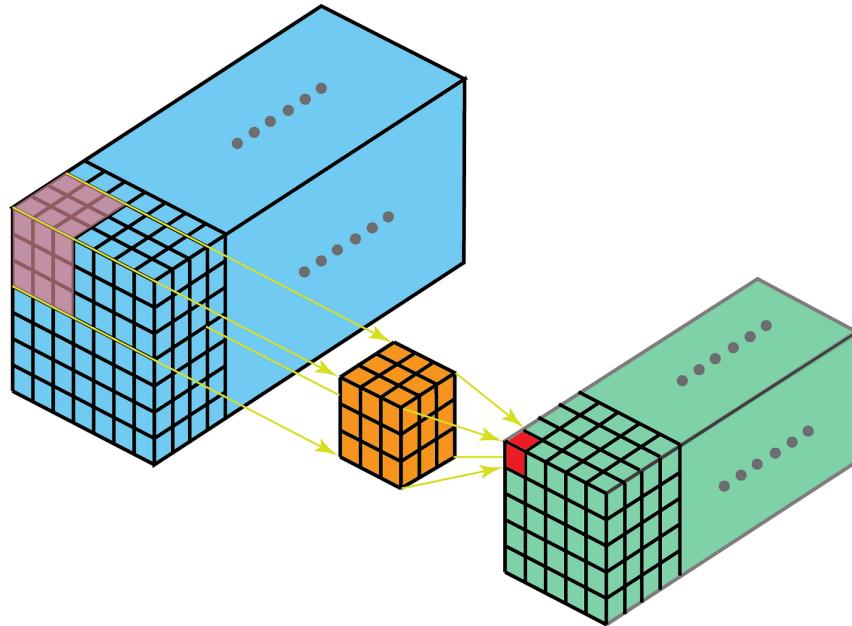
$$I * \omega = \sum_{k=1}^d I_k * \omega_k$$


Figura 3.5: Convolución 3D

El resultado de aplicar  $\omega \in Mat_{d \times a \times b}$  sobre  $I \in Mat_{d \times h \times w}$ , se conoce como **mapa de activación**, que denotaremos con  $A \in Mat_{m \times n}$ . Identificando  $s$  con el *stride* y  $p$  con el *padding*, la dimensión del mapa de activación viene dada por:

$$\blacksquare m = \frac{h-a+2p}{s} + 1$$

- $n = \frac{w-b+2p}{s} + 1$

Cuando entrenemos redes neuronales convolucionales, aplicaremos varios filtros con la finalidad de mejorar la precisión de nuestra red. Esto hará que nuestra capa convolucional produzca tantos mapas de activación como filtros hayamos aplicado. Una vez tengamos, los mapas de características resultantes, para pasar a la capa de agrupamiento, tendremos antes que aplicar las funciones de activación, cuyo resultado será  $Z \in Mat_{d_k \times m \times n}$ , con  $d_k$  el número de filtros de la capa:  $Z(i, j, k) = f_a(A(i, j, k)) \quad \forall i, j, k \quad i \in [1, d_k], \quad j \in [1, m], \quad k \in [1, n]$

El entrenamiento de una capa convolucional también requiere el gradiente de dicha capa con respecto a sus pesos correspondientes. Para simplificar el problema, estudiamos la retropropagación en una capa convolucional unidimensional. La figura 3.6 muestra dos capas de una *ConvNet* donde las neuronas de la capa derecha comparten el mismo peso y también están conectadas localmente. Esto muestra integralmente que la salida de la segunda capa se obtiene convolucionando los pesos  $W^2$  con  $H^1$ .

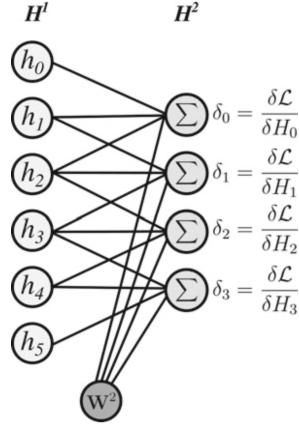


Figura 3.6: Dos capas del medio de una red neuronal indicando la convolución unidimensional. El peso  $W^2$  se comparte entre las neuronas de  $H^2$ . Además,  $\delta_i$  muestra el gradiente de las funciones de pérdida con respecto a  $H_i^2$  [1].

Como ya sabemos del capítulo anterior,  $\frac{\partial L}{\partial w_i}$  viene dado por:

$$\frac{\partial L}{\partial w_i} = \frac{\partial H_0^2}{\partial w_i} \frac{\partial L}{\partial H_0^2} + \frac{\partial H_1^2}{\partial w_i} \frac{\partial L}{\partial H_1^2} + \frac{\partial H_2^2}{\partial w_i} \frac{\partial L}{\partial H_2^2} + \frac{\partial H_3^2}{\partial w_i} \frac{\partial L}{\partial H_3^2} = \frac{\partial H_0^2}{\partial w_i} \delta_0 + \frac{\partial H_1^2}{\partial w_i} \delta_1 + \frac{\partial H_2^2}{\partial w_i} \delta_2 + \frac{\partial H_3^2}{\partial w_i} \delta_3, \quad \text{donde } \delta_i = \frac{\partial L}{\partial H_i^2}$$

Así, obtenemos :

$$\begin{aligned} \frac{\partial L}{\partial w_0} &= h_0 \delta_0 + h_1 \delta_1 + h_2 \delta_2 + h_3 \delta_3 \\ \frac{\partial L}{\partial w_1} &= h_1 \delta_0 + h_2 \delta_1 + h_3 \delta_2 + h_4 \delta_3 \\ \frac{\partial L}{\partial w_2} &= h_2 \delta_0 + h_3 \delta_1 + h_4 \delta_2 + h_5 \delta_3 \end{aligned}$$

Si denotamos  $\delta^2 = (\delta_0, \delta_1, \delta_2, \delta_3)$  como el vector de gradientes de  $H^2$  y  $h^1 = (h_0, h_1, h_2, h_3, h_4, h_5)$  como la salida de las neuronas en  $H^1$ , entonces:  $\frac{L}{W} = (\frac{L}{w_0}, \frac{L}{w_1}, \frac{L}{w_2}) = h^1 * \delta^2$

También tendremos que calcular el error en  $H^1$  para pasarlo a las capas previas:

$$\begin{aligned} \frac{\partial L}{\partial h_0} &= \frac{\partial H_0^2}{\partial h_0} \delta_0, \quad \frac{\partial L}{\partial h_1} = \frac{\partial H_0^2}{\partial h_1} \delta_0 + \frac{\partial H_1^2}{\partial h_1} \delta_1, \quad \frac{\partial L}{\partial h_2} = \frac{\partial H_0^2}{\partial h_2} \delta_0 + \frac{\partial H_1^2}{\partial h_2} \delta_1 + \frac{\partial H_2^2}{\partial h_2} \delta_2, \quad \frac{\partial L}{\partial h_3} = \frac{\partial H_0^2}{\partial h_3} \delta_0 + \frac{\partial H_1^2}{\partial h_3} \delta_1 + \frac{\partial H_2^2}{\partial h_3} \delta_2 + \frac{\partial H_3^2}{\partial h_3} \delta_3, \\ \frac{\partial L}{\partial h_4} &= \frac{\partial H_2^2}{\partial h_4} \delta_2 + \frac{\partial H_3^2}{\partial h_4} \delta_3, \quad \frac{\partial L}{\partial h_5} = \frac{\partial H_3^2}{\partial h_5} \delta_3 \end{aligned}$$

### 3.3. Capa de agrupamiento

La función principal de la capa de agrupamiento es reducir progresivamente la dimensión espacial (ancho y alto, no la profundidad) de los mapas de características entrantes, lo que disminuye la cantidad de parámetros y cálculos en la red, y por ende, mejora la eficiencia computacional. Además, al realizar esta reducción, la capa de agrupamiento contribuye a hacer que la representación de las características sea más invariante a pequeñas traslaciones en la entrada, aumentando así la capacidad de la red para generalizar. Las características de esta capa serán:

- **Tamaño de la ventana:** Define el área sobre la que se aplicará la operación de agrupamiento.
- **Stride:** Al igual que con los filtros, indica cuántos píxeles se desplaza la ventana de agrupamiento después de cada operación. Un stride mayor resulta en una reducción más significativa de las dimensiones.
- **Padding:** Aunque bastante menos común que en las capas convolucionales, el padding puede usarse para ajustar cómo se manejan los bordes de los mapas de características.

**Definición 3** (Operación de agrupamiento). *Una función de agrupamiento,  $f$ , es cualquier aplicación que tiene como conjunto de llegada los números reales, es decir:  $f : \mathbb{R}^n \rightarrow \mathbb{R}$ . Una operación de agrupamiento es el resultado de aplicar cualquier función de agrupamiento a una matriz  $\text{Mat}_{m \times n}$*

Las funciones de agrupamiento más populares son:

1. **Max-pooling:** selecciona el valor máximo de una región definida de los mapas de características. Es el tipo de agrupamiento más utilizado.

$$\text{maxpooling}(A) := \max(A), \text{ donde } A \in \text{Mat}_{m \times n}.$$

2. **Average pooling:** calcula el valor medio de los elementos dentro de la región definida de los mapas de características.

$$\text{average\_pooling}(A) := \frac{\sum a_{i,j}}{m \cdot n}, \text{ donde } A \in \text{Mat}_{m \times n}.$$

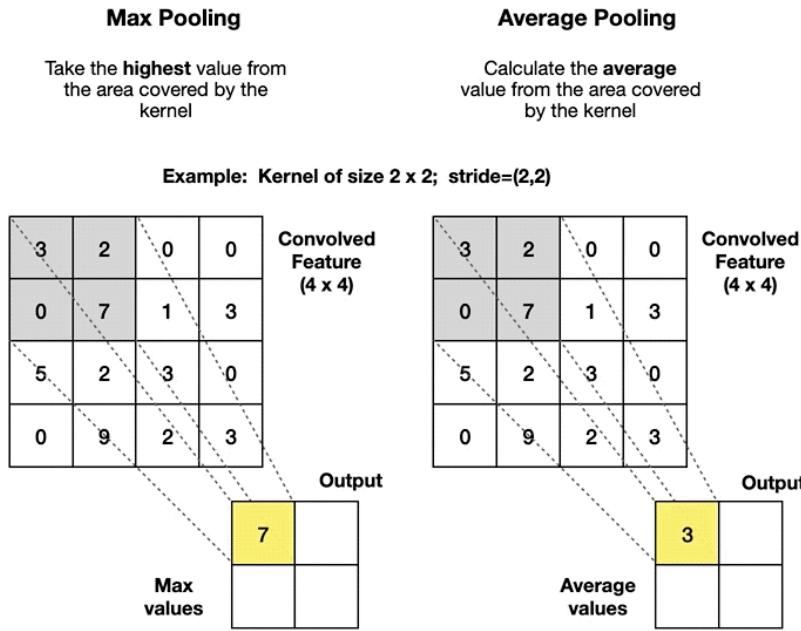


Figura 3.7: Funciones de agrupamiento

Las capas de agrupamiento no tienen parámetros entrenables, a diferencia de las capas convolucionales. Por tanto, su gradiente se calcula respecto a la capa anterior sin necesidad de ajuste de parámetros. En *max-pooling*, el gradiente se transfiere solo a la neurona que contribuyó al valor máximo, mientras que las demás neuronas no reciben gradiente. En *average pooling*, el gradiente se distribuye equitativamente entre todas las entradas de la región de agrupamiento. Por ejemplo, en la figura 3.8 que muestra una capa de ejemplo, el gradiente se calcularía de la siguiente forma:

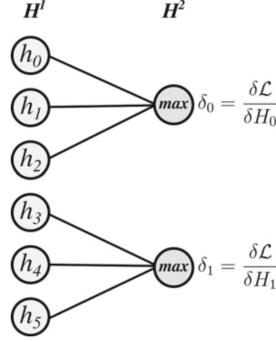


Figura 3.8: Una capa de agrupamiento máximo unidimensional donde las neuronas en  $H^2$  calculan el máximo de sus entradas [1].

$$\begin{aligned} \frac{\partial H_0^2}{\partial h_0} &= \begin{cases} 1 & \max(h_0, h_1, h_2) == h_0 \\ 0 & \text{resto} \end{cases} \\ \frac{\partial H_0^2}{\partial h_1} &= \begin{cases} 1 & \max(h_0, h_1, h_2) == h_1 \\ 0 & \text{resto} \end{cases} \\ \frac{\partial H_0^2}{\partial h_2} &= \begin{cases} 1 & \max(h_0, h_1, h_2) == h_2 \\ 0 & \text{resto} \end{cases} \\ \frac{\partial H_1^2}{\partial h_3} &= \begin{cases} 1 & \max(h_3, h_4, h_5) == h_3 \\ 0 & \text{resto} \end{cases} \\ \frac{\partial H_1^2}{\partial h_4} &= \begin{cases} 1 & \max(h_3, h_4, h_5) == h_4 \\ 0 & \text{resto} \end{cases} \\ \frac{\partial H_1^2}{\partial h_5} &= \begin{cases} 1 & \max(h_3, h_4, h_5) == h_5 \\ 0 & \text{resto} \end{cases} \end{aligned}$$

### 3.4. Capa densa

El último paso en una red convolucional es la capa densa o *fully-connected layer*, que se trata de una capa neuronal tradicional (ya definidas en el capítulo 2), donde cada elemento de la matriz resultante de la capa de agrupamiento se conecta a cada una de las neuronas de la capa densa. Esto permite realizar una transformación lineal para poder realizar una clasificación de la información procesada en las capas anteriores.

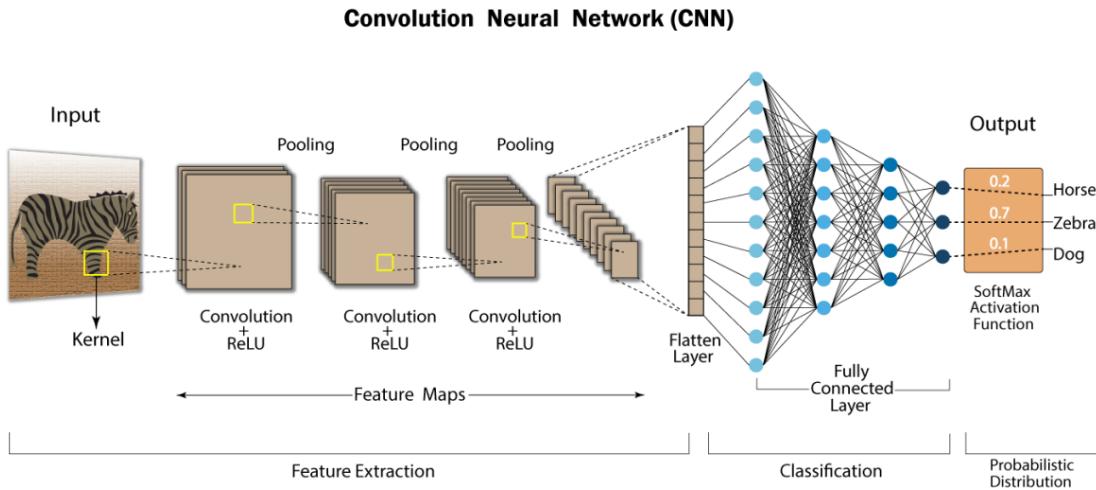


Figura 3.9: Ejemplo de red neuronal convolucional

Con esto, ya tenemos todos los conceptos necesarios para afrontar el proyecto práctico de este capítulo: la clasificación de dígitos escritos.

### 3.5. Reconocimiento de dígitos

Esta sección tiene como objetivo implementar y entrenar una Red Neuronal Convolutacional para el reconocimiento de dígitos escritos a mano, utilizando el conjunto de datos MNIST. El objetivo es demostrar de forma empírica la mejora que supone el uso de redes neuronales convolucionales para procesamiento de imágenes, con respecto a una red neuronal artificial como la utilizada en el capítulo anterior.

El código utilizado se encuentra en el archivo `cnn_MNIST.ipynb` del repositorio del proyecto<sup>1</sup>.

