

Proyecto Fin de Carrera

Ingeniería de Telecomunicación

Deep Learning para el Reconocimiento de Texto Manuscrito

Autor: Hugo Rubio Sánchez

Tutor: Juan José Murillo Fuentes

Dep. Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2018



Proyecto Fin de Carrera
Ingeniería de Telecomunicación

Deep Learning para el Reconocimiento de Texto

Manuscrito

Autor:
Hugo Rubio Sánchez

Tutor:
Juan José Murillo Fuentes
Prof. Catedrático de Universidad

Dep. de Teoría de la Señal y Comunicaciones
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2018

Proyecto Fin de Carrera: Deep Learning para el Reconocimiento de Texto Manuscrito

Autor: Hugo Rubio Sánchez

Tutor: Juan José Murillo Fuentes

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

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2018

El Secretario del Tribunal

A mi madre y a mi abuelo.

Agradecimientos

Agradecimientos a mis padres Eladio y Milagros, a mi abuela Eduarda y al resto de mi familia, por haberme ayudado y apoyado en las distintas etapas de mi vida, guiándome y creyendo en mí en todo momento. Soy resultado de sus esfuerzos.

A todos mis compañeros de clase y del trabajo, que muchos hoy día son buenos amigos. Sin vosotros todo hubiera sido más duro y, sobre todo, más aburrido.

A la Universidad de Sevilla y en concreto a la Escuela Técnica Superior de Ingeniería y a la Escuela Técnica Superior de Ingeniería Informática. Por todas las enseñanzas y por los conocimientos transmitidos.

Y agradecimientos especiales a mis dos maestros *Jedi* en el mundo de la Inteligencia Artificial, Fernando y Pedro. Sin su ayuda no hubiera sido posible.

Actualmente, vivimos en un mundo rodeados de aplicaciones y sistemas basados en algoritmos de Aprendizaje Automático, o Aprendizaje Profundo, dado que cuentan con una gran capacidad para adaptarse a multitud de situaciones.

En este Trabajo Fin de Grado se pretende exponer el comportamiento de un tipo concreto de dichos algoritmos, las Redes Neuronales Artificiales, afrontando un problema de Reconocimiento de Escritura Manuscrita. Durante el desarrollo del proyecto, se ha puesto especial interés en el apartado teórico, en relación con los algoritmos y la arquitectura del sistema.

Cabe destacar que, aunque no se ha perseguido obtener un alto rendimiento en las transcripciones, se ha diseñado un sistema implementado en Python3, usando la librería TensorFlow, que se podría marcar como punto de inicio para construir un sistema de transcripción robusto.

Abstract

Today, we live in a world surrounded by applications and systems based on Machine Learning, or Deep Learning, since they have a great capacity to adapt to many situations.

This Final Degree Project intends to expose the behavior of a specific type of these algorithms, the Artificial Neural Networks, facing a problem of handwriting recognition. During the development of the project, special interest has been placed on the theoretical aspect, in relation to algorithms and system architecture.

It should be noted that, although it has not sought to obtain high performance in transcriptions, a system implemented in Python3 has been designed, using the TensorFlow library, which could be marked as a starting point to build a robust transcription system.

Índice

Agradecimientos	ix
Resumen	xi
Abstract	xiii
Índice	xiv
Índice de Tablas	xvi
Índice de Figuras	xviii
Notación	xx
1 Introducción	1
2 Datos	5
2.1 Fuente	5
2.2 Preprocesado de los Datos.	5
2.3 Imágenes como Tensores.	7
3 Capas ANN	9
3.1 Redes Neuronales Artificiales (ANN).	9
3.1.1 Descenso del Gradiente y Retropropagación del Error.	10
3.2 Redes Neuronales Convolucionales (CNN).	12
3.2.1 Convolución entre tensores.	12
3.2.2 Capa CNN.	13
3.3 Redes Neuronales Recurrentes (RNN).	15
3.4 Redes Long Short-Term Memory (LSTM).	17
3.5 Connectionist Temporal Classification (CTC).	19
3.5.1 Softmax.	19
3.5.2 De Distribuciones de Probabilidad a Etiquetas.	19
3.5.3 Algoritmo CTC forward-backward.	20
3.5.4 Medida de Error y Función objetivo.	22
4 Modelo ANN	25
4.1 Arquitectura para Extracción de Características (AEC).	25
4.2 Arquitectura Recurrente Bidireccional (ARB).	26
4.3 Arquitectura para Reducción de Dimensionalidad (ARD).	26
4.4 Arquitectura Global.	27
5 Pruebas y Resultados	28
5.1 Validación Cruzada (Cross-Validation).	28
5.2 Test IAM.	30
6 Conclusiones y Trabajos Futuros	33
6.1 Conclusiones.	33
6.2 Trabajo Futuro.	35
Referencias	36
Anexos	38

<i>Anexo A. Código de la Implementación.</i>	38
Directorio del Proyecto.	38
hw_utils.py.	39
clean_IAM.py.	44
ANN_model.py.	45
cross-validation.py.	48
train.py.	51
test.py.	55
config.json.	58
<i>Anexo B. Manual de Uso.</i>	60
1. Deep Learning para el Reconocimiento de Texto Manuscrito implementado en TensorFlow.	60
1.1. Estructura	60
1.1.1. Ficheros Python	60
1.1.2. Ficheros CSV	60
1.1.3. Ficheros de Configuración	60
1.2. Primeros Pasos	61
1.2.1. Requisitos	61
1.2.2. Instalación y preprocesado de datos	61
1.3. Ejecución	62
1.3.1. Cross-Validation	62
1.3.2. Test IAM	62