#### 3.5.1. Dataset MNIST

En este caso, en lugar de cargar las imágenes como vectores, lo haremos directamente como matrices de dimensión  $28 \times 28$

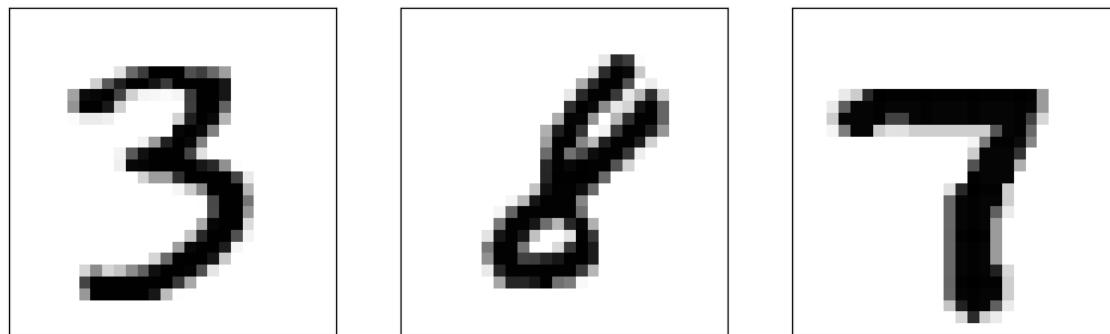


Figura 3.10: Primeras 3 muestras del conjunto de entrenamiento del dataset MNIST.

---

<sup>1</sup>Github del proyecto: <https://github.com/miguelangelmoralesramon/tfg.git>

### 3.5.2. Arquitectura

He implementado dos arquitecturas, la primera en busca de la máxima precisión posible y la segunda, en busca de la mejor relación tiempo de entrenamiento/precisión.

### 3.5.3. Modelo A

1. Capa convolucional: 16 filtros de tamaño 5x5,  $stride=1$  y  $padding=2$ , con función de activación ReLU. El tamaño de los mapas resultantes se mantiene en 28x28.
2. Capa de agrupamiento: *max-pooling* con filtro de tamaño 2x2,  $stride=1$  y  $padding=2$ . Se reduce el tamaño de los mapas a la mitad, 14x14.
3. Capa convolucional: 32 filtros de tamaño 5x5,  $stride=1$  y  $padding=2$ , con función de activación ReLU. El tamaño de los mapas resultantes se mantiene en 14x14.
4. Capa de agrupamiento: *max-pooling* con filtro de tamaño 2x2,  $stride=1$  y  $padding=2$ . Se reduce el tamaño de los mapas a la mitad, 7x7.
5. Capa densa: el tamaño resultante es 1568 ( $32 \times 7 \times 7$ ).
6. Capa neuronal: de tamaño de entrada 1568 y tamaño de salida 512.
7. Capa de *dropout*: con 50 % de probabilidad.
8. Capa neuronal: de tamaño de entrada 512 y tamaño de salida 10.

### 3.5.4. Modelo B

1. Capa convolucional: 4 filtros de tamaño 5x5,  $stride=1$  y  $padding=2$ , con función de activación ReLU. El tamaño de los mapas resultantes se mantiene en 28x28.
2. Capa de agrupamiento: *max-pooling* con filtro de tamaño 2x2,  $stride=1$  y  $padding=2$ . Se reduce el tamaño de los mapas a la mitad, 14x14.
3. Capa densa: el tamaño resultante es 784 ( $4 \times 14 \times 14$ ).
4. Capa neuronal: de tamaño de entrada 784 y tamaño de salida 64.
5. Capa neuronal: de tamaño de entrada 64 y tamaño de salida 10.

### 3.5.5. Entrenamiento

### 3.5.6. Modelo A

Para el entrenamiento he dividido los conjuntos de entrenamiento y validación en *batch sizes* de 64 muestras y han sido necesarias 20 *epochs*. Para calcular la pérdida he utilizado la función de *Cross Entropy Loss* ya que es la más representativa en problemas de clasificación múltiple. He probado varias tasas de aprendizaje,  $\eta \in [10^{-4}, 10^{-1}]$ , obteniendo un mejor resultado con  $\eta \approx 10^{-3}$ . El tiempo de entrenamiento con esta arquitectura y parámetros ha sido de **1082.14 segundos**. En la figura 3.11, se muestran los resultados de entrenamiento con estos parámetros.

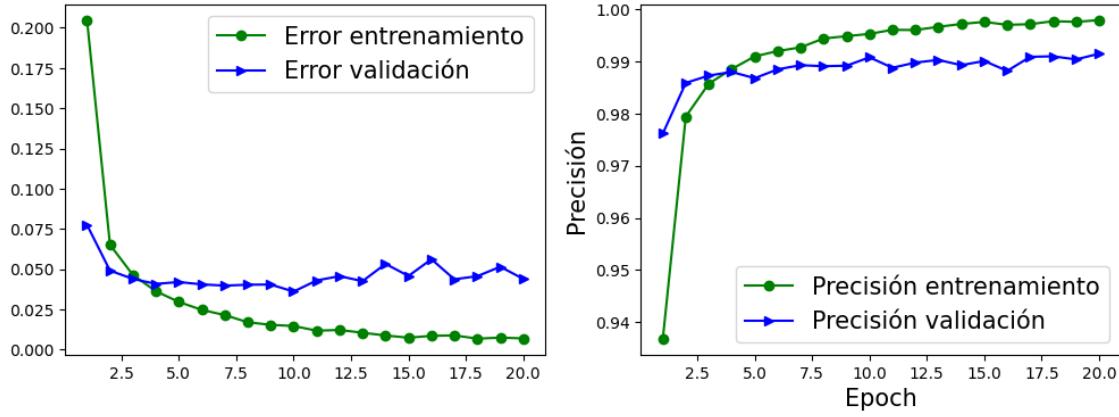


Figura 3.11: Resultados de entrenamiento con  $\eta = 0,001$ ,  $batch\_size = 64$  y  $n\_epochs = 20$ .

### 3.5.7. Modelo B

En este caso he seleccionado un tamaño de lote de 100 muestras y han sido necesarias 10 epochs. Para calcular la pérdida también he utilizado la función de *Cross Entropy Loss*. He probado varias tasas de aprendizaje,  $\eta \in [10^{-4}, 10^{-1}]$ , obteniendo un mejor resultado con  $\eta \approx 10^{-3}$ . El tiempo de entrenamiento con esta arquitectura y parámetros ha sido de **159.97 segundos**. En la figura 3.12, se muestran los resultados de entrenamiento con estos parámetros.

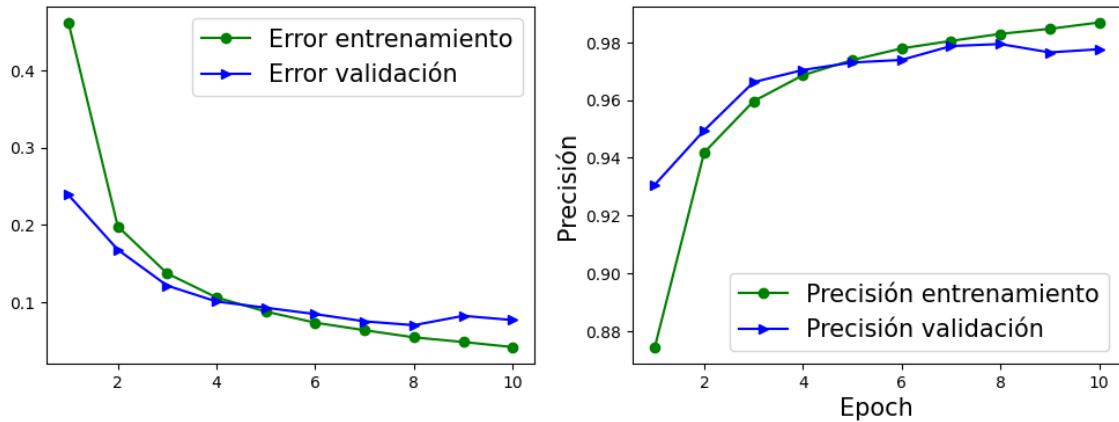


Figura 3.12: Arquitectura de la red convolucional para el reconocimiento de dígitos.

### 3.5.8. Resultados del conjunto de prueba

Finalmente, he medido la pérdida de ambos modelos, enfrentándolos al conjunto de prueba. El *modelo A* ha obtenido una precisión final del 99,37 % y el *Modelo B* del 97,98 %. En el *modelo A*, esto supone una mejora del  $3,64 \pm 0,01$  % con respecto a la red neuronal empleada en el capítulo anterior, pero con **921.37 segundos más** de entrenamiento. Por contrapartida, el *modelo B* mejora en un  $2,25 \pm 0,01$  % al modelo del capítulo anterior, con **0.79 segundos menos** de entrenamiento. En la figura 3.13 se muestran algunas predicciones del modelo sobre 12 muestras del conjunto de prueba.

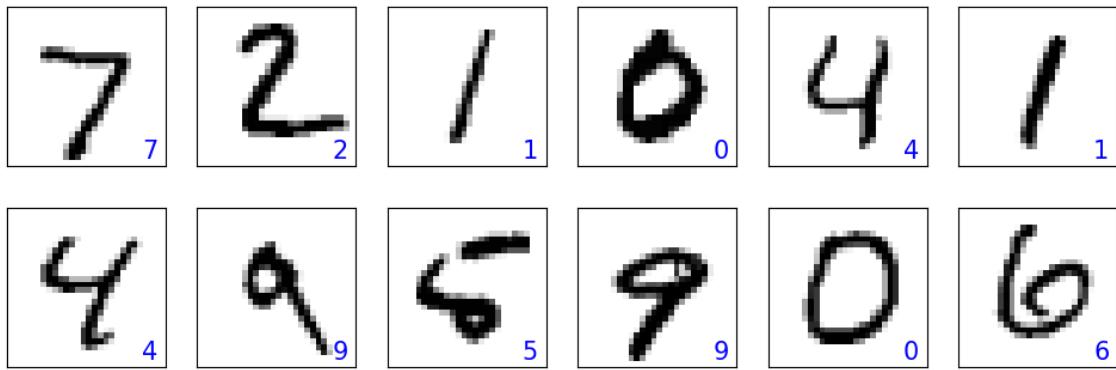


Figura 3.13: Resultados de los modelos A y B contra una muestra de 12 imágenes del conjunto de prueba. En azul se muestra la predicción del modelo.

# Capítulo 4

## Redes neuronales recurrentes

Este capítulo se centra en las redes neuronales recurrentes (RNN) y su aplicación en el análisis de secuencias temporales, una tarea crucial en diversos campos como el procesamiento del lenguaje natural y la predicción de series temporales. Se comienza con una explicación de la arquitectura básica de las RNN (4.1), abordando su diseño y funcionamiento fundamental (4.2). Posteriormente, se profundiza en dos variantes importantes: las celdas *Long Short-Term Memory* (LSTM) (4.3) y *Gated Recurrent Units* (GRU) (4.4), que han sido desarrolladas para mejorar la capacidad de las RNN tradicionales de manejar memoria a largo plazo. Finalmente, se presenta un caso práctico de análisis de reviews de IMDb (4.5), en el cual se detalla el dataset utilizado (4.5.1), la arquitectura del modelo diseñado (4.5.2), el proceso de entrenamiento implementado (4.5.3) y los resultados obtenidos (4.5.4). Este capítulo proporciona una visión integral de cómo las RNN y sus variantes avanzadas pueden aplicarse eficazmente al análisis de secuencias temporales, subrayando su potencial y versatilidad en diferentes dominios de aplicación.

### 4.1. Arquitectura

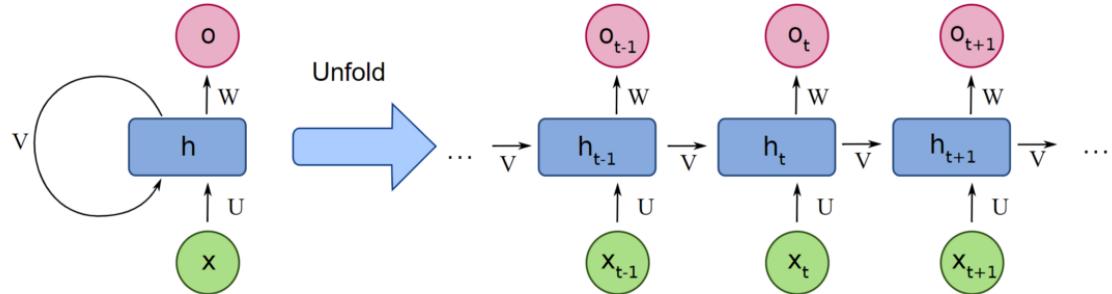


Figura 4.1: Arquitectura básica de una red neuronal recurrente. El grafo de la izquierda se denomina **grafo plegado** o *fold* y el de la derecha, **grafo desplegado** o *unfold*.

Una red neuronal recurrente (RNN) [15] es una clase de red neuronal artificial diseñada para procesar secuencias de datos, aprovechando su capacidad para mantener un estado o memoria sobre los datos anteriores que ha visto, lo cual es útil en tareas de procesamiento de lenguaje natural, reconocimiento de voz, y otras aplicaciones de procesado de datos secuenciales. La característica distintiva de una RNN es su estructura cíclica, que permite mantener información de estados anteriores.

Como sucede con las redes neuronales convolucionales, las redes neuronales recurrentes pueden estar compuestas por capas neuronales tradicionales. Sin embargo, lo que las distingue de estas es la capa neuronal recurrente.

**Definición 4** (Capa neuronal recurrente). *Una capa neuronal recurrente es un sistema dinámico  $S^t := \{x^t, h^t, o^t, \{W_{hh}, W_{xh}, W_{ho}\}, f_a\}$  tal que  $h^t = f_a(W_{hh}h^{t-1} + W_{xh}x^t + b_h)$ , donde:*

- $f_a$  es la función de activación de la capa
- $h^t$  es el estado actual o valor de la neurona oculta en el tiempo  $t$ . Es el equivalente a  $a^t$  en las redes neuronales artificiales, con la diferencia de que depende del valor oculto del tiempo anterior. Se calcula como sigue:  

$$h^t = f_a(W_{hh}h^{t-1} + W_{xh}x^t + b_h)$$
- $h^{t-1}$  es el estado anterior, es decir, el valor de la neurona oculta de la capa en el tiempo  $t - 1$ .
- $W_{xh} \in Mat$  es la matriz de pesos para la conexión entre el input y capa oculta.
- $W_{hh} \in Mat$  es la matriz de pesos para la conexión entre el estado anterior,  $h^{t-1}$ , y el estado actual,  $h^t$ .
- $b_h$  es el sesgo para la neurona oculta.
- $o^t$  es el output o salida en el tiempo  $t$ :  

$$o^t = W_{ho}h^t + b_o$$
- $W_{ho} \in Mat$  es la matriz de pesos para la conexión entre el estado actual,  $h^t$ , y el output,  $o^t$ .
- $b_o$  es el sesgo para la salida.
- $x^t$  es la entrada de la capa en el tiempo  $t$ .

## 4.2. Diseño de redes neuronales recurrentes

Desarrollando el concepto básico de capa neuronal recurrente, podemos diseñar distintos tipos de redes dependiendo de dónde establezcamos la conexión recurrente. De esta forma, podríamos calcular el estado actual de la neurona oculta ( $h^t$ ) en función de la estado anterior ( $h^{t-1}$ ) o en función de la salida anterior ( $o^{t-1}$ ). Los patrones más comunes son los siguientes:

- RNNs que producen una salida en cada unidad de tiempo y tienen conexiones recurrentes entre las capas ocultas. En la figura 4.2, se muestra un grafo para calcular la pérdida durante el entrenamiento de una red recurrente de tipo oculta-oculta, que establece una correspondencia entre la secuencia de entrada  $\mathbf{x}$  y la secuencia de salida  $\mathbf{o}$ . La función de pérdida  $\mathbf{L}$  mide la distancia entre  $\mathbf{o}$  y la variable objetivo  $\mathbf{y}$ .  $\mathbf{L}$  calcula la predicción  $\hat{\mathbf{y}} = softmax(\mathbf{o})$  y la compara con la variable objetivo. La RNN tiene conexiones entrada-oculta, parametrizadas por la matriz de pesos  $\mathbf{U}$ , conexiones oculta-oculta, parametrizadas por la matriz de pesos  $\mathbf{W}$  y conexiones oculta-salida, parametrizadas por la matriz  $\mathbf{V}$ . [5]

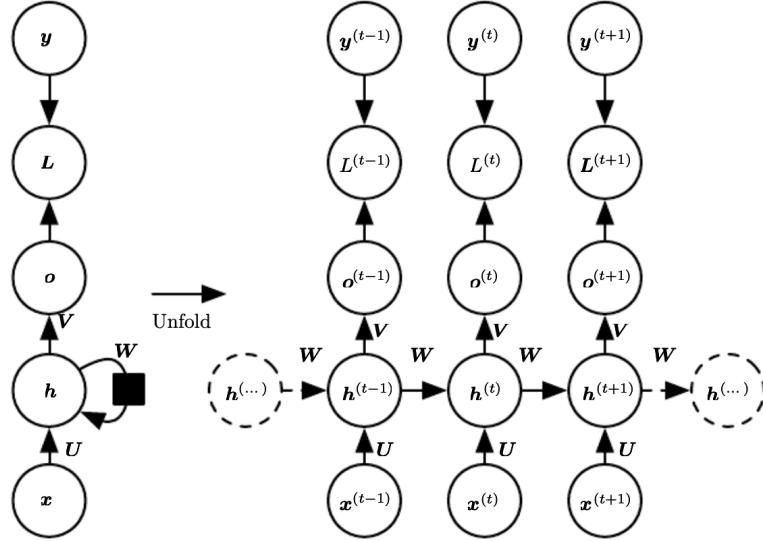


Figura 4.2: Grafo de RNN con conexiones recurrentes capa a capa.

- RNNs que producen una salida en cada unidad de tiempo y únicamente tienen conexiones recurrentes entre la salida en  $t$  y la capa oculta en  $t+1$ . Este tipo de red, tiene menos contexto, que la anterior, en el sentido de que, en este tipo de red únicamente podemos alimentar la siguiente capa con el dato de salida  $\mathbf{o}$ . Sin embargo, cuando tenemos conexiones recurrentes capa oculta-capacapta oculta, podemos elegir qué información de la capa anterior,  $h^t$ , queremos enviar a la siguiente capa,  $h^{t+1}$ . La ventaja de este tipo de red es más fácil de entrenar puesto que en cada unidad de tiempo se puede entrenar de forma aislada, lo que permite que la paralelización durante el entrenamiento sea posible. En la figura 4.3 se muestra una RNN donde la única recurrencia es entre la salida y la capa oculta. En cada unidad de tiempo, la entrada es  $\mathbf{x}^t$ , las activaciones de la capa oculta,  $\mathbf{h}^t$ , la salida  $\mathbf{o}^t$ , la variable objetivo es  $\mathbf{y}^t$  y la pérdida es  $L^t$ . [5]

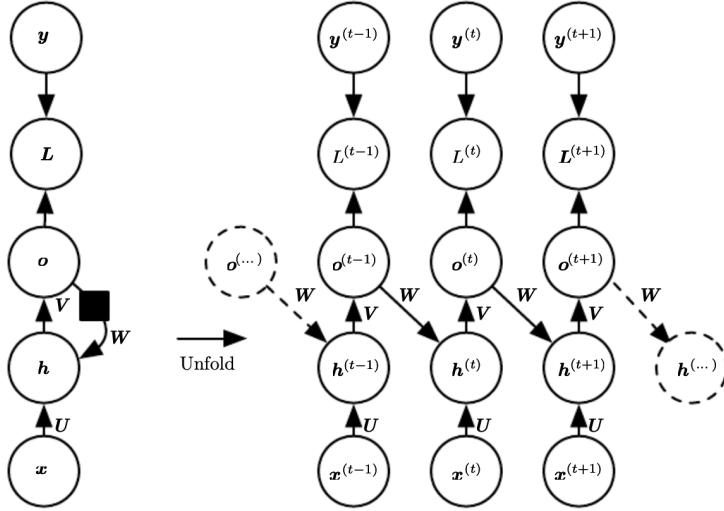


Figura 4.3: RNN con conexión entre la salida y la capa oculta.

- RNNs con conexiones recurrentes entre capas ocultas, que leen una secuencia entera y producen una única salida. Estas redes pueden utilizarse para resumir una secuencia y producir una salida de tamaño fijo. En la figura 4.4 se muestra una red de este tipo.

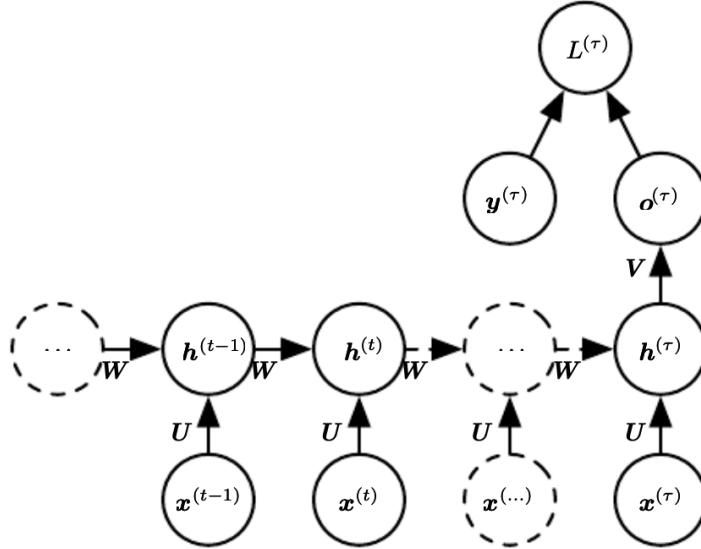


Figura 4.4: RNN con conexiones capa oculta-capía oculta, que procesan la secuencia y producen una salida de tamaño fijo. [5]

Nos centraremos en desarrollar las ecuaciones del primer tipo de red recurrente, puesto que estas son las más comunes. La función de activación más utilizada en este tipo de redes es la función tangente hiperbólica,  $f_a(x) := \tanh(x)$ , por los siguientes motivos:

1. **Gradiente desvaneciente.** Es el principal problema de este tipo de redes. En el algoritmo de

retropropagación, al actualizar los pesos, el gradiente de las primeras capas de redes profundas es prácticamente nulo, por lo que estas son mucho más difíciles de entrenar. Para solventar este problema, necesitamos una función cuya segunda derivada pueda mantenerse durante un largo período antes de llegar a cero. La derivada de la función tangente hiperbólica es más *fuerte* (es decir, más alejada de cero) para entradas cercanas a cero en comparación con la función sigmoide. Esto significa que para las activaciones cercanas al centro de su rango, *tanh* proporciona gradientes más significativos, facilitando un aprendizaje más efectivo durante la retropropagación.

2. **Centrada en cero.** A diferencia de la función sigmoide, cuyo rango está entre 0 y 1, la función *tanh* tiene un rango de salida entre -1 y 1. Este rango centrado en cero ayuda a mantener la media de las salidas cercana a 0 a lo largo del tiempo, lo que puede mejorar la convergencia durante el entrenamiento de la red.
3. **Simetría.** La simetría de la función *tanh* alrededor del origen (es decir,  $\tanh(-x) = -\tanh(x)$ ) facilita el aprendizaje de pesos para minimizar el error, ya que las actualizaciones de peso pueden ajustarse de manera eficiente tanto en direcciones positivas como negativas, dependiendo de la dirección del error.

Con esto, para comenzar la propagación hacia adelante, necesitamos especificar un estado inicial  $h^0$ , y para cada unidad de tiempo  $t \in [1, N]$ , aplicamos las siguientes ecuaciones:

$$\begin{aligned} a^t &= W \cdot h^{t-1} + U \cdot x^t + b \\ h^t &= \tanh(a^t) \\ o^t &= V \cdot h^t + c \\ \hat{y}^t &= \text{softmax}(o^t) \end{aligned}$$

Una vez realizada la propagación hacia adelante, debemos aplicar el algoritmo de retropropagación. En el caso de las redes recurrentes, se utiliza una variante específica conocida como **algoritmo de retropropagación a través del tiempo** (algoritmo BPTT) [5].

### 4.3. Long Short-Term Memory (LSTM)

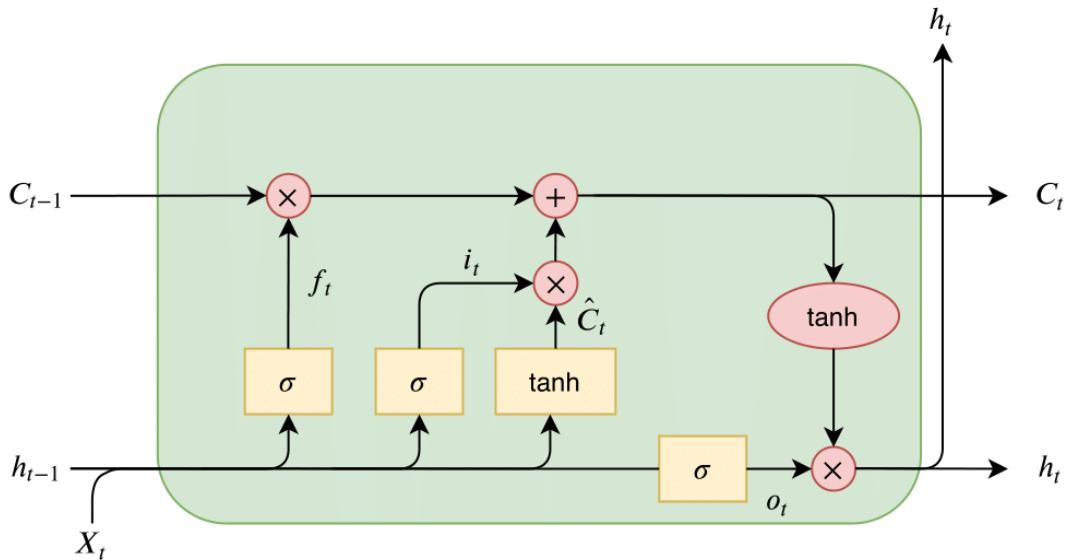


Figura 4.5: Arquitectura de una celda LSTM.

Las celdas *Long Short-Term Memory* (LSTM) son una variante especializada de las redes neuronales recurrentes, diseñadas para solucionar el problema del desvanecimiento y explosión del gradiente que ocurre con las RNN tradicionales, especialmente en el aprendizaje de dependencias a largo plazo. Las LSTM logran esto a través de una estructura de celdas compleja que incluye mecanismos de puertas (*gates*) para regular el flujo de información.

Una celda LSTM gestiona su estado a través del tiempo con tres componentes principales: una **puerta de olvido** (*forget gate*), una **puerta de entrada** (*input gate*), y una **puerta de salida** (*output gate*). Estas puertas determinan qué información debe ser recordada, actualizada o olvidada en cada paso de tiempo, permitiendo que la red mantenga dependencias a largo y corto plazo de manera efectiva. En la figura 4.5 se muestra la arquitectura de una celda LSTM.

**Notación 1.** Para este tipo de red utilizamos la siguiente notación:

- $x_t$ : entrada en el tiempo  $t$ .
- $h_t$ : estado de las unidades ocultas en el tiempo  $t$ .
- $C_t$ : estado de la celda en el tiempo  $t$ .
- $f_t$ : activación de la puerta de olvido en el tiempo  $t$ .
- $i_t$ : activación de la puerta de entrada en el tiempo  $t$ .
- $o_t$ : activación de la puerta de salida en el tiempo  $t$ .
- $W_{hf}, W_{hi}, W_{hc}, W_{ho}$  representan las matrices de peso correspondientes a las conexiones entre  $h_{t-1}$  y la puerta de olvido, entrada y salida, respectivamente.
- $W_{xf}, W_{xi}, W_{xc}, W_{xo}$  representan las matrices de peso correspondientes a las conexiones entre  $x_t$  y la puerta de olvido, entrada y salida, respectivamente.

La puerta de olvido ( $f_t$ ) permite a la memoria de la celda reiniciar el estado y decide qué información puede seguir siendo propagada. Este valor se calcula como se muestra en la figura 4.5:

$$f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f)$$

La puerta de entrada ( $i_t$ ) y el valor candidato ( $\hat{C}_t$ ) son los responsables de actualizar el estado de la celda ( $C_t$ ), de la siguiente forma:

$$i_t = \sigma(W_{xi}x_t + W_{hi}h_{t-1} + b_i)$$

$$\hat{C}_t = \tanh(W_{xc}x_t + W_{hc}h_{t-1} + b_c)$$

$C_t = (C_{t-1} \odot f_t) \oplus (i_t \odot \hat{C}_t)$ , donde  $\odot, \oplus$  representan el producto y la suma uno a uno, respectivamente.

La puerta de salida ( $o_t$ ) decide cómo actualizar los valores de las unidades ocultas, tal que así:

$$o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o)$$

Y con esto, calculamos el valor de las unidades ocultas en  $t$ :

$$h_t = o_t \odot \tanh(C_t)$$

## 4.4. Gated Recurrent Unit (GRU)

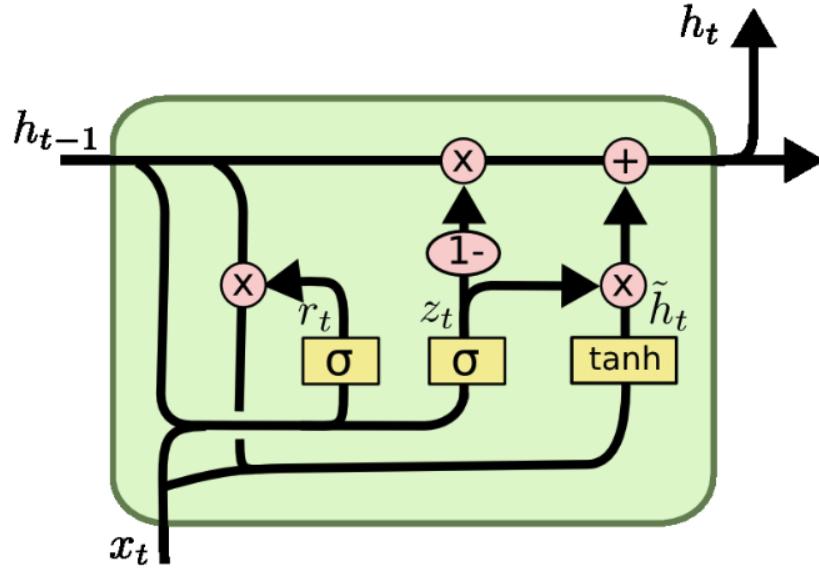


Figura 4.6: Arquitectura de una celda GRU.

La **Gated Recurrent Unit** (GRU) es una variante de las redes neuronales recurrentes diseñadas para capturar dependencias de diferentes escalas de tiempo de manera más efectiva que las RNNs estándar, y con una estructura menos compleja que las LSTM. Las GRUs simplifican el modelo de LSTM mediante la combinación de las puertas de olvido y entrada en una sola puerta de actualización, reduciendo así el número total de parámetros. Tienen únicamente dos puertas: la puerta de actualización y la puerta de reinicio. En la figura 4.6 se muestra la arquitectura de una celda GRU.

**Notación 2.** Para este tipo de red recurrente, utilizamos la siguiente notación:

- $x_t$ : entrada en el tiempo  $t$ .
- $h_t$ : estado de las unidades ocultas en el tiempo  $t$ .
- $\hat{h}_t$ : se denomina candidato a estado oculto en el tiempo  $t$ .
- $r_t$ : activación de la puerta de reinicio en el tiempo  $t$ .
- $z_t$ : activación de la puerta de actualización en el tiempo  $t$ .
- $W_{hr}, W_{hz}, W_{\hat{h}h}$  representan las matrices de peso correspondientes a las conexiones entre  $h_{t-1}$  y la puerta de reinicio, actualización y  $\hat{h}_t$ , respectivamente.
- $W_{xr}, W_{xz}, W_{x\hat{h}}$  representan las matrices de peso correspondientes a las conexiones entre  $x_t$  y la puerta de reinicio, actualización y  $\hat{h}_t$ , respectivamente.