ÍNDICE DE TABLAS

Tabla 2–1. Caracteres y sus correspondientes etiquetas.	7
Tabla 2–2. Correspondencias de los Tensores.	7
Tabla 4–1. Hiperparámetros del modelo.	27
Tabla 5–1. Valores mínimos para el LER y el Coste.	30
Tabla 5–2. Resultados del Test IAM.	32

ÍNDICE DE FIGURAS

Figura 1-1. Capacidades del Reconocimiento de Caracteres.	1
Figura 1-2. Abstracción del Modelo objetivo.	3
Figura 2-1 Ejemplos de la <i>IAM Handwriting Database</i> .	5
Figura 2-2. Imagen con la palabra “down” y su tensor asociado.	6
Figura 2-3. Imagen procesada y tensor asociado.	6
Figura 2-4. Imagen procesada y tensor asociado normalizado.	6
Figura 2-5. Ejemplos de tensores de distinto orden.	8
Figura 3-1. Ejemplo de una ANN unidireccional con 4 capas.	9
Figura 3-2. Neurona artificial inspirada en la neurona biológica.	9
Figura 3-3. Representación del error en función de los pesos w_0 y w_1	11
Figura 3-4. Ejemplo de tensor de orden 2 y dimensiones 32×32 extendiendo hasta 36×36 .	12
Figura 3-5. Ejemplo de convolución entre tensores de orden 2, con $P = [1,1]$ y sin relleno en X.	13
Figura 3-6. Abstracción de una capa aislada	14
Figura 3-7. Ejemplos de topologías recurrentes y representación de una neurona aislada.	15
Figura 3-8. Abstracción de una RNN extendida en el tiempo.	16
Figura 3-9. Representación simbólica de los efectos del Desvanecimiento del Gradiente.	17
Figura 3-10. Estructura de una celda Modern LSTM.	17
Figura 3-11. Salida de la capa Softmax, representación simbólica.	20
Figura 4-1. Arquitecturas que componen el modelo.	25
Figura 4-2. Max-Pool.	25
Figura 4-3. Dropout.	26
Figura 4-4. Arquitectura Global.	27
Figura 5-1. Representación del dataset en las distintas evaluaciones.	28
Figura 5-2. Resultados del Coste para la validación cruzada, representados en media.	29
Figura 5-3. Resultados del LER para la validación cruzada, representados en media.	29
Figura 5-4. Resultados Coste para el test IAM.	31
Figura 5-5. Resultados LER para test IAM.	31

Notación

A	Tensor de un determinado orden.
$A_{i_0, i_1, \dots}$	Subtensor del tensor A , de orden menor.
$a_{i_0, i_1, \dots}$	Elemento escalar del tensor A
X	Entrada del sistema.
Y	Salida objetivo.
Y'	Salida del sistema.
Ω	Conjunto de todos los casos posibles.
Λ	Conjunto de todas las etiquetas posibles.
S	Conjunto de muestras.
$f(\cdot)$	Función genérica.
$f_{err}(\cdot)$	Función de error genérica.
$f_a(\cdot)$	Función de activación genérica.
\tanh	Función tangente hiperbólica.
<i>Sigmoide</i>	Función sigmoide.
$X * Y$	Convolución de los tensores X e Y .
$\sum_{i=0}^k$	Sumatorio desde $i = 0$ hasta $i = k$.
$\prod_{i=0}^k$	Productorio desde $i = 0$ hasta $i = k$.
$\frac{\partial y}{\partial x}$	Derivada parcial de y respecto de x
$p(a)$	Probabilidad del suceso a .
$ $	Condicionado.
$p(a b)$	Probabilidad del suceso a condicionada por el suceso b .
\forall	Para todo.
\in	Pertenece.
$\max\{\}$	Valor máximo.
$\min\{\}$	Valor mínimo.
$<$	Menor.
$>$	Mayor.
\leq	Menor o igual.
\geq	Mayor o igual.
e.o.c.	En cualquier otro caso.

1 INTRODUCCIÓN

Hasta la fecha, no se ha diseñado un ordenador que sea consciente de lo que está haciendo; pero, la mayor parte del tiempo, nosotros tampoco lo somos

- Marvin Minsky -

El reconocimiento automático de escritura manuscrita es un desafío realmente interesante desde el punto de vista académico, ya que para afrontarlo es necesario combinar visión artificial con aprendizaje secuencial. A diferencia del reconocimiento óptico de caracteres, en el caso manuscrito la segmentación de cada carácter es difícil debido a los distintos tipos de caligrafía y a la naturaleza cursiva de la escritura a mano. Estos factores obligaron a diseñar modelos de reconocimiento de palabras completas o líneas de texto, es decir, la secuencia de caracteres.

El durante el presente proyecto se mostrará: (i) cómo se definen este tipo de problemas, donde se pretende extraer la palabra manuscrita de una imagen, (ii) cómo ha sido afrontado con algoritmos basados en el Aprendizaje Automático [1], que consiguen “leer” el contenido de dicha imagen, y (iii) cuales han sido los resultados de las evaluaciones del sistema y las conclusiones obtenidas.