De forma similar a las celdas LSTM, calculamos los distintos valores internos como se muestra en la figura 4.6:

$$r_t = \sigma(W_{xr}x_t + W_{hr}h_{t-1})$$

$$z_t = \sigma(W_{xz}x_t + W_{hz}h_{t-1})$$

$$h_t = z_t \odot h_{t-1} \oplus (1 - z_t) \odot \hat{h}_t, \text{ donde } \hat{h}_t = \tanh(W_{x\hat{h}}x_t + W_{\hat{h}h}(r_t \odot h_{t-1}))$$

## 4.5. Análisis de reviews de IMDb

IMDb (Internet Movie Database) es una base de datos que recopila información relacionada con películas, programas de televisión, producción cinematográfica, videojuegos y, en general, todo lo relacionado con el mundo del entretenimiento audiovisual. Se ha convertido en una herramienta esencial para aficionados al cine, profesionales de la industria y críticos.

El código utilizado se encuentra en el archivo `Analisis_de_reviews_IMDb.ipynb` del repositorio del proyecto<sup>1</sup>.

### 4.5.1. Dataset

El conjunto de datos de reseñas de películas consiste en 50000 reseñas de películas que están etiquetadas como positivas o negativas; aquí, positivo significa que una película fue calificada con más de seis estrellas en IMDb, y negativo significa que una película fue calificada con menos de cinco estrellas.

- Datos: cada elemento del dataset es un par de *strings*, donde la primera corresponde a una reseña y la segunda, a la etiqueta 'pos' (positivo) o 'neg' (negativo).
- División: el dataset tiene una partición que lo divide en conjunto de entrenamiento y conjunto de testing, cada uno con 25000 muestras. Del conjunto de entrenamiento tomaremos 5000 muestras para el conjunto de validación.
- Tratamiento de los datos. Las etiquetas las cambiaremos de 'pos' y 'neg' a 1 y 0, respectivamente. En cuanto a las reseñas, aplicaremos un filtro para quitar el formato HTML y cualquier carácter que no sea una letra o un espacio. Aprovecho la iteración sobre todo el dataset para contar y almacenar el número de palabras únicas de este (69023), que determina el tamaño de nuestro vocabulario. Después, utilizo el módulo *Vocab* para asignarle a cada palabra un entero y añado los tokens "`<pad>`", de relleno, y "`<unk>`", para tokens desconocidos, a los que asigno 0 y 1, respectivamente. Por último, aplicamos a cada lote esta transformación a través de la función *collate\_batch*, que también iguala las longitudes de las secuencias de cada lote, introduciendo tokens de relleno ("`<pad>`") cuando sea necesario. Con esto, ya tenemos las secuencias listas para aplicar el *embedding*, es decir, para pasárlas a una codificación numérica que pueda entender nuestro modelo. El funcionamiento de los *embeddings* se explica en detalle en el siguiente capítulo.

### 4.5.2. Arquitectura

Se aplicarán 3 arquitecturas distintas, con el objetivo de ver el rendimiento de cada uno de los modelos expuestos en el capítulo.

#### Modelo RNN

1. Capa de *embedding* cuyo tamaño de entrada es la longitud del vocabulario (69023) y tamaño de salida 20.
2. Capa recurrente con tamaño de entrada 20 y tamaño de salida 64 unidades.
3. Capa densa de tamaño entrada 64 y tamaño de salida 64, con función de activación *ReLU*.
4. Capa densa con tamaño de entrada 64 y tamaño de salida 1, con función de activación sigmoide.

---

<sup>1</sup>Github del proyecto: <https://github.com/miguelangelmoralesramon/tfg.git>

## Modelo LSTM

1. Capa de *embedding* cuyo tamaño de entrada es la longitud del vocabulario (69023) y tamaño de salida 20.
2. Capa recurrente con tamaño de entrada 20 y 32 unidades LSTM.
3. Capa densa de tamaño entrada 32 y tamaño de salida 64, con función de activación *ReLU*.
4. Capa densa con tamaño de entrada 64 y tamaño de salida 1, con función de activación sigmoide.

## Modelo GRU

1. Capa de *embedding* cuyo tamaño de entrada es la longitud del vocabulario (69023) y tamaño de salida 20.
2. Capa recurrente con tamaño de entrada 20 y 32 unidades GRU.
3. Capa densa de tamaño entrada 32 y tamaño de salida 64, con función de activación *ReLU*.
4. Capa densa con tamaño de entrada 64 y tamaño de salida 1, con función de activación sigmoide.

### 4.5.3. Entrenamiento

Para los 3 modelos he utilizado una tasa de aprendizaje  $\eta = 0,001$  y la función de pérdida *Binary Cross-Entropy Loss*, puesto que esta es la que más representativa en tareas de clasificación binaria.

## Modelo RNN

Este modelo ha sido entrenado en 15 *epochs*, con un tiempo de entrenamiento total de 1893,13s. La precisión obtenida en este proceso ha sido de 87,8% y 76,42% para los conjuntos de entrenamiento y validación, respectivamente. En la figura 4.7, se muestran los resultados de entrenamiento de este modelo.

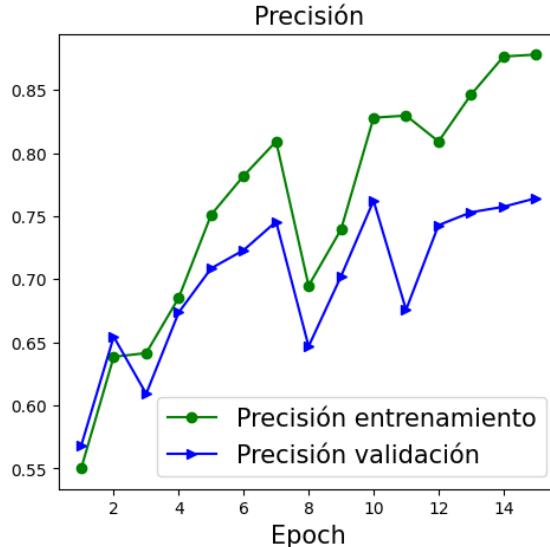


Figura 4.7: Precisión del modelo RNN durante la fase de entrenamiento.

## Modelo LSTM

Este modelo ha sido entrenado en 10 *epochs*, con un tiempo de entrenamiento total de 5843,35s. La precisión obtenida en este proceso ha sido de 93,51 % y 85,442 % para los conjuntos de entrenamiento y validación, respectivamente. Los resultados del entrenamiento se muestran en la figura 4.8.

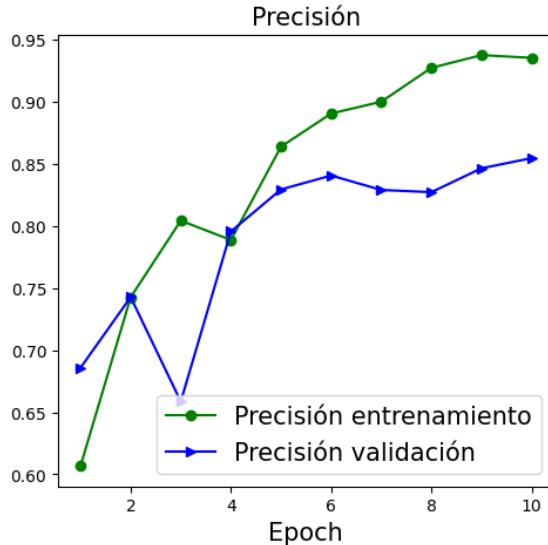


Figura 4.8: Precisión del modelo LSTM durante la fase de entrenamiento.

## Modelo GRU

Este modelo ha sido entrenado en 10 *epochs*, con un tiempo de entrenamiento total de 4692,81s. La precisión obtenida en este proceso ha sido de 97,17 % y 86,34 % para los conjuntos de entrenamiento y validación, respectivamente. En la figura 4.9, se muestran los resultados de entrenamiento con estos parámetros.

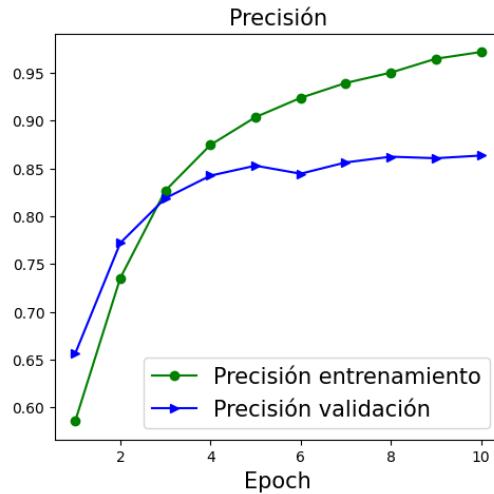


Figura 4.9: Precisión del modelo GRU durante la fase de entrenamiento.

#### 4.5.4. Resultados y conclusiones

##### Modelo RNN

Como se muestra en la figura 4.10, la precisión en el conjunto de test del modelo RNN ha sido del 76,57 %, que no sería aceptable para un modelo predictivo en producción.

```
Epoch 0 accuracy: 0.5502 val_accuracy: 0.5682
Epoch 1 accuracy: 0.6385 val_accuracy: 0.6544
Epoch 2 accuracy: 0.6414 val_accuracy: 0.6092
Epoch 3 accuracy: 0.6852 val_accuracy: 0.6736
Epoch 4 accuracy: 0.7510 val_accuracy: 0.7086
Epoch 5 accuracy: 0.7819 val_accuracy: 0.7230
Epoch 6 accuracy: 0.8092 val_accuracy: 0.7456
Epoch 7 accuracy: 0.6948 val_accuracy: 0.6468
Epoch 8 accuracy: 0.7395 val_accuracy: 0.7024
Epoch 9 accuracy: 0.8279 val_accuracy: 0.7622
Epoch 10 accuracy: 0.8296 val_accuracy: 0.6758
Epoch 11 accuracy: 0.8091 val_accuracy: 0.7428
Epoch 12 accuracy: 0.8462 val_accuracy: 0.7530
Epoch 13 accuracy: 0.8764 val_accuracy: 0.7574
Epoch 14 accuracy: 0.8780 val_accuracy: 0.7642
test_accuracy: 0.7657
Tiempo de entrenamiento: 2893.1333107500004
```

Figura 4.10: Precisión de entrenamiento, validación y test del modelo RNN.

##### Modelo LSTM

Como se muestra en la figura 4.11, la precisión en el conjunto de test del modelo LSTM ha sido del 83,07 %, que representa una mejora del 8,49 % con respecto al modelo RNN, pero con un incremento en tiempo de entrenamiento del 102 %.

```
Epoch 0 accuracy: 0.6069 val_accuracy: 0.6858
Epoch 1 accuracy: 0.7425 val_accuracy: 0.7426
Epoch 2 accuracy: 0.8041 val_accuracy: 0.6586
Epoch 3 accuracy: 0.7885 val_accuracy: 0.7954
Epoch 4 accuracy: 0.8636 val_accuracy: 0.8292
Epoch 5 accuracy: 0.8902 val_accuracy: 0.8404
Epoch 6 accuracy: 0.8998 val_accuracy: 0.8288
Epoch 7 accuracy: 0.9270 val_accuracy: 0.8270
Epoch 8 accuracy: 0.9373 val_accuracy: 0.8462
Epoch 9 accuracy: 0.9351 val_accuracy: 0.8544
test_accuracy: 0.8307
Tiempo de entrenamiento: 5843.345772860001
```

Figura 4.11: Precisión de entrenamiento, validación y test del modelo LSTM.

##### Modelo GRU

Como se muestra en la figura 4.12, la precisión en el conjunto de test del modelo GRU ha sido del 85,75 %, que representa una mejora del 3,23 % con respecto al modelo LSTM, pero con una disminución en tiempo de entrenamiento del 20 %. Esto es una prueba empírica del desarrollo teórico de este capítulo sobre la mejoría del rendimiento de las celdas GRUs sobre las LSTMs.

```
Epoch 0 accuracy: 0.5858 val_accuracy: 0.6564
Epoch 1 accuracy: 0.7353 val_accuracy: 0.7720
Epoch 2 accuracy: 0.8267 val_accuracy: 0.8188
Epoch 3 accuracy: 0.8746 val_accuracy: 0.8424
Epoch 4 accuracy: 0.9035 val_accuracy: 0.8528
Epoch 5 accuracy: 0.9237 val_accuracy: 0.8444
Epoch 6 accuracy: 0.9393 val_accuracy: 0.8562
Epoch 7 accuracy: 0.9501 val_accuracy: 0.8622
Epoch 8 accuracy: 0.9646 val_accuracy: 0.8606
Epoch 9 accuracy: 0.9717 val_accuracy: 0.8634
test_accuracy: 0.8575
Tiempo de entrenamiento: 4692.805375315
```

Figura 4.12: Precisión de entrenamiento, validación y test del modelo GRU.

# Capítulo 5

## *Transformers: La clave del procesamiento del lenguaje natural*

En 2017, un grupo de científicos de Google [14], encabezado por Ashish Vaswani, decide prescindir de capas recurrentes y convolucionales en el procesamiento del lenguaje natural y centrar sus esfuerzos en desarrollar mecanismos de atención. Así, descubrieron el *transformer*, una arquitectura completamente innovadora, que se encuentra tras la revolución de IA que vivimos actualmente.

Este concepto contraintuitivo, que nació con la intención de mejorar la precisión de los traductores de texto, fue aplicado por primera vez en 2018 por OpenAI [10] para predecir texto, a través de su primer LLM *GPT*. Los sorprendentes resultados que obtuvieron desencadenaron una ola de LLMs, de la que han nacido nuevos modelos como *GPT4* de OpenAI y *Gemini* de Google.

Al eliminar las capas recurrentes, evitamos uno de los principales inconvenientes de estas, la dependencia entre estados. Esto hacía imposible la paralelización, lo que dificultaba el entrenamiento de este tipo de redes. Además, las RNN tenían memoria a *corto plazo*, mientras que los *transformers* son capaces de identificar la correlación entre las distintas palabras del texto y por tanto, podrán tener memoria mucho más largoplacista.

El objetivo de este capítulo es explicar el funcionamiento de este tipo de red neuronal, a través de un desarrollo teórico ayudado de ejemplos. Así, se estudiará la arquitectura general (5.1), el *encoder* (5.2), el *decoder* (5.3) y el funcionamiento de este durante la fase de entrenamiento (5.4) e inferencia (5.5). Por último, construiré y entrenaré varios *transformers* para traducir texto de inglés a español (5.6). El código con la arquitectura del transformer y el archivo con los entrenamientos de los modelos se encuentran en `transformers.py` y en `transformer_translation.ipynb`, ambos en el github del proyecto<sup>1</sup>.

### 5.1. Arquitectura

El *transformer* está compuesto por dos piezas fundamentales [14]:

- **Encoder:** procesa la secuencia de entrada para crear una representación rica en contexto de cada palabra o *token* de la secuencia. Esta representación captura no solo la información semántica de cada palabra, sino también cómo cada palabra se relaciona con todas las otras palabras del texto.
- **Decoder:** tiene la tarea de generar una secuencia de salida paso a paso, basándose en la representación contextualizada de la secuencia de entrada proporcionada por el *encoder*.

---

<sup>1</sup>Github del proyecto: <https://github.com/miguelangelmoralesramon/tfg.git>

A continuación, explicaré con más detalle esta arquitectura, que se muestra en la figura 5.1.

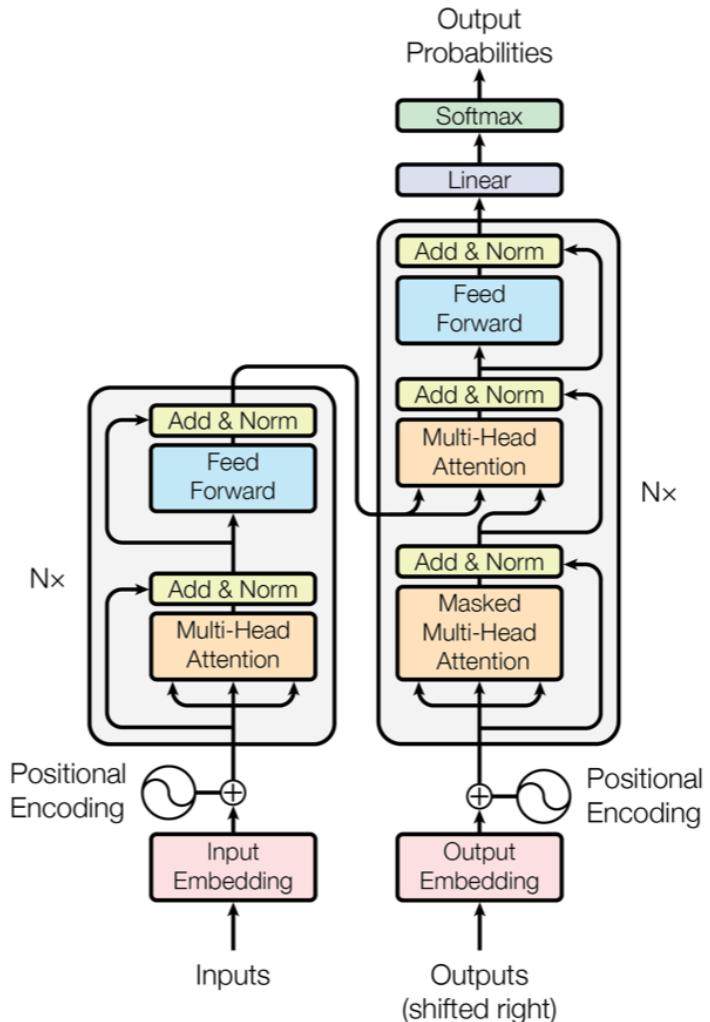


Figura 5.1: Arquitectura del transformer [14].

### 5.1.1. Embeddings y codificación posicional

Para poder alimentar tanto el *encoder* como el *decoder*, es necesario traducir el texto a lenguaje numérico. Para ello, se aplica un ***embedding*** [9] que, en el contexto de NLP, es una representación vectorial de datos de mayor dimensión, como palabras, frases o incluso documentos enteros. La idea es capturar el significado semántico, la sintaxis, y las relaciones entre los elementos en un espacio vectorial más compacto. Sus principales características son:

- Reducen la dimensión de los datos textuales, facilitando el manejo y procesamiento de grandes volúmenes de texto..
- Pueden capturar relaciones semánticas entre palabras, como se muestra en la figura 5.2. Por ejemplo, en un espacio vectorial de *embeddings*, las palabras que son similares en significado

(como “perro” y “gato”) estarán más cerca entre sí en comparación con palabras que no están relacionadas (como “perro” y “tobogán”).

- Al convertir texto en vectores numéricos, permiten que los modelos de aprendizaje automático trabajen directamente con datos textuales. Esto es esencial para tareas como clasificación de texto, análisis de sentimientos o traducción automática
- Existen *embeddings* pre-entrenados, como *word2vec* [11], *GloVe* o *BERT*, que permiten aprovechar el conocimiento adquirido a partir de grandes corpus de texto y aplicarlo a tareas específicas.

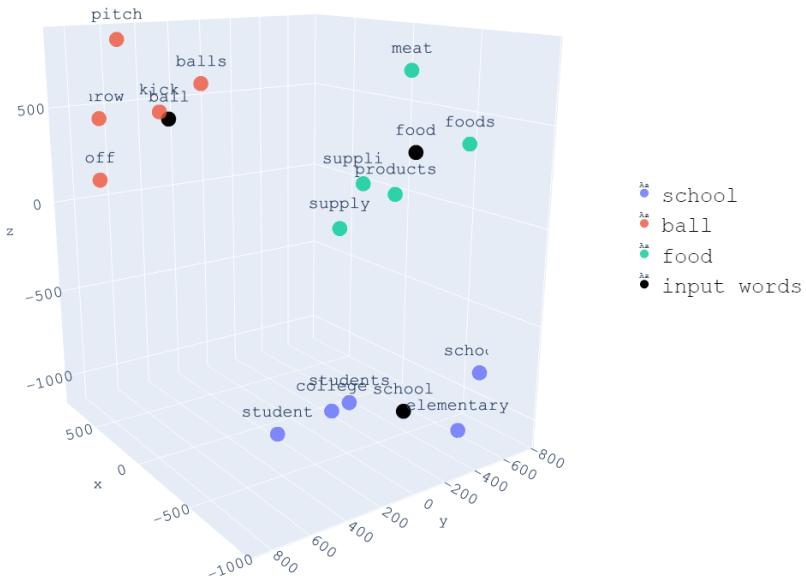


Figura 5.2: Espacio vectorial de *embeddings* de palabras relacionadas con *school*, *food* y *ball*.

**Definición 5** (Embedding). *Dado un conjunto  $\mathbf{V}$ , denominado **vocabulario**, con  $\|\mathbf{V}\| = n$  , un embedding de dimensión  $d$  es una aplicación inyectiva  $f : \mathbf{V} \rightarrow \mathbb{R}^d$ , tal que  $d \ll n$ .*

No entraremos más en detalle en el funcionamiento de los algoritmos y modelos que calculan estos *embeddings*, pues queda fuera del objetivo de este trabajo.

El *embedding* que utilizaremos será el *FastText*, que es una extensión del *embedding word2vec* utilizado en el paper de *Google* [14], para representar las palabras de nuestro vocabulario (en este caso el diccionario inglés y español), en vectores de dimensión 512. Este *embedding* se puede calcular haciendo uso del paquete `torch.nn.Embedding` de la librería *pytorch*.

Tras calcular el *embedding*, se aplica una **codificación posicional** o *positional encoding* para almacenar en los vectores de cada palabra información sobre la posición de dicha palabra en el texto. Estos vectores de posición contienen patrones que el modelo puede aprender a asociar con ciertas distancias o posiciones relativas entre *tokens* en la secuencia.

**Definición 6** (Codificación posicional). *La codificación posicional, PE, es una operación sobre las componentes de un vector, que viene dada por las siguientes ecuaciones, donde pos es la posición*

del token en la secuencia,  $i$  es la coordenada en el vector de embedding y  $d$  es la dimensión total de los embeddings:  $PE_{(pos,2i)} = \sin(\frac{pos}{10000^{2i/d}})$

$$PE_{(pos,2i+1)} = \cos(\frac{pos}{10000^{2i+1/d}})$$

De esta forma, el vector de codificación posicional viene dado por:

$$PE := (PE_{(pos,1)}, PE_{(pos,2)}, \dots, PE_{(pos,d)})$$

Y el resultado,  $I$ , de aplicar la codificación posicional al vector de embeddings,  $E \in \mathbb{R}^d$  es:  
 $I := E + PE$

Con esto, ya tenemos nuestra secuencia lista para ser procesada por el transformer. Como podemos observar en la figura, ambas componentes tienen subcapas con operaciones en común, que detallaremos a continuación.

### 5.1.2. Producto escalar escalado y multi-head attention

Las operación más importante del transformer se denomina *multi-head attention* y para entenderla, es necesario primero definir el producto escalar escalado. Para realizar estas operaciones necesitaremos tres matrices cuyos pesos son entrenables,  $Q$  (consulta o *query*),  $K$  (clave o *key*) y  $V$  (valor o *value*) que servirán para extraer distintas características sobre el texto de entrada y las relaciones entre sus palabras. La matriz consulta ( $Q$ ) recogerá la información que busca cada elemento, la matriz clave ( $K$ ) representará la esencia o resumen de lo que ofrece cada elemento y la matriz valor ( $V$ ) recopilará la información que verdaderamente contiene cada elemento. En la teoría,  $Q, K \in Mat_{n \times d_k}$  y  $V \in Mat_{n \times d_v}$ , donde  $n$ ,  $d_k$ ,  $d_v$  son las dimensiones de la secuencia de entrada, matriz consulta y matriz valor, respectivamente. Sin embargo, en la práctica para simplificar el modelo se suele tomar  $d_k = d_v$ . A  $d_k$  se le asigna la dimensión del *embedding*.

Cada una de estas matrices se calcula a partir de valor de entrada del *encoder* o del *decoder*. Esto se realiza a través de una red neuronal artificial de una única capa densa de  $2d_k + d_v$  neuronas. Por tanto, cada matriz tiene asociada una matriz de pesos  $W$ . Denotaremos como  $W^Q$ ,  $W^K$  y  $W^V$  a las matrices de pesos asociadas a las matrices  $Q$ ,  $K$  y  $V$ , respectivamente. La representación visual del producto escalar escalado aparece en la figura 5.4.

**Definición 7** (Producto escalar escalado). *Dadas las matrices  $Q, K, V \in Mat_{n \times d_k}$ , se denomina matriz de atención a:*

$$A(Q, K) := \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)$$

Así, se define el producto escalar escalado como:

$$\text{Atencion}(Q, K, V) := A(Q, K)V$$

$\sqrt{d_k}$  se llama factor de escala y nos ayuda a controlar la varianza de los productos escalares, lo que hace el proceso de entrenamiento más estable.

Cada elemento  $(i, j)$  de la matriz de atención indica la correlación entre la palabra en la posición  $i$  de la primera secuencia y la de la posición  $j$  de la segunda secuencia. En el *encoder*, este cálculo se realiza enfrentando cada secuencia consigo misma para obtener las correlaciones entre las distintas palabras de la misma frase, lo que se conoce como **auto-atención**. En el *decoder*, veremos que este cálculo se computa entre la secuencia de entrada y la predicción. De esta forma, si calculamos la matriz de auto-atención para la secuencia “Mi nombre es Miguel”, obtendremos una matriz de tamaño  $4 \times 4$ . Esta matriz tendrá valores muy cercanos a 1 en la diagonal, pues representan las correlaciones de cada palabra consigo misma. Como se muestra en la figura 5.3, el elemento  $(3, 7)$  de la matriz tendrá también un valor cercano a 1 pues la palabra “nombre” se refiere a “Miguel”.

	Mi	nombre	es	Miguel
Mi	1.0	0.5	0.1	0.25
nombre	0.4	1.0	0.2	0.75
es	0.1	0.4	1.0	0.3
Miguel	0.2	0.65	0.35	1.0

Figura 5.3: Matriz de auto-atención para la secuencia “Mi nombre es Miguel” (elaboración propia).

Además, podemos incluir una **máscara** antes de aplicar *softmax*, para no recopilar información futura. Esta máscara, convierte todos los valores  $\{(i, j) \in A | i < j\}$  de la matriz de atención en  $-\infty$ . La motivación de esto, es que, al aplicar la función *softmax*, los valores de la diagonal superior, que son  $-\infty$ , se conviertan todos en 0. De esta forma, ninguna palabra tendrá información sobre otra que se encuentre después, lo que será útil a la hora de realizar predicciones en el *decoder*.

**Definición 8** (Producto escalar escalado enmascarado). *Una máscara es una función, que dada una matriz  $M \in \text{Mat}$  sustituye los elementos de la diagonal superior en  $-\infty$ .*

$$\text{Atencion}_{mask} = \text{softmax}\left(\text{mask}\left(\frac{QK^T}{\sqrt{d_k}}\right)\right)V$$

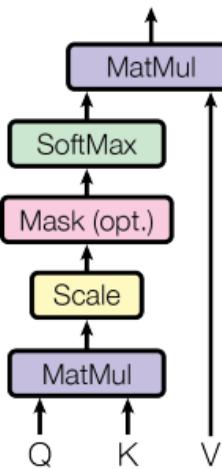


Figura 5.4: Producto escalar escalado [14].

La *multi-head attention* es un mecanismo que amplía la idea de atención del producto escalar escalado, permitiendo que el modelo atienda conjuntamente a la información de diferentes subespacios de representación en diferentes posiciones. Al hacer esto, aumenta la capacidad del modelo para centrarse en diferentes partes de la secuencia de entrada, lo que mejora su capacidad de comprensión de las relaciones entre los elementos de la secuencia de entrada.

La motivación tras este algoritmo es ejecutar en paralelo distintos productos escalares escalados para obtener una mejor representación de la secuencia de entrada. Al igual que pasaba con los filtros de las redes convolucionales, cuando aplicamos un mecanismo de atención a través del producto escalar escalado, este tiende a desarrollar un sesgo hacia una característica en concreto de la secuencia. Para computar la atención *multi-head* primero debemos establecer el número de cabezales de nuestro algoritmo, digamos  $h$ . Entonces, se proyectan linealmente las consultas, claves y valores  $h$  veces, para lo que se utiliza una capa *feedforward* lineal. Por cada una de las  $h$  proyecciones, se aplica en paralelo la atención del producto escalar escalado, lo que resulta en valores de salida  $d_v$ -dimensionales. Estas salidas se concatenan y se transforman linealmente para reducir la dimensión, obteniendo finalmente la dimensión requerida. En la figura 5.5 se ilustra este proceso.

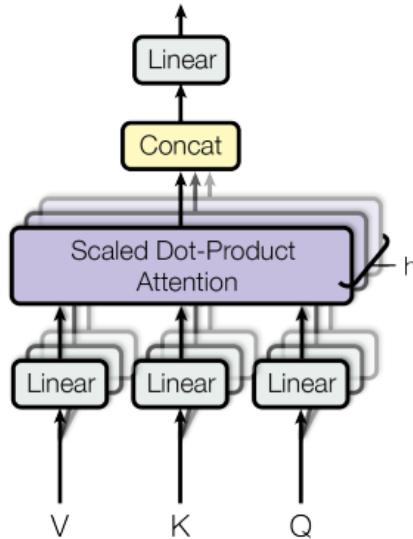


Figura 5.5: *Multi-head attention* [14].

**Definición 9** (*Multi-head attention*). *Dado un número de cabezas  $h$ , para ejecutar el algoritmo de atención multicabezal procedemos de la siguiente forma:*

1. *Proyección.* Para cada cabeza  $i \in [1, h]$ , obtenemos las matrices  $Q_i, K_i, V_i$  de la siguiente forma:

$$Q_i = QW_i^Q, \quad K_i = KW_i^K, \quad V_i = VW_i^V$$

donde  $W_i^Q, W_i^K \in \mathbb{R}^{d_{model} \times d_k}$  y  $W_i^V \in \mathbb{R}^{d_{model} \times d_v}$  son las matrices de pesos correspondientes a las matrices de consulta, clave y valor.

2. *Producto escalar escalado.* Aplicamos el mecanismo de atención del producto escalar escalado para cada cabeza:

$$\text{Head}_i = \text{Atencion}(Q_i, K_i, V_i) = \text{softmax}\left(\frac{Q_i K_i^T}{\sqrt{d_k}}\right) V_i$$

3. *Concatenación.* Concatenamos las salidas de cada una de las cabezas:

$$C = \text{Concat}(\text{Head}_1, \text{Head}_2, \dots, \text{Head}_h)$$

4. *Proyección final.* Aplicamos una proyección lineal para reducir la dimensión de los datos y que estos se ajusten a la dimensión del modelo:

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{Head}_1, \text{Head}_2, \dots, \text{Head}_h) \cdot W^O$$

donde  $W^O \in \mathbb{R}^{hd_v \times d_{model}}$  es la matriz de pesos de la transformación lineal.

La última operación que necesitamos, para entender la estructura del transformer es la normalización por capas, explicada en el capítulo 1.

## 5.2. Encoder

El objetivo del *encoder* es obtener una representación en forma de matriz, de las relaciones entre las palabras de la secuencia de entrada. Estos valores permiten que la red tenga un contexto sobre el texto y sea capaz de entender la secuencia, aunque solo haya visto una representación numérica de esta. Dichos datos serán utilizados por el *decoder* para completar la tarea para la que se haya construido (traducción, completar una frase...).