1.1 Estado del Arte

En los sistemas de reconocimiento de caracteres tiene lugar un proceso de conversión de imágenes de texto impreso, o texto manuscrito, a un formato de fácil procesamiento por un ordenador, tal como es el código UTF-8. Estos sistemas de reconocimiento se pueden clasificar como sistemas de Reconocimiento Óptico de Caracteres (OCR, del inglés *Optical Character Recognition*), sistemas de Reconocimiento Inteligente de Caracteres (ICR, del inglés *Intelligent character recognition*), y sistemas de Reconocimiento de Manuscrito Natural (NHR, del inglés *Natural Handwriting Recognition*).

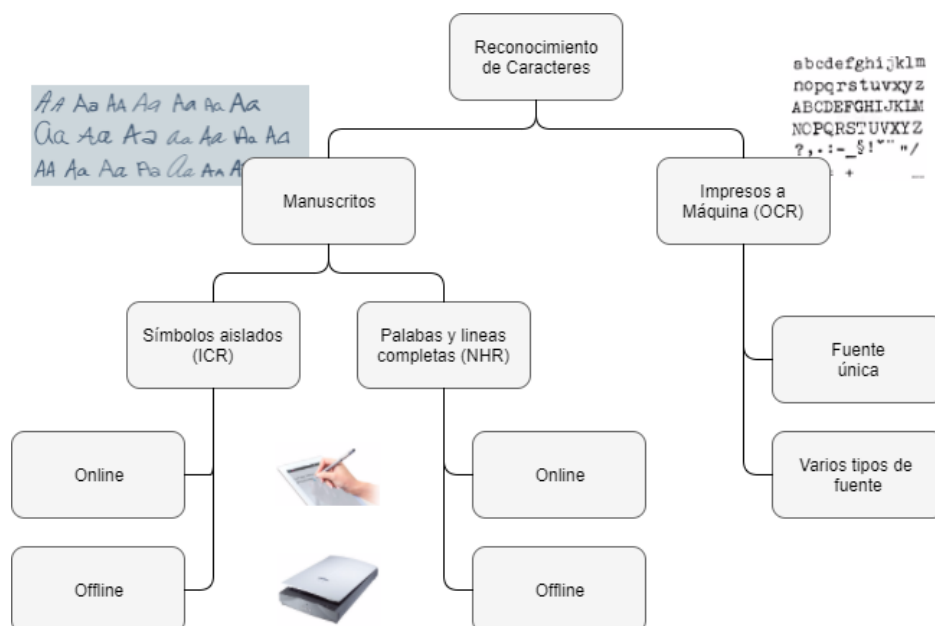


Figura 1-1. Capacidades del Reconocimiento de Caracteres.

Los sistemas OCR e ICR permiten el reconocimiento de caracteres aislados, impresos y manuscritos respectivamente, lo que requiere un proceso previo de segmentación y extracción para la transcripción de palabras o líneas. Ese preprocesado limita la eficiencia del sistema, especialmente en el reconocimiento de textos manuscritos, siendo en algunos casos imposible aislar correctamente los símbolos. Para resolver este problema hubo que cambiar el enfoque y aparecieron los sistemas NHR, capaces de reconocer palabras y líneas manuscritas, y que no solo eliminan las limitaciones del preprocesado, sino que son capaces de extraer más información del contexto al trabajar con la entrada completa.

Dentro de los sistemas ICR y NHR podemos hacer distinciones según el tipo de entrada, donde podemos diferenciar entre sistemas de reconocimiento dinámico y sistemas de reconocimiento estático, también conocidos como sistemas online y offline, respectivamente. Los sistemas de reconocimiento dinámico reconocen el texto mientras el usuario está escribiendo con un dispositivo de escritura en línea, capturando la información temporal o dinámica de la escritura. Esta información incluye el número, la duración y el orden de cada trazo. Por otro lado, los sistemas de reconocimiento estático trabajan con el texto completo, que ha sido previamente manuscrito y digitalizado en una imagen como un mapa de bits. Los sistemas estáticos no tienen acceso a la información dependiente del tiempo capturada en sistemas dinámicos y, por esta razón, el reconocimiento estático se considera una tarea más compleja y desafiante.

Dicho esto, podemos definir nuestra tarea como un problema de reconocimiento estático de manuscritos (OHR, del inglés *Offline Handwriting Recognition*) [2]. Tradicionalmente, estos problemas han sido afrontados con Modelos Ocultos de Markov (HMM, del inglés *Hidden Markov Model*) [3] y en las últimas décadas con arquitecturas de Aprendizaje Profundo (DL, del inglés *Deep Learning*) [4] basadas en Redes Neuronales Artificiales (ANN, del inglés *Artificial Neural Networks*) [5].

En los HMMs, los caracteres son modelados como secuencias de estados ocultos, asociados con un modelo de probabilidad de emisión, siendo el Modelo de Mezcla Gaussiana (GMM, del inglés *Gaussian Mixture Model*) [6] el modelo óptico estándar en HMMs. Sin embargo, los estados ocultos siguen una cadena de Markov de primer orden y no pueden manejar dependencias a largo plazo. Por otra parte, en cada paso los HMMs sólo pueden seleccionar un estado oculto, por lo tanto, un HMM con n estados ocultos por lo general puede llevar sólo $\log_2(n)$ bits de información sobre su dinámica [7].

Las Redes Neuronales Recurrentes (RNN, del inglés *Recurrent Neural Networks*) [8] no tienen tales limitaciones y han demostrado su eficacia en el modelado de secuencias [9]. Las RNNs son un tipo de ANN donde las conexiones entre las neuronas forman ciclos dirigidos, lo que genera un estado interno que consigue un comportamiento temporal dinámico para modelar secuencias largas con estructuras complejas. Estas arquitecturas son intrínsecamente profundas en el tiempo y pueden tener muchas capas, haciendo que su entrenamiento sea un problema de optimización difícil. La explosión y el desvanecimiento del gradiente fueron las razones de la falta de aplicaciones prácticas de las RNN [10]. Para afrontar estos problemas se realizaron avances en el diseño de RNNs, donde destacan las llamadas redes *Long Short-Term Memory* (LSTM) [11], que están compuestas por celdas cuidadosamente diseñadas que sustituyen a las clásicas neuronas, dando un rendimiento superior en una amplia gama de problemas de modelado de secuencias.

Además, un sistema OHR basado en DL necesita procesar imágenes y extraer características con las que finalmente realizar el aprendizaje secuencial. Una tarea que cae dentro del campo de la Visión Artificial. En la última década las Redes Neuronales Convolucionales (CNN, del inglés *Convolutional Neural Networks*) [12] han obtenido buenos resultados dentro de dicho campo [13, 14, 15], lo que ha resultado en su extendido uso en OHR.

Aunque las arquitecturas que incluían RNNs y CNNs consiguieron mejorar los resultados, estos sistemas no experimentaron un progreso sustancial hasta la aparición de las redes *Multi-Dimensional Long Short-Term Memory* (MLSTM) [16, 17] y las *Connectionist Temporal Classification* (CTC) [18], redes que consiguen aprovechar más información y mejorar el aprendizaje de manera significativa.

1.2 Objetivo

La finalidad de este proyecto ha sido exponer el funcionamiento de modelos basados en DL sobre problemas de OHR, donde se ha preferido ofrecer una robusta fundamentación teórica del sistema, destacando los aspectos matemáticos, a cambio de perseguir un alto rendimiento en las transcripciones.

Para ello se pretende crear un sistema capaz de tomar una imagen, en forma de tensor de entrada, y devolver una palabra, en forma de secuencia de etiquetas.

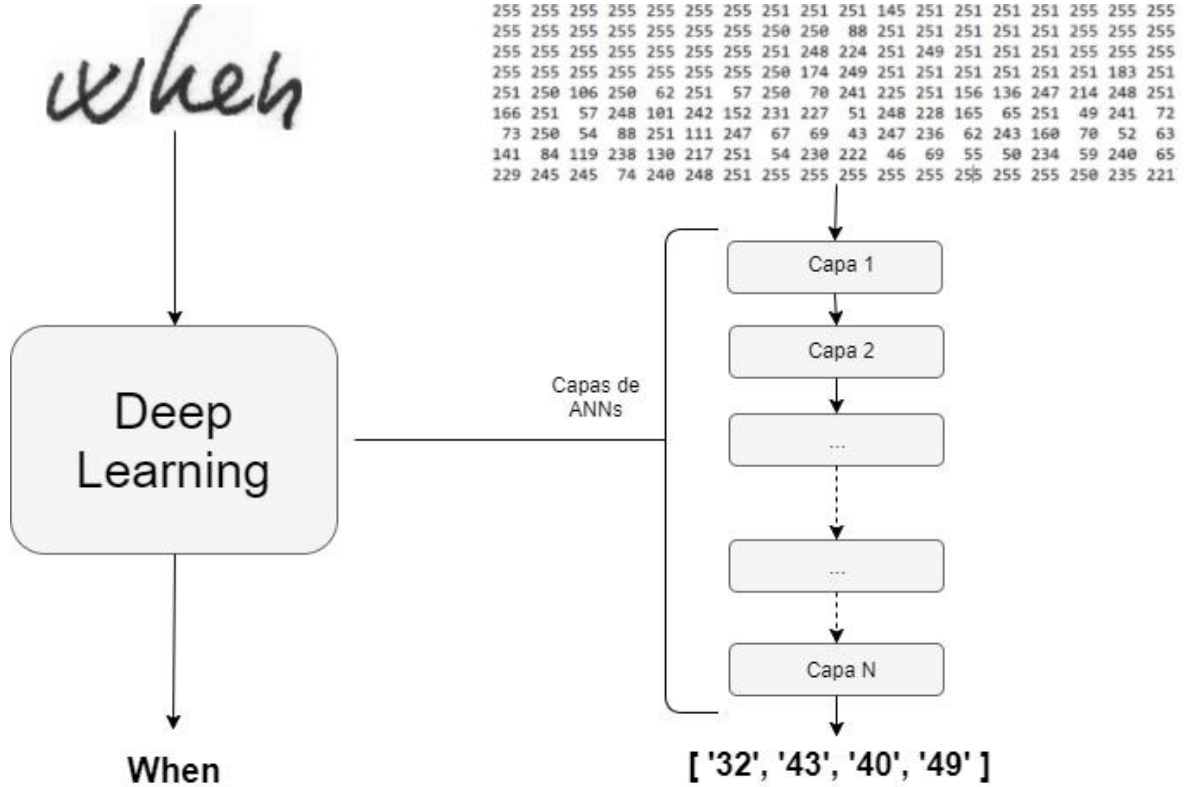


Figura 1-2. Abstracción del Modelo objetivo.