Suponiendo que ya tenemos una secuencia de entrada,  $I \in \text{Mat}_{n \times d_{model}}$ , a la que se le ha aplicado el *embedding* y la codificación posicional, podemos resumir el funcionamiento del *encoder* en los siguientes pasos:

- Atención *multi-head*. Dada la entrada  $I$ , calculamos las matrices  $Q, K$  y  $V$  y obtenemos la matriz de atención *multi-head*  $M$ :

$$M = \text{MultiHead}(Q, K, V)$$

- Suma residual y normalización. Para evitar el problema del devanecimiento del gradiente, al resultado del paso anterior ( $M$ ) sumamos de nuevo la entrada del *decoder* ( $I$ ). Esto es lo que se conoce como **suma residual**:  $S = M + I$

A continuación, le aplicamos una normalización por capas, que es un método de regularización estudiado en el segundo capítulo:  $N = \text{LayerNorm}(S)$

- Capa *feed-forward*. A la salida del paso anterior primero le aplicamos una capa densa de dimensión  $d_f$ , que normalmente es mayor que  $d_{model}$ . Además, esta capa tendrá una función de activación no lineal (en nuestro caso, ReLU). Esto permite al modelo obtener relaciones más complejas y no lineales entre los datos:  $F_1 = \text{FeedForward}(N)$

Seguidamente, aplicamos otra capa densa de dimensión  $d_{model}$  para devolver los datos a su dimensión original. Es decir:  $F_2 = \text{FeedForward}(F_1)$

- Suma residual y normalización. Al igual que en el paso 2, a la salida del paso anterior se le aplica la suma residual y la normalización para obtener la salida final del *encoder*  $O_E$ :

$$S = F_2 + N, O_E = \text{LayerNorm}(S)$$

Como ejemplo, vamos a procesar frases en inglés y español para crear un traductor de español a inglés. Empezamos obteniendo la matriz de atención con un *encoder*. Lo haremos en lotes de 30 secuencias, cada una contendrá una frase de longitud máxima  $n = 50$  palabras, lo que resulta en 30 vectores de tamaño  $n$ . Si la frase tiene longitud menor que  $n$ , los espacios en blanco los rellenaremos con un **token auxiliar de relleno** (en el código “`<PAD>`”). Ahora debemos de aplicar el *embedding* que en este caso será de tamaño 512 ( $d_{model} = 512$ ), lo que aumentará la dimensión de nuestra matriz de  $30 \times n$  a  $30 \times n \times 512$ . A esto le sumamos la codificación posicional y obtenemos la matriz de entrada del *encoder*  $I \in \text{Mat}_{30 \times n \times 512}$ . Supongamos que el

primer par es  $\{x_1 = \text{"me llamo Miguel"}, y_1 = \text{"my name is Miguel"}\}$ , por lo que la secuencia de entrada será  $x_1$ . Para computar el *embedding* es necesario definir por completo el vocabulario sobre el que estamos trabajando, lo cual en este ejemplo teórico no es un problema, pero al final del capítulo veremos que deberemos de cambiar este enfoque. Para este ejemplo, la codificación y el procesamiento de las secuencias será palabra a palabra. El resultado de aplicarle el *embedding* y la codificación posicional a la variable  $x_1$  sería una matriz de tamaño  $50 \times 512$ . Este proceso lo realizamos simultáneamente con las 30 secuencias del lote.

Ahora, ejecutaremos el mecanismo de atención *multi-head*, con 8 cabezas. En lugar de crear 8 tríos de matrices  $Q_i, K_i$  y  $V_i$  (para lo que se necesitaría una capa densa de tamaño de entrada 512 y tamaño de salida  $512 \cdot 3 \cdot 8 = 12228$ ), lo que haremos será generar únicamente un trío de matrices (a través de una capa densa de tamaño de entrada 512 y tamaño de salida  $512 \cdot 3 = 1536$ ) y dividir cada una en 8 trozos, correspondientes a cada una de las cabezas. Este proceso resulta en una matriz  $P \in Mat_{30 \times 50 \times 1536}$ . De la última dimensión de la matriz, las primeras 512 columnas corresponden a la matriz consulta, las segundas 512, a la matriz clave y las últimas a la matriz valor.



Figura 5.6: Diagrama de la atención *multi-head* (elaboración propia).

Sobre esta matriz, ejecutamos en paralelo los 8 procesos correspondientes a cada uno de los cabezales. Para esto, dividimos cada submatriz  $Q, K, V$  en 8 partes  $Q_i, K_i, V_i$ , lo que resulta en una matriz  $P' \in Mat_{30 \times 50 \times 8 \times 64}$ . Ahora, para cada cabezal  $i \in 1, 2, \dots, 8$  calculamos el producto escalar escalado:

$$H_i := Atencion(Q_i, K_i, V_i) \in Mat_{30 \times 50 \times 64}$$

Para obtener el resultado final del algoritmo de *multi-head attention* simplemente tenemos que concatenar cada una de los cabezales y devolver la matriz al tamaño del modelo original. Como en este caso hemos dividido cada matriz  $Q, K, V$  en lugar de aumentar la dimensión del modelo, no hará falta aplicar ninguna transformación.

$$H := Concat(H_1, H_2, \dots, H_8) \in Mat_{30 \times 300 \times 512}$$

Obtenemos  $H' \in Mat_{30 \times 50 \times 512}$  tras aplicarle la suma residual y la normalización a  $H$  y procedemos a aumentar la dimensión a través de una capa *feed-forward* (con función de activación ReLU) a  $F_1 = FeedForward(H') \in Mat_{30 \times 50 \times 2048}$ . Aplicamos la segunda capa *feed-forward* para obtener de vuelta una matriz de la dimensión del modelo  $F = FeedForward(F_1) \in Mat_{30 \times 50 \times 512}$ . Por último, calculamos la suma residual y normalizamos y obtenemos la salida del *encoder*  $O_E$ .

$$O_E = LayerNorm(F + H') \in Mat_{30 \times 50 \times 512}$$

### 5.3. Decoder

A diferencia del *encoder*, que procesa cada secuencia de una sola vez, el *decoder* utilizará la matriz de atención generada por el *encoder* para predecir *token* a *token* la secuencia de salida. El *decoder* se denomina como un modelo **auto-regresivo** porque, durante la inferencia, su entrada en el paso  $t$  utiliza su salida en el paso  $t - 1$ . Para cada secuencia, añadiremos al principio un ***token de comienzo*** (que definiremos en el código como "`< SOE >`") y el ***token final*** ("`< EOE >`" en el código) marcará el final de la predicción. Si la longitud de la secuencia es menor que la dimensión del modelo, utilizaremos un *token* de relleno para completar ("`< PAD >`"). Lo que contenga la

secuencia de entrada tras el *token* de comienzo, dependerá de si estamos entrenando al transformer o si, este ya está entrenado y estamos en la fase de inferencia. Para estudiar el *decoder*, vamos a obviar este aspecto de momento y vamos a tratar la secuencia de entrada como un conjunto de *tokens* al que le añadimos los *tokens* auxiliares y le aplicamos un *embedding*, obteniendo la matriz  $I \in Mat_{batch \times n \times d_{model}}$

- Atención auto-regresiva. Dada la secuencia  $I \in Mat_{batch \times n \times d_{model}}$ , correspondiente al resultado de aplicar el *embedding* a la secuencia de entrada, le aplicamos el algoritmo de atención *multi-head* enmascarado. Este algoritmo en lugar de aplicar el producto escalar escalado, aplica el producto escalar escalado enmascarado. Como vimos al comienzo del capítulo, este último, aplica un filtro al calcular la atención con el objetivo de que la matriz resultante tenga la diagonal superior nula. Por tanto, cada *token* no tendrá información sobre *tokens* que se encuentren en una posición más avanzada en la secuencia.

$$H_{mask} = MultiHead_{mask}(Q, K, V) = Concat(H_1, \dots, H_h) \cdot W^O$$

$$\text{donde } H_i = Atencion_{mask}(Q_i, K_i, V_i)$$

Al resultado de la atención *multi-head* enmascarada  $H$ , le aplicamos la suma residual y normalización por capas, obteniendo  $H_1 = LayerNorm(H + I)$

- Atención *encoder-decoder*. Ahora entran en juego los cálculos realizados en el *encoder*. Utilizaremos un capa *feed-forward* para generar una matriz clave  $K^E$  y una matriz valor  $V^E$ , a partir de la salida del *encoder*. Dicho de otra forma, tomaremos el *output* del *encoder*,  $O_E$ , y le aplicaremos una capa neuronal *feed-forward* para generar dos matrices,  $K^E, V^E$  que tendrán cada una la misma dimensión que la matriz original ( $O_E$ ). Estas matrices recopilarán todo el contexto de la secuencia, lo que permitirá al *decoder* extraer la información relevante de la secuencia de entrada para generar el siguiente *token*. Utilizaremos estas matrices, junto con la matriz consulta  $Q$ , extraída de  $H_1$ , para calcular la atención *multi-head*.

$$H_E = MultiHead(Q, K^E, V^E)$$

A continuación, sumamos el residuo y normalizamos para obtener  $H_2$ :

$$H_2 = LayerNorm(H_E + H_1)$$

- Capa *Feed-Forward*. De forma similar al *encoder*, aplicamos una capa densa que mantiene la dimensión de la matriz de entrada para refinar la predicción y obtenemos  $F = FeedForward(H_2)$ .
- Generación del siguiente *token*. Por último, aplicamos a  $F$  una transformación lineal para proyectar el resultado sobre el espacio  $\mathbf{V}_f$  que denominamos vocabulario final. Seguidamente, aplicamos al resultado la función *softmax* para obtener la probabilidad sobre cada *token* de salida. El *token* que obtenga la mayor probabilidad será el que seleccionemos en la predicción final.

## 5.4. Entrenamiento

En la fase de entrenamiento, se alimenta al *decoder* con las secuencias a predecir ( $y_i$ ) con los *tokens* auxiliares. Es decir, para el caso  $\{x_1 = \text{"me llamo Miguel"}, y_1 = \text{"my name is Miguel"}\}$ , tomariamos  $v_1 = [< SOE >, \text{"my"}, \text{"name"}, \text{"is"}, \text{"Miguel"}, < PAD >, \dots, < PAD >]$ , con  $longitud(v_1) = n = 30$ . Es importante el detalle de añadir el *token* de comienzo a la secuencia  $v_1$ , ya que desplaza la secuencia 1 *token* a la derecha. Como en el *decoder* se aplica una máscara, esto hará que la predicción del transformer en la posición  $i$  solo tenga información sobre la variable objetivo hasta la posición  $i - 1$  de esta. Es decir, si la predicción del *transformer* en fase de entrenamiento es  $o_1 = [\text{"I"}, \text{"like"}, \text{"football"}, < EOE >, < PAD >, \dots, < PAD >]$ , el *token* “I“ se ha generado únicamente teniendo constancia del token de comienzo [ $< SOE >$ ], “like“ se ha generado

con el contexto [*< SOE >*, “*my*“], “*football*“ con [*< SOE >*, “*my*“, “*name*“] y *< EOE >* con [*< SOE >*, “*my*“, “*name*“, “*is*“].

A continuación, a la secuencia  $v_1$  le aplicamos el *embedding* y la codificación posicional, obteniendo el resultado de la figura. Esto lo realizamos con las 30 secuencias del lote y sacamos  $I \in Mat_{30 \times 50 \times 512}$ .

Ahora, pasamos esta matriz por la capa de atención *multi-head* enmascarada (de 8 cabezales), al igual que en el *encoder*, en lugar de generar 8 tríos de matrices  $Q, K, V$ , generaremos un único trío y dividiremos cada matriz en 8 submatrices. Con estas matrices utilizaremos el producto escalar escalado enmascarado, por lo que la matriz de atención  $A_i = Atencion_{mask}(Q_i, K_i, V_i) \in Mat_{30 \times 50 \times 50}$  tendrá la diagonal superior nula. Al trabajar con esta matriz, conseguimos que cada *token* tenga únicamente información sobre los *tokens* que le preceden.

El resultado de la atención *multi-head* enmascarada es  $H_{mask} \in Mat_{30 \times 50 \times 512}$  que normalizamos para obtener  $H_1 \in Mat_{30 \times 50 \times 512}$ . Seguidamente, aplicamos atención 8-cabezal con la matriz consulta  $Q$  del paso anterior y las matrices clave y valor del *encoder*  $K^E, V^E$ . Tras normalizar, obtenemos  $H_2 \in Mat_{30 \times 50 \times 512}$ . Aplicamos la capa *feed-forward*, de tamaño de entrada y tamaño de salida iguales a 512, con lo que tenemos la salida del *decoder*  $F = FeedForward(H_2) \in Mat_{30 \times 50 \times 512}$ .

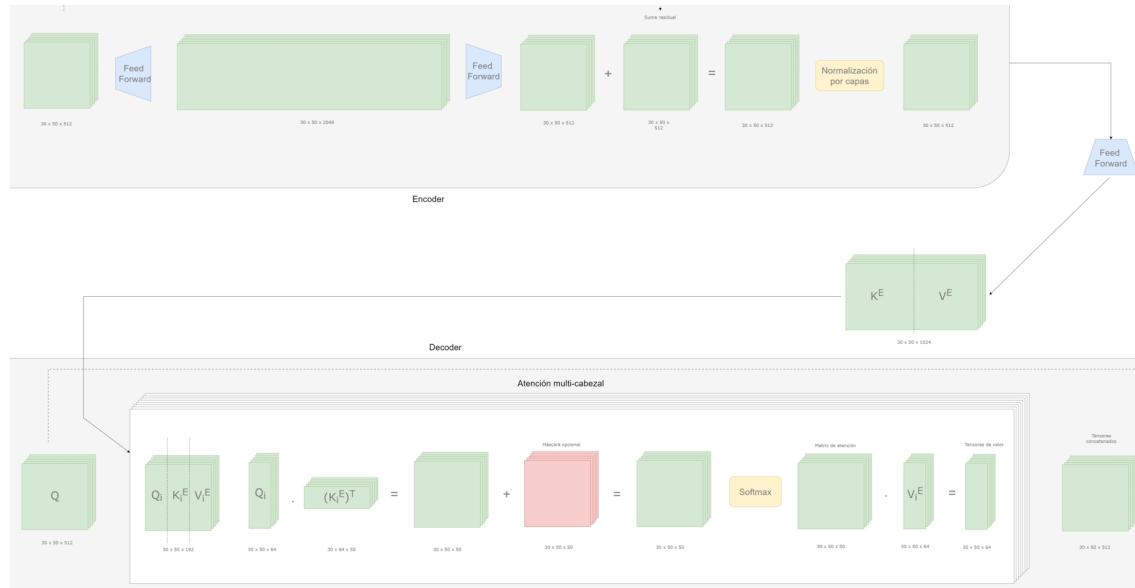


Figura 5.7: Mecanismo de atención *encoder-decoder* (elaboración propia).

Por último, a través de una transformación lineal a la dimensión del vocabulario inglés ( $\mathbf{V}_{eng}$ ) y la función *softmax*, sacamos la predicción final. Supongamos que para nuestro ejemplo, “*me llamo Miguel*“; estamos en la primera iteración de entrenamiento, por lo que nuestra red aún no ha sido ajustada, y el resultado del *decoder* es  $o_1 = [“I”, “like”, “football”, *< EOE >*, *< PAD >*, ..., *< PAD >*]$ . Entonces calcularíamos la pérdida para cada *token* con respecto a la variable objetivo  $y_1 = [“my”, “name”, “is”, “Miguel”, *< EOE >*, *< PAD >*, ..., *< PAD >*]$ . Esta pérdida se calcula entre el *embedding* de la frase predicha por el modelo y el *embedding* de la frase objetivo, haciendo uso de la función de pérdida *cross-entropy loss*. Por último, retro-propagaremos el error para ajustar la red. Este proceso lo repetiríamos con cada uno de los lotes, hasta alcanzar el número de *epochs* ( $n\_epochs$ ) determinado.

## 5.5. Inferencia

Cuando ya tenemos el modelo entrenado, a la hora de inferir las predicciones cambia el funcionamiento del *decoder*. Supongamos que hemos entrenado nuestra red con el dataset español-inglés y ahora queremos probarlo con la frase  $x_1 = "En un lugar de La Mancha, de cuyo nombre no quiero acordarme."$ , cuyo valor objetivo es  $y_1 = "Somewhere in La Mancha, in a place whose name I do not care to remember."$ .

En primer lugar, el *encoder* procesa  $x_1$  para obtener  $K^E, V^E$ , matrices clave y valor muy ricas en cuanto al contexto y las relaciones entre las palabras de  $x_1$ . Entonces, el *decoder* empezaría utilizando como entrada en el primer paso  $t = 0$  la secuencia  $s_0 = [<SOE>, <PAD>, \dots, <PAD>]$  y la procesaría teniendo en cuenta el contexto proporcionado por  $K^E, V^E$ , para obtener (si ha sido correctamente entrenado) como salida  $o_0 = ["Somewhere", <PAD>, \dots, <PAD>]$ . Esta predicción, es utilizada en el paso  $t = 1$  como entrada y este proceso se repite hasta que la predicción sea " $<EOE>$ " o se alcance la longitud de secuencia máxima ( $n$ ).