Para diseñar este sistema es necesario asumir que exista una relación entre un tensor X , que representa una imagen, y una secuencia Y de etiquetas, que representa la palabra escrita. Es lógico pensar que esta relación existe puesto que las personas somos capaces de reconocer la secuencia de símbolos cuando vemos una palabra escrita. Desde un punto de vista abstracto, podemos pensar en una función $f(\cdot)$ que transforma el tensor X en la secuencia Y dentro de un conjunto Ω , formado por pares (X, Y) , que representa todas las posibles imágenes con palabras escritas:

$$Y = f(X), \quad \forall (X, Y) \in \Omega \quad (1-1)$$

Bajo esta premisa y dado que conocemos la palabra escrita en cada imagen, podemos diseñar un sistema que “aprenda a leer” mostrándole imágenes y ajustando la salida hasta llegar a la salida ideal y conocida. Este tipo de aprendizaje se conoce como aprendizaje supervisado y, a diferencia del aprendizaje no supervisado, se trata deducir una función a partir de datos de entrenamiento generalmente compuestos por pares (X, Y) .

En este sentido, el sistema de OHR también se comporta como una función $f'(\cdot)$ que toma como entrada un tensor X y unos parámetros θ , propios del sistema, para dar como resultado una secuencia Y' , que tomaremos como salida del sistema:

$$Y' = f'(X, \theta) \quad (1-2)$$

Nuestro propósito es aproximar Y' , salida propia del sistema, a Y , salida ideal, para cualquier par (X, Y) de todo Ω . Para lograrlo, y reescribiendo nuestro problema como un problema de optimización, podemos declarar un primer objetivo genérico como:

$$\text{Minimizar } E = f_{err}(Y', Y), \quad \forall (X, Y) \in \Omega \quad (1-3)$$

Siendo $f_{err}(\cdot)$ una función error que mide la diferencia entre dos secuencias, y que toma el valor 0 cuando ambas son iguales. Partiendo de esta función objetivo y un subconjunto $S \subset \Omega$, ya que no podemos manejar todo el conjunto Ω , el problema general de aprendizaje consiste en minimizar, o encontrar valores suficientemente pequeños de este error y realizar distintas evaluaciones sobre los diversos modelos encontrados.

La función $f'(\cdot)$ del sistema de OHR se obtiene mediante arquitecturas DL y nuestro objetivo de optimización se convierte por tanto en optimizar los parámetros θ aplicando algoritmos de aprendizaje basados en el Descenso del Gradiente [19].

Para ello usaremos arquitecturas formadas por distintas capas de ANNs, donde podemos destacar el uso de capas CNNs y RNNs, en particular las LSTM, para el procesamiento de la imagen, y una última capa CTC para el aprendizaje secuencial. El modelo ha sido implementado en Python3 usando TensorFlow como librería para el diseño de la arquitectura y ha sido entrenado con palabras manuscritas individuales, aunque aplicando pequeños cambios se podría adaptar para líneas completas de texto manuscrito.

La arquitectura planteada en este proyecto está fundamentada en distintas publicaciones donde se afronta este problema desde el DL [15, 17, 20, 21].

2 DATOS

Como ya se ha comentado en el capítulo anterior, es necesario un subconjunto de muestras $S \subset \Omega$ para realizar el entrenamiento y la evaluación del sistema. En el problema que nos ocupa este subconjunto estará formado por un conjunto de imágenes, con sus correspondientes transcripciones, en un formato determinado.

2.1 Fuente

El Instituto de Informática y Matemáticas Aplicadas de la Universidad de Berna (Suiza) publicó en 1999 la *IAM Handwriting Database* [22], una base de datos pública que en su última versión contiene 1.539 páginas manuscritas en inglés de 657 escritores. Los manuscritos han sido transcritos, escaneados y segmentados en frases, líneas de texto y palabras.

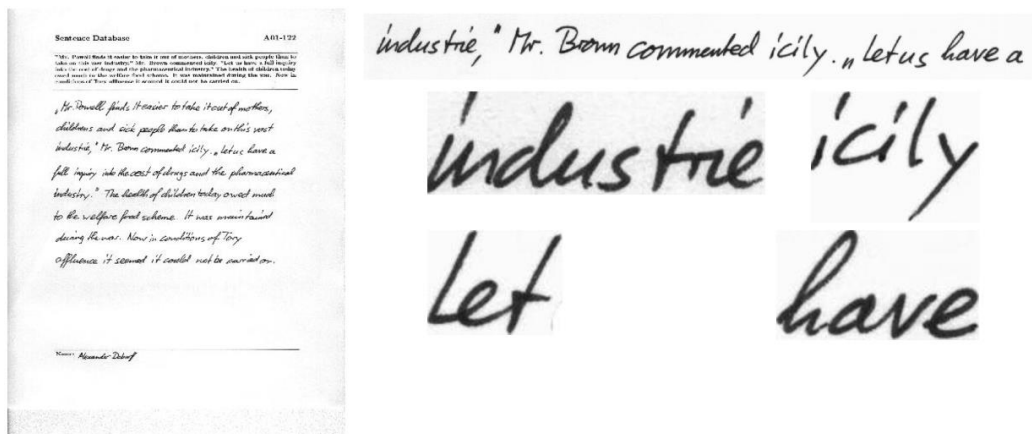


Figura 2-1 Ejemplos de la *IAM Handwriting Database*.

Todos los documentos tienen una resolución de 300dpi y están almacenados como imágenes PNG en escala de grises con 256 niveles. Concretamente, esta base de datos contiene un dataset de 115.320 palabras formado por las imágenes y su metainformación asociada, incluyendo su transcripción.

2.2 Preprocesado de los Datos.

Para simplificar el problema, y con el fin de manejar una cantidad menor de etiquetas, se han filtrado estas muestras de imágenes quedándonos con aquellas que incluyen únicamente números y letras en la palabra transcrita, dando como resultado un dataset de 87.107 de muestras.

Tras este filtrado, se ha realizado un preprocesamiento de las imágenes, dándoles a todas ellas una altura de 64 píxeles y añadiendo relleno hasta alcanzar una anchura de 1024 píxeles, manteniendo sus proporciones. Tras el reescalado, se han invertido los tonos en la escala de grises, transformando su tensor asociado en el proceso. Estos dos tratamientos no son estrictamente necesarios, pero ayudan a implementar el modelo en TensorFlow al uniformizar el formato de los datos de entrada.

Tabla 2–1. Caracteres y sus correspondientes etiquetas.

0	0		A	10		K	20		U	30		e	40		o	50		y	60
1	1		B	11		L	21		V	31		f	41		p	51		z	61
2	2		C	12		M	22		W	32		g	42		q	52			
3	3		D	13		N	23		X	33		h	43		r	53			
4	4		E	14		O	24		Y	34		i	44		s	54			
5	5		F	15		P	25		Z	35		j	45		t	55			
6	6		G	16		Q	26		a	36		k	46		u	56			
7	7		H	17		R	27		b	37		l	47		v	57			
8	8		I	18		S	28		c	38		m	48		w	58			
9	9		J	19		T	29		d	39		n	49		x	59			

Como resultado final tenemos un dataset $S \subset \Omega$ compuesto por pares (X, Y) donde X es un tensor normalizado que representa una imagen con un texto manuscrito e Y es una secuencia de etiquetas, expresadas como números enteros, que representa la palabra escrita.

Cabe destacar que no se han tenido en cuenta los signos de puntuación, ni las tildes, dado que nos encontramos ante un problema de transcripción de palabras aisladas y escritas en inglés.

2.3 Imágenes como Tensores.

Dado que el concepto de tensor tiene gran importancia en este proyecto, tanto en el modelado del problema como en la implementación del código, vamos a exponer una pequeña definición y el sentido de su uso.

Un **tensor** es una entidad algebraica de un conjunto de componentes, que generaliza los conceptos de escalar, vector o matriz como tensores de distinto orden. El **orden** de un tensor será el número de índices necesario para especificar sin ambigüedad cada una de sus componentes. La correspondencia se muestra en la siguiente tabla:

Tabla 2–2. Correspondencias de los Tensores.

Entidad Matemática	Ejemplo
Tensor de orden 0, Escalar.	288
Tensor de orden 1, Vector	[2.58 25.89 58.87]
Tensor de orden 2, Matriz	$\begin{bmatrix} 1.54 & 48.6 & 12.5 \\ 84.2 & 15.9 & 78.3 \\ 1.85 & 78.5 & 55.5 \end{bmatrix}$
Tensor de orden 3, “Cubo de elementos”	$\left[\begin{bmatrix} 1 & 6 & 2 \\ 8 & 9 & 3 \\ 5 & 7 & 5 \end{bmatrix} \begin{bmatrix} 4 & 8 & 5 \\ 4 & 5 & 7 \\ 1 & 8 & 5 \end{bmatrix} \begin{bmatrix} 5 & 4 & 1 \\ 2 & 1 & 8 \\ 8 & 5 & 5 \end{bmatrix} \right]$
...	...
Tensor de orden N	...

Un tensor T de orden N^T podría pensarse como una matriz multidimensional de dimensiones $d_0 \times d_1 \times \dots \times d_{N^T-1}$ compuesta de elementos que son referidos como $t_{i_0, i_1, \dots, i_{N^T-1}}$, desde $t_{0,0,\dots,0}$ hasta $t_{d_0-1, d_1-1, \dots, i_{N^T-1}-1}$.

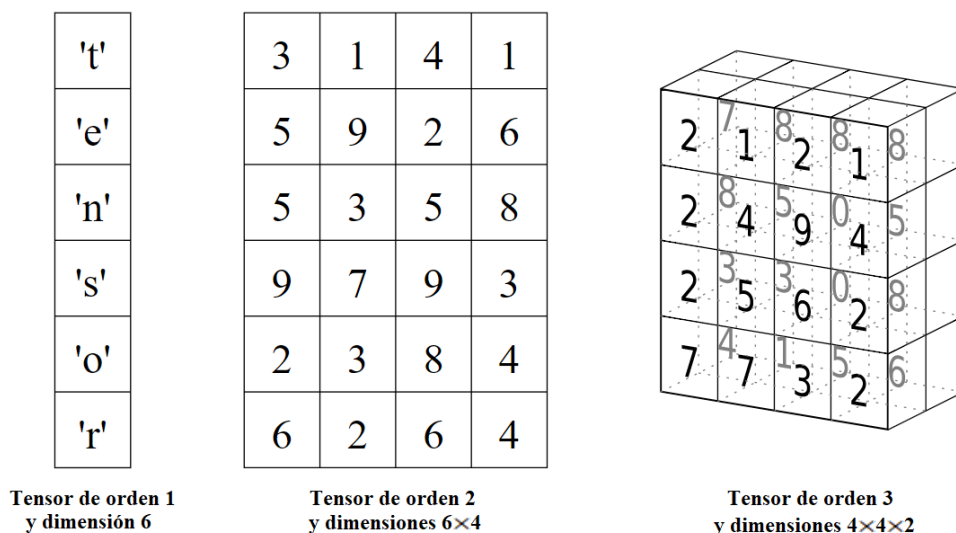


Figura 2-5. Ejemplos de tensores de distinto orden.