- $t = 1$ :

```
s_1=[<SOE>, "Somewhere", <PAD>, \dots, <PAD>]
o_1=["Somewhere", "in", <PAD>, \dots, <PAD>]
```

- $t = 2$ :

```
s_2=[<SOE>, "Somewhere", "in", <PAD>, \dots, <PAD>]
o_2=["Somewhere", "in", "La", <PAD>, \dots, <PAD>]
```

- $t = 3$ :

```
s_3=[<SOE>, "Somewhere", "in", "La", <PAD>, \dots, <PAD>]
o_3=["Somewhere", "in", "La", "Mancha", <PAD>, \dots, <PAD>]
```

- $t = 4$ :

```
s_4=[<SOE>, "Somewhere", "in", "La", "Mancha", <PAD>, \dots, <PAD>]
o_4=["Somewhere", "in", "La", "Mancha", "", "<PAD>", \dots, <PAD>]
.
.
.
```

- $t = 16$ :

```
s_{16}=[<SOE>, "Somewhere", "in", "La", "Mancha", "", "in", "a", "place",
"whose", "name", "I", "do", "not", "care", "to", "remember", \dots, <PAD>]
o_{16}=["Somewhere", "in", "La", "Mancha", "", "in", "a", "place", "whose",
"name", "I", "do", "not", "care", "to", "remember", <EOE>, <PAD>, \dots, <PAD>]
```

Como la frase de salida termina con el *token* " $<EOE>$ ", el modelo lo detecta y devuelve la frase traducida: "*Somewhere in La Mancha, in a place whose name I do not care to remember.*"

Como podemos observar, si el modelo se equivoca en alguna de las predicciones, este error se arrastra hasta el final, pudiendo resultar en una secuencia completamente equivocada. Por ello, es esencial que el modelo esté entrenado con una gran cantidad de datos y con la mayor capacidad de computación posible para alcanzar una gran precisión.

## 5.6. Traductor de inglés a español

Para finalizar este capítulo, pondré en práctica los conceptos teóricos vistos anteriormente y entrenaré un *transformer* cuyo objetivo será **traducir texto de inglés a español**. Para esta tarea, los recursos utilizados en capítulos anteriores (*Macbook Pro 13-inch, 2017, 2,3 GHz Dual-Core Intel Core i5*) eran insuficientes. Por ello, he tenido que hacer uso de *Google Colab*, una herramienta *cloud* diseñada para ciencia de datos, que permite hacer uso de GPUs de NVIDIA. En concreto, he hecho uso de una tarjeta gráfica *NVIDIA A100*. En total, he requerido de 500 *computing units*, que equivalen a 44 horas de uso de una GPU A100 con un coste de 55,95€.

Tanto el desarrollo del *transformer*, como el código utilizado para obtener los resultados se encuentra disponible en la carpeta `transformers` del repositorio de Github<sup>2</sup>. En el script `transformer.py` está el desarrollo de la arquitectura del *transformer* y en `transformer_training.ipynb` se encuentra el código utilizado para entrenar los modelos, haciendo uso de esta arquitectura. En `transformers_BLEU.ipynb`, se evalúa el rendimiento de cada *transformer*. En el archivo `README.md` están las instrucciones para utilizar cada uno de los modelos ya entrenados.

### 5.6.1. Dataset

El *dataset* elegido ha sido Helsinki-NLP/opus-100<sup>3</sup> de *Hugging Face*. Está compuesto por  $10^6$  pares de frases. Este dataset tiene 3 *splits*, de entrenamiento, validación y prueba, y viene como un diccionario, en el que cada elemento es un diccionario con una única clave *translation* y como valor correspondiente, un diccionario de la forma {“en”:“sentence in english”, “es”:“frase en español”} donde las frases tienen el mismo significado. Aplicaremos una transformación al conjunto de entrenamiento para obtener  $D = \{\{x_i, y_i\} | i \in \{1, \dots, 10^6\}\}$ , donde para cada par la primera componente ( $x_i$ ) está en español y la segunda ( $y_i$ ) es su equivalente en inglés.

Una vez obtenida esta parte del *dataset*, creo dos *arrays* correspondientes a la frases en español e inglés en minúsculas. A diferencia de el ejemplo enunciado en la sección anterior, los vocabularios de ambos idiomas serán listas de los caracteres más frecuentes, como se muestra en la figura 5.8. Esto se debe a que construir vocabularios de palabras de ambos idiomas es una tarea compleja por la gran cantidad de palabras que tiene un idioma. De esta forma, al considerar vocabularios de caracteres, los modelos que construiré harán **predicciones carácter a carácter**, en lugar de palabra a palabra.

---

<sup>2</sup>Github del proyecto: <https://github.com/miguelangelmoralesramon/tfg.git>

<sup>3</sup>Dataset utilizado: <https://huggingface.co/datasets/Helsinki-NLP/opus-100/viewer/en-es>

```

english_vocab = [START_TOKEN, ' ', '!', '"', '#', '$', '%', '&', "", '(', ')', '*', '+', ',', '-', '.', '/', 
                 '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
                 ':', '<', '=' , '>', '?', '@',
                 '[', '\\", ']', '^', '_',
                 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l',
                 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x',
                 'y', 'z',
                 '{', '|', '}', '~', PADDING_TOKEN, END_TOKEN]
spanish_vocab = [START_TOKEN, ' ', '!', '"', '#', '$', '%', '&', "", '(', ')', '*', '+', ',', '-', '.', '/',
                 '0', '1', '2', '3', '4', '5', '6', '7', '8', '9',
                 ':', '<', '=' , '>', '?', '@',
                 '[', '\\", ']', '^', '_',
                 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l',
                 'm', 'n', 'ñ', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x',
                 'y', 'z', 'á', 'é', 'í', 'ó', 'ú',
                 '{', '|', '}', '~', PADDING_TOKEN, END_TOKEN]

```

Figura 5.8: Vocabularios en inglés y español.

Seguidamente, calculo el percentil 97 de la longitud de las frases en el datasets, que resulta en 291,00. Viendo esto, establezco la longitud máxima de cada secuencia en  $n = 300$ . Para finalizar el tratamiento de los datos, filtro las frases de ambos *arrays* para quedarme con aquellas que cumplen con la longitud máxima establecida y cuyos caracteres están contemplados en los vocabularios.

### 5.6.2. Modelos

He contemplado 3 modelos distintos, los 2 primeros para explorar empíricamente cómo afectan distintas arquitecturas a la precisión del *transformer* y en el tercer modelo incorporo los hallazgos de los anteriores modelos para crear un modelo más preciso.

Todos los modelos tienen en común las siguientes variables:

- Tamaño del *embedding*:  $d_{model} = 512$
- Tamaño de la capa *feed-forward* del *encoder*:  $ffn = 2048$
- Probabilidad de *dropout*:  $p = 10\%$
- Longitud máxima de secuencia:  $n = 300$

Así, lo único que cambiará en cada modelo será el número de cabezales en el algoritmo de *multi-head attention* y el número de capas, es decir, de *encoders* y *decoders*.

#### *Transformer base*

He entrenado un modelo base, para comparar con las siguientes dos variaciones y evaluar qué contribuye más a mejorar el modelo, si aumentar el número de cabezales o aumentar el número de capas. Para establecer la arquitectura de este modelo, me he inspirado en el paper de *Google* [14] y lo he creado con 8 cabezales y 1 sola capa. Lo he entrenado con las  $2 \cdot 10^5$  primeras frases del *dataset*, con un tamaño de lote  $batch\_size = 50$ , 10 *epochs* y optimizador *Adam* con tasa de aprendizaje  $\eta = 10^{-4}$ . El tiempo de entrenamiento total fue de 3821,92 segundos.

El modelo está disponible en el archivo `transformer_base` del *OneDrive* del proyecto <sup>4</sup>.

<sup>4</sup> *OneDrive* del proyecto: [https://ucomplutense-my.sharepoint.com/:f/g/personal/mimora02\\_ucm\\_es/EpydMcjnsEJHkTRVycbAA8EBFRQMjzfQFhnJGbe5bDzXug](https://ucomplutense-my.sharepoint.com/:f/g/personal/mimora02_ucm_es/EpydMcjnsEJHkTRVycbAA8EBFRQMjzfQFhnJGbe5bDzXug)

## *Transformer 1*

Este modelo tiene una única capa, es decir, un solo *encoder-decoder* y 16 cabezales en el algoritmo *multi-head attention*. Para entrenarlo he utilizado las  $2 \cdot 10^5$  primeras frases del *dataset*, con un tamaño de lote *batch\_size = 50*, 10 *epochs* y optimizador *Adam* con tasa de aprendizaje  $\eta = 10^{-4}$ . El tiempo de entrenamiento total ha sido de 4332,8 segundos. A continuación se muestran los resultados de entranamiento de las últimas 500 iteraciones de la última *epoch*. El campo *Iteration* indica el número de iteración, *English* muestra la primera frase en inglés del lote, *Spanish Translation* es la traducción correcta, **Spanish Prediction** es la predicción del modelo y *Evaluation translation* es el resultado de traducir la frase *somewhere in la mancha, in a place whose name I do not care to remember*. (“en algún lugar de la mancha, de cuyo nombre no quiero acordarme.”).

Iteration 3400 :

English: zima, grunge, every alex rodriguez rookie card in demand, and a pager.  
really?

Spanish Translation: zima, grunge, todos los cromos de novato de alex rodríguez  
bajo demanda, y un buscador. ¡en serio?

Spanish Prediction: zima, grunge, todos los poede ae lesilo ee llejododiigee  
ayaje, ye anda, y un paecanor

Evaluation translation: ('algunos en la mancha, en un place que no hace carrera  
no estar recuerdo.<EOE>',)

---

Iteration 3500 :

English: we recently reconnected.

Spanish Translation: hemos vuelto a contactar recientemente.

Spanish Prediction: eemos reelto rlroneectom.recientemente.

Evaluation translation: ('algunos en la mancha, en un place que no hecho no hacer  
no estar recuerdo.<EOE>',)

---

Iteration 3600 :

English: i got a brother-in-law in l.a.

Spanish Translation: tenía un cuñado en l.a.

Spanish Prediction: tenga un porado en la...?....

Evaluation translation: ('algunos en la mancha, en un place que no hace en carrera  
no estar recuerdo.<EOE>',)

---

Iteration 3700 :

English: about a dozen people in brooklyn have fallen into unexplained comas.

Spanish Translation: alrededor de una docena de personas en brooklyn han caído en  
comas inexplicables.

Spanish Prediction: ab ededor ue un poeu a ee lersonas en lrooklyn has dosdi cn  
eomas.dntxplcadles

Evaluation translation: ('algunas en la mancha, en un place que no es me hacer no  
estoy no estoy no estoy no estoy no estoy no estos miembro.<EOE>',)

---

Iteration 3800 :

English: - nope. - four?

Spanish Translation: - cuatro?

Spanish Prediction: - joanro.

Evaluation translation: ('algunos en la mancha, en una place que no hacer no  
estaba a recuerdo a recuerdo.<EOE>',)

---

Iteration 3900 :

English: do you know what you have to do to get back?  
Spanish Translation: ¿sabes cómo hacemos para regresar?  
Spanish Prediction: ¿sabes qómo qacesos qara hegresar  
Evaluation translation: ('algunos en la mancha, en un place que no estoy no esta como no estar recuerdo.<EOE>',)

---

El modelo está disponible en el archivo `transformer1` del *OneDrive* del proyecto<sup>5</sup>.

### **Transformer 2**

Este segundo modelo tenía 8 cabezales en el algoritmo de *multi-head attention* y estaba compuesto por 2 capas, es decir, tenía 2 estructuras *encoder-decoder*. Para entrenarlo he utilizado las mismas  $2 \cdot 10^5$  frases que para el modelo anterior, con un tamaño de lote *batch\_size* = 50, 10 *epochs* y optimizador *Adam* con tasa de aprendizaje  $\eta = 10^{-4}$ . El tiempo de entrenamiento ha sido de 6634,13 segundos. Los resultados de las últimas 500 iteraciones de la *epoch* final son los siguientes:

Iteration 3400 :

English: zima, grunge, every alex rodriguez rookie card in demand, and a pager.  
really?

Spanish Translation: zima, grunge, todos los cromos de novato de alex rodríguez  
bajo demanda, y un buscador. ¿en serio?

Spanish Prediction: zima, grunge, codos los roodes re riritosde rlex rooiígeez  
eajo ee anda, y un paecdador

Evaluation translation: ('alguna manza en un lugar en un lugar que no me importa  
a la lugar a la cuidada de la lugar de la lugar de la luz.<EOE>',)

---

Iteration 3500 :

English: we recently reconnected.

Spanish Translation: hemos vuelto a contactar recientemente.

Spanish Prediction: remos reelto r roneictad.eeconnte.ente.

Evaluation translation: ('algún en un lugar en un lugar que no tengo que nombre  
no cuidar a recordar.<EOE>',)

---

Iteration 3600 :

English: i got a brother-in-law in l.a.

Spanish Translation: tenía un cuñado en l.a.

Spanish Prediction: tenga un hoaaloren laa.....

Evaluation translation: ('algún mancha, en un lugar que nombre que no me importa  
que no recordaré a recordar a recordar a la carrera de la lugar de la lugar  
de la lugar de la lugar de la lugar de la lugar de la lugar de la lugar de  
la lugar de la lugar de la lugar de la lugar de la lugar de la lugar.<EOE>',)

---

Iteration 3700 :

English: about a dozen people in brooklyn have fallen into unexplained comas.

Spanish Translation: alrededor de una docena de personas en brooklyn han caído  
en comas inexplicables.

Spanish Prediction: sc ed dor ue un gosena ee lersonas hn broollyn han eordo en  
eomas enexplicadles.

Evaluation translation: ('algunas en la mancha, en un lugar que nombre que  
nombre nombre.<EOE>',)

<sup>5</sup>OneDrive del proyecto: [https://ucomplutense-my.sharepoint.com/:f/g/personal/mimora02\\_ucm\\_es/EpydMcjnsEJHkTRVycbAA8EBFRQMjzfQFhnJGbe5bDzXug](https://ucomplutense-my.sharepoint.com/:f/g/personal/mimora02_ucm_es/EpydMcjnsEJHkTRVycbAA8EBFRQMjzfQFhnJGbe5bDzXug)

```
-----  
Iteration 3800 :  
English: - nope. - four?  
Spanish Translation: - cuatro?  
Spanish Prediction: - cuatro.  
Evaluation translation: ('algún mancha, en un lugar en un lugar que no llamé no  
cuidaré a recordar.<EOE>',)  
-----
```

```
Iteration 3900 :  
English: do you know what you have to do to get back?  
Spanish Translation: ¿sabes cómo hacemos para regresar?  
Spanish Prediction: ¿sabes qomo qaceros qara qegresart  
Evaluation translation: ('algún mancha, en un lugar en un lugar que nombre que  
no me importa a recordar.<EOE>',)  
-----
```

El modelo está disponible en el archivo `transformer2` del *OneDrive* del proyecto<sup>6</sup>.

### Comparativa *transformer 1 vs transformer 2*

Tras entrenar el modelo base y las dos variaciones, he aplicado la *score* de *sacreBLEU*<sup>7</sup>, que es la métrica más utilizada para evaluar la calidad del texto que ha sido traducido con algoritmos de *machine learning*. Fue una de las primeras métricas en lograr una alta correlación con los juicios humanos sobre la calidad y sigue siendo una de las métricas más populares y utilizadas en el campo de la traducción automática. Para ello, he utilizado la librería de *python sacrebleu* y he traducido con cada uno de los modelos las frases del conjunto de prueba, obteniendo los resultados que se muestran en la figura 5.9.

```
BLEU SCORE TRANSFORMER BASE: 7.271007363617218  
BLEU SCORE TRANSFORMER 1: 7.77536842701581  
BLEU SCORE TRANSFORMER 2: 11.382946616278247
```

Figura 5.9: Score de *sacreBLEU* para los *transformers* 1 y 2.

Como se puede observar, el modelo base obtiene una puntuación de 7,27, mientras que el *transformer* 1 obtiene una puntuación de 7,78, lo cual no supone una gran mejora. Sin embargo, el *transformer* 2 puntuó 11,38, que es un 46,27% superior al *transformer* 1. Como consecuencia, optaré por aumentar el número de capas, en lugar del número de cabezales, para construir el modelo final.