Este enfoque nos permite tratar las imágenes y videos como tensores de un determinado orden, concretamente para problemas de visión por computador se suelen tomar las imágenes como tensores de orden 3, donde las dimensiones representan la altura, la anchura y los tres canales RGB que forman el color.

Sin embargo, para los problemas de reconocimiento de caracteres el color de la imagen no suele ser un factor determinante y para reducir la dimensionalidad de la entrada se trabaja con imágenes en escala de grises. Podríamos en este caso representar la imagen como un tensor de orden 2, donde las dimensiones son la altura y la anchura, y sería suficiente, pero ciertas capas de ANN trabajan con una tercera dimensión, por lo que la imagen de entrada al sistema se tratará como un tensor normalizado de orden 3 y dimensiones $64 \times 1024 \times 1$.

3 CAPAS ANN

El presente capítulo comienza con un resumen sobre ANNs como punto de partida para explicar el conjunto de capas que forman la arquitectura de DL del sistema. Se ha puesto especial énfasis en las redes CNN, LSTM y CTC. El objetivo es presentar los fundamentos teóricos que permitan entender la arquitectura diseñada y facilite la interpretación de los resultados obtenidos en los experimentos llevados a cabo.

3.1 Redes Neuronales Artificiales (ANN).

Una ANN es un modelo matemático cuya estructura y funcionamiento está inspirado en las redes neuronales biológicas. Estas redes están formadas por un conjunto de elementos simples, llamados nodos o **neuronas**, interconectados para procesar y transmitir señales de una forma dirigida. Generalmente en las arquitecturas para DL, las ANNs están compuestas por varias **capas de neuronas** que operan en paralelo y propagan la señal desde una capa de entrada hacia una capa de salida atravesando varias capas ocultas, manteniendo solo una dirección posible de transmisión de la información (razón por la cual también se les ha llamado Redes Feedforward).

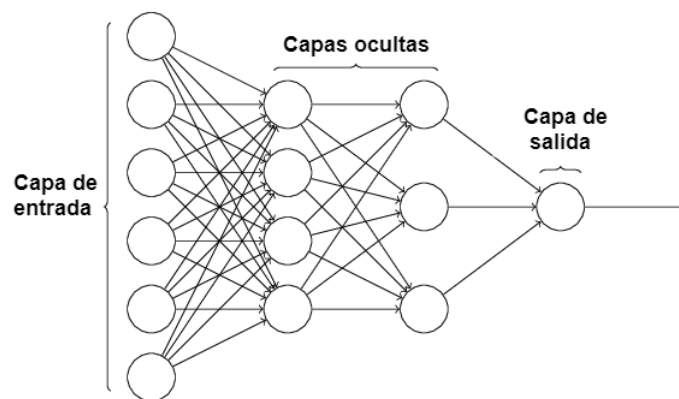


Figura 3-1. Ejemplo de una ANN unidireccional con 4 capas.

Cada neurona consta de unas entradas, con **pesos** asociados a cada entrada, un **umbral** propio de cada neurona, y una **función de activación** que genera una salida a partir de un agregado de las entradas recibidas. La figura 3-2 muestra una versión esquemática explicitando estas partes fundamentales y destacando el paralelismo existente entre el modelo artificial y el biológico en el que se inspira.

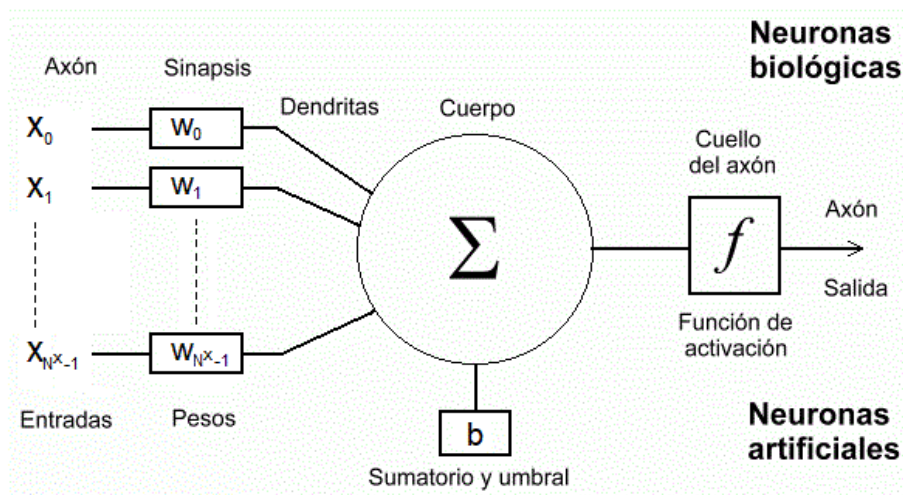


Figura 3-2. Neurona artificial inspirada en la neurona biológica.