### *Transformer* final

El análisis de los resultados anteriores permite ver que aumentar el número de cabezales tiene un mayor impacto en cuanto a tiempos de entrenamiento que aumentar el número de capas. Por esta razón, decidí crear un modelo final en el que dejo el número de cabezales en 8 y aumento el número de capas a 4, es decir, este modelo tendrá 4 *encoders* seguidos de 4 *decoders*. En la figura 5.10 y la figura 5.11, así como en el archivo `transformer_final_architecture.png` del repositorio del proyecto<sup>8</sup>, se muestra la arquitectura del modelo en detalle.

<sup>6</sup> *OneDrive* del proyecto: [https://ucomplutense-my.sharepoint.com/:f/g/personal/mimora02\\_ucm\\_es/EpydMcjnsEJHkTRVycbAA8EBFRQMjzfQFhnJGbe5bDzXug](https://ucomplutense-my.sharepoint.com/:f/g/personal/mimora02_ucm_es/EpydMcjnsEJHkTRVycbAA8EBFRQMjzfQFhnJGbe5bDzXug)

<sup>7</sup> Métrica *sacreBLEU*: <https://huggingface.co/spaces/evaluate-metric/sacrebleu>

<sup>8</sup> Github del proyecto: <https://github.com/miguelangelmoralesramon/tfg.git>

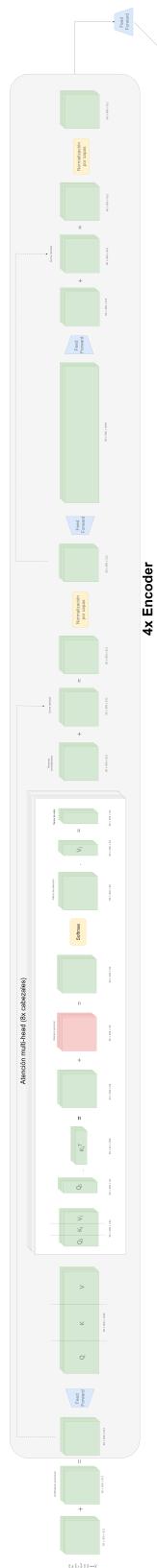


Figura 5.10: Arquitectura del encoder del *transformer* final (elaboración propia).

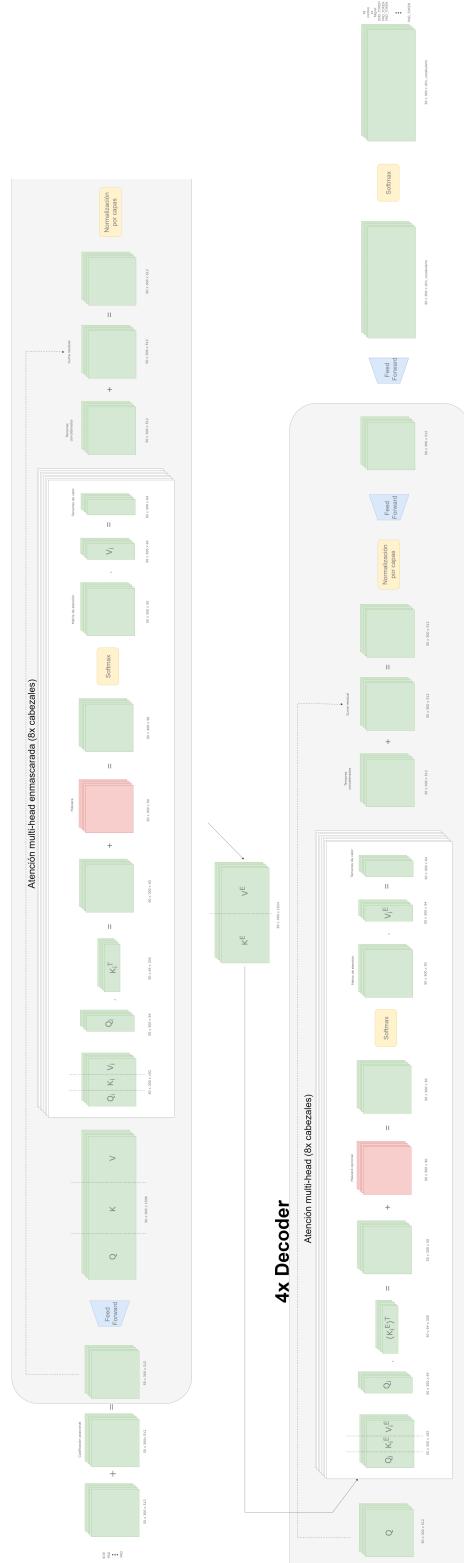


Figura 5.11: Arquitectura del *decoder* del *tranformer* final (elaboración propia).

Para entrenar al modelo he utilizado las primeras  $5 \cdot 10^5$  frases o la mitad del conjunto de entrenamiento, con 50 *samples* por lote, 10 *epochs* y optimizador *Adam* con tasa de aprendizaje  $\eta = 10^{-4}$ . El tiempo total de entrenamiento ha sido de 30161,23 segundos (8 horas, 22 minutos y 41 segundos) y la *score* BLEU obtenida sobre el conjunto de prueba ha sido de 28,18, lo cual supone una importante mejora y está a la altura de las métricas publicadas en el paper de referencia [14]. A continuación, se muestran algunas traducciones de ejemplo de este modelo. Como se puede observar, en algunas ocasiones la traducción generada por el *transformer* es mejor que la traducción del *dataset*.

Frase: - oh, i love you.

Traducción: - le quiero.

Traducción transformer: - oh, te amo.

---

Frase: my apartment will be ready next week.

Traducción: mi departamento va estar listo la próxima semana.

Traducción transformer: mi apartamento estará listo para la semana que viene.

---

Frase: many years later, there are still many people who believe that mao zedong only started to attack the intellectuals after becoming impatient with their overly harsh criticisms.

Traducción: muchos años después, aún hay muchas personas que creen que mao zedong sólo ordenó el ataque a los intelectuales cuando las críticas excesivamente duras le hicieron perder la paciencia.

Traducción transformer: muchos años más tarde, hay muchas personas que creen que mao zedong sólo comenzó a atacar los intelectuales después de que se vuelvan impacientes con sus críticas en toda la mayoría.

---

Frase: don't tell me it wasn't you. you'll just make it twice as bad if you lie to me.

Traducción: sólo lo harás dos veces peor si me mientes.

Traducción transformer: no me digas que no eras tú, sólo lo harás dos veces como si me mientes.

---

Frase: also we cannot say that the markets or the investment houses should have won, but everyone should be in this market together in the future, in order then to attract customers with the best offer.

Traducción: tampoco podemos decir que los mercados o las sociedades de inversión hayan ganado, sino que todos deberán estar juntos en este mercado en el futuro a fin de atraer a los clientes mediante la mejor oferta.

Traducción transformer: también no podemos decir que los mercados o las casas de inversión deberían haber ganado, pero todo el mundo deberían estar juntos al futuro, para atraer a los clientes a alguien mejor.<EOE>

---

Frase: i would like, in particular, to highlight the desirability of enhanced international cooperation between competition law enforcement authorities.

Traducción: concretamente, me gustaría destacar la conveniencia de una cooperación internacional cada vez más amplia entre las autoridades encargadas de aplicar el derecho de competencia.

Traducción transformer: quisiera en particular, a destacar la deseabilidad de ampliar la cooperación internacional entre las autoridades de la ley de la competencia.

---

Frase: the development of civil societies, economic growth, and openness to the world are equally important.

Traducción: el desarrollo de sociedades civiles, el crecimiento económico y la apertura hacia el mundo también son importantes.

Traducción transformer: el desarrollo de las sociedades civiles, el crecimiento económico, y abierto al mundo son igualmente importantes.

---

Frase: i heard that the special investigation committee was formed after what happened to dr. kim myung guk.

Traducción: he oido que una comisión especial de investigación se... formó después de lo que pasó con el dr. kim myung guk.

Traducción transformer: oí que el comité de investigación especial fue formado después de lo que le pasó al dr. kim myung guk.

---

Frase: mr. rechetov said that while that proposal was of interest, if the committee decided to include such information in the first paragraph of its concluding observations for france, it would have to instruct the secretariat to do so in the concluding observations for all countries.

Traducción: el sr. rechetov dice que si bien la propuesta es interesante, si el comité decide incluir esa información en el primer párrafo de sus observaciones finales para francia, tendría que pedir a la secretaría que lo hiciera en las observaciones finales para todos los países.

Traducción transformer: el sr. rechetov dijo que aunque propuesta era de interés, si el comité decidió incluir tal información en el párrafo primero de sus observaciones concluidas para francia, tendría que instruir la secretaría para hacerlo en las observaciones concluidas para todos los países.

---

Frase: the qualitative study also showed that the media broadcasting through television satellite channels have an adverse cultural and social impact on the child's social integration in society, which is noticeable in child behaviour, particularly violence.

Traducción: el estudio también demostró que los programas de los canales de televisión que se captan por satélite tienen repercusiones culturales y sociales negativas para la integración social del niño, lo que se pone de manifiesto en su comportamiento, sobre todo en el grado de violencia.

Traducción transformer: el estudio cualitativo también mostró que los medios de comunicación de televisión satélite tienen un efecto cultural y social sobre la integración social del niño, que es notificable en comportamiento infantil de la violencia particular de la niña.

---

El modelo final entrenado está disponible en el archivo `transformer_final.pth.tar` del *OneDrive* del proyecto<sup>9</sup>.

---

<sup>9</sup> *OneDrive* del proyecto: [https://ucomplutense-my.sharepoint.com/:f/g/personal/mimora02\\_ucm\\_es/EpydMcjnsEJHkTRVycbAA8EBFRQMjzfQFhnJGbe5bDzXug](https://ucomplutense-my.sharepoint.com/:f/g/personal/mimora02_ucm_es/EpydMcjnsEJHkTRVycbAA8EBFRQMjzfQFhnJGbe5bDzXug)

# Capítulo 6

## Conclusiones

Uno de los objetivos principales del proyecto era realizar un desarrollo teórico de los tipos de redes neuronales más importantes en el campo del aprendizaje automático. He logrado no solo abarcar las redes neuronales artificiales, convolucionales, recurrentes y los transformers, sino que también he profundizado en cada una de ellas de manera rigurosa y matemática. He construido cada concepto sobre los anteriores, como si de un castillo de naipes se tratara, asegurando una comprensión coherente y progresiva de cada tipo de red.

El enfoque adoptado ha permitido que cada tipo de red neuronal sea presentado con una base teórica sólida. Por ejemplo, las redes neuronales artificiales se introducen primero, estableciendo los fundamentos necesarios para comprender las redes convolucionales y recurrentes. Posteriormente, los transformers se presentan como una evolución natural y sofisticada de los conceptos anteriores. Este enfoque incremental no solo facilita la comprensión, sino que también muestra cómo las innovaciones en arquitectura y algoritmo han llevado a mejoras en el rendimiento y capacidad de las redes neuronales.

Además del desarrollo teórico, he llevado a cabo experimentos empíricos al final de cada capítulo para validar la teoría presentada. Estos experimentos no solo sirven como pruebas de concepto, sino que también permiten comparar de manera directa el rendimiento de los distintos tipos de redes neuronales. Por ejemplo, en el capítulo sobre redes neuronales artificiales, se realiza un experimento utilizando un *dataset* específico. Este mismo *dataset* se emplea posteriormente en el capítulo sobre redes convolucionales, lo que permite observar claramente cómo las redes convolucionales mejoran el rendimiento en comparación con las redes neuronales artificiales básicas. Esta metodología de comparación directa facilita la apreciación de las ventajas y limitaciones de cada tipo de red.

De forma similar, en el capítulo 4 se ponen a prueba los 3 tipos de redes recurrentes más usadas: las RNN, las LSTM y las GRU. Las he entrenado con los mismos datos y condiciones de entrenamiento y la que mejor resultado obtuvo fue la red GRU, en línea con lo que indicaba la teoría. Esto se debe la estructura que tiene este tipo de red, más simplificada y eficiente que las LSTM y con más capacidad para manejar memoria a largo plazo que las RNN. La GRU combina la capacidad de capturar dependencias temporales prolongadas con una menor complejidad computacional, lo que resulta en un mejor rendimiento. Además, en este caso la GRU ha mitigado con mayor eficacia los problemas de desvanecimiento del gradiente, al haber aprendido más rápido que los otros tipos de redes.

También, en el capítulo dedicado a los transformers, he podido entrenar este tipo de modelos pese a las dificultades iniciales de *hardware* que tenía, que pude superar alquilando una GPU NVIDIA A100 en *Google Colab*. Este tipo de arquitectura requiere una capacidad de computación que difícilmente se puede alcanzar sin acceso a recursos avanzados como *Google Colab* o GPUs de NVIDIA. Los resultados obtenidos son inmensamente satisfactorios para mí, ya que he sido capaz de desarrollar la arquitectura del *transformer* desde 0 en *python* (archivo *Transformer.py*<sup>1</sup>) y

---

<sup>1</sup>Github del proyecto: <https://github.com/miguelangelmoralesramon/tfg.git>

crear una clase que replica a la perfección este tipo de redes y que funciona correctamente. Este ha sido sin duda la mayor dificultad y, a su vez, el mayor éxito de este trabajo. El modelo final ha obtenido una puntuación de 28,18 en la métrica *sacreBLEU* y en muchas ocasiones ha generado mejores traducciones que las del propio *dataset*, lo cual es una muestra del potencial de este tipo de redes neuronales.

Hubiera sido interesante incluir una aplicación práctica de un *Large Language Model* (LLM) como *GPT* o *Gemini*, que están basados en transformers. Esta adición habría proporcionado una visión práctica y aplicada del poder de los transformers en procesamiento del lenguaje natural y otras tareas complejas. Hubiera sido interesante desarrollar un *chatbot* especializado en conceptos matemáticos y científicos, pensado para la comunidad de investigadores. Mi primera aproximación a este problema habría pasado por aplicar técnicas de *fine-tunning* [16] a un LLM pre-entrenado, junto con técnicas de *Retrieval-Augmented Generation* (RAG) [7].

Otra línea de investigación que considero prometedora trata sobre las redes neuronales de tipo *Kolmogorov-Arnold* (KAN) [8]. Este tipo de red, que salió a la luz en mayo de 2024, ha supuesto un cambio de paradigma en el aprendizaje automático. Las redes KAN reducen sustancialmente los tiempos de entrenamiento y aumentan la precisión en comparación con las redes tradicionales que hemos estudiado. Se basan en el teorema de representación de Kolmogorov-Arnold, el cual permite descomponer funciones multivariadas en sumas de funciones de una sola variable. Este enfoque innovador no solo simplifica la estructura de la red, sino que también mejora la eficiencia computacional.

# Bibliografía

- [1] Hamed Habibi Aghdam, Elnaz Jahani Heravi, et al. Guide to convolutional neural networks. *New York, NY: Springer*, 10(978-973):51, 2017.
- [2] Fernando Berzal. *Redes neuronales & deep learning: Volumen II*. Independently published, 2019.
- [3] Eduardo Francisco Caicedo Bravo. *Una aproximación práctica a las redes neuronales artificiales*. Universidad del Valle, 2009.
- [4] Nithin Buduma, Nikhil Buduma, and Joe Papa. *Fundamentals of deep learning*. .Ó'Reilly Media, Inc.", 2022.
- [5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.
- [6] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.
- [7] Patrick S. H. Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. Retrieval-augmented generation for knowledge-intensive NLP tasks. *CoRR*, abs/2005.11401, 2020.
- [8] Ziming Liu, Yixuan Wang, Sachin Vaidya, Fabian Ruehle, James Halverson, Marin Soljačić, Thomas Y. Hou, and Max Tegmark. Kan: Kolmogorov-arnold networks, 2024.
- [9] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013.
- [10] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.
- [11] Xin Rong. word2vec parameter learning explained. *CoRR*, abs/1411.2738, 2014.
- [12] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016.
- [13] Abhishek Thakur. *Approaching (almost) any machine learning problem*. Abhishek Thakur, 2020.
- [14] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.
- [15] Aston Zhang, Zachary C Lipton, Mu Li, and Alexander J Smola. Dive into deep learning. *arXiv preprint arXiv:2106.11342*, 2021.
- [16] Biao Zhang, Zhongtao Liu, Colin Cherry, and Orhan Firat. When scaling meets llm finetuning: The effect of data, model and finetuning method, 2024.