La actividad que una neurona artificial realiza en un sistema de este tipo es simple. Consiste en tomar los valores de entrada que recibe de otras neuronas conectadas a ella o de la entrada propia del sistema, multiplicarlos por sus respectivos pesos y sumar estos valores con el umbral para, finalmente, aplicarles una función de activación y obtener la salida. Formalmente podemos expresar la llamada función de transferencia como:

$$y'_i = f(X, W_i, b_i) = f_a \left(\sum_{p=0}^{N^X-1} w_{p,i} x_p + b_i \right) \quad (3-1)$$

Donde X es el conjunto de entradas $[x_0, x_1, \dots, x_{N^X-1}]$, los valores $w_{p,i}$ son los pesos asociados a cada entrada x_p de la neurona i que forman el conjunto W_i , y b_i representa el llamado umbral o bias, que actúa directamente sobre la neurona y es independiente de las entradas (y del resto de neuronas, si las hubiera). Por último, una función de activación representada como f_a , que suele considerarse derivable y no lineal para adaptarse a un número mayor de problemas, aunque no es necesario, con la que se genera una salida y'_i .

Aunque una neurona aislada no parece ser capaz de resolver problemas de una dificultad elevada, la interconexión de ellas siguiendo determinadas topologías permite modelar funciones complejas.

El teorema de Aproximación Universal [23], establece que una sola capa intermedia es suficiente para aproximar, con una precisión arbitraria, cualquier función con un número finito de discontinuidades, siempre y cuando las funciones de activación de las neuronas ocultas sean no lineales. El teorema establece que las redes multicapa no añaden capacidad a menos que la función de activación de las capas sea no lineal.

El resultado de este tipo de topologías, donde la salida de la capa l es la entrada de la capa $l + 1$ tal y como muestra la Figura 3-1, se puede expresar como:

$$\begin{aligned} Y' &= f(H^L, W^Y, B^Y) \\ H^L &= f(H^{L-1}, W^L, B^L) \\ H^{L-1} &= f(H^{L-2}, W^{L-1}, B^{L-1}) \\ &\dots \\ H^0 &= f(X, W^0, B^0) \end{aligned} \quad (3-2)$$

Donde Y' , W^Y y B^Y representan, respectivamente, la salida del sistema, los pesos, y los bias asociados a la capa de salida, mientras que H^l , W^l , B^l representan, respectivamente, la salida, los pesos, y los bias de la capa l , siendo L el número total de capas ocultas.

Como podemos observar este tipo de sistemas es el resultado de encadenar funciones que dependen de una entrada fija y unos parámetros variables. En el caso, indicado anteriormente, de que todas las funciones involucradas sean derivables es posible aplicar algoritmos de aprendizaje basados en el **Descenso del Gradiente** y la **Retropropagación del Error** [24] para conseguir regular estos parámetros en función de su influencia en la respuesta del sistema y su error frente a la salida esperada.

3.1.1 Descenso del Gradiente y Retropropagación del Error.

El procedimiento de aprendizaje es sencillo, se trata de un proceso iterativo donde en cada iteración n se toma un subconjunto de muestras $s \subset S$ de pares (X, Y) y se obtiene la respuesta del sistema Y' para el subconjunto de entradas X , los pesos $W(n)$ y los bias $B(n)$, que se alteran en cada iteración. Con los valores Y y Y' se calcula el error cometido E y se trata de minimizarlo.

Dado que la salida Y' es el resultado de aplicar varias funciones, derivables y no lineales, encadenadas es posible calcular el gradiente de E y obtener los nuevos valores para $W(n+1)$ y $B(n+1)$ reajustándolos adecuadamente en dirección opuesta al gradiente.

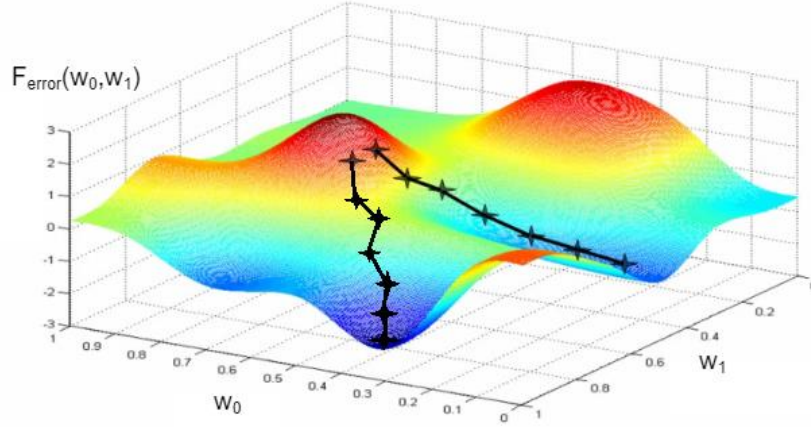


Figura 3-3. Representación del error en función de los pesos w_0 y w_1 para dos posibles trayectorias.

De manera formal podríamos determinar el reajuste de los pesos y los bias como:

$$\begin{aligned} w_{p,i}^l(n+1) &= w_{p,i}^l(n) + \Delta w_{p,i}^l(n) \\ b_i^l(n+1) &= b_i^l(n) + \Delta b_i^l(n) \end{aligned} \quad (3-3)$$

Donde:

$$\begin{aligned} \Delta w_{p,i}^l &= -\gamma \frac{\partial E}{\partial w_{p,i}^l} = -\gamma \frac{\partial E}{\partial Y'} \frac{\partial Y'}{\partial w_{p,i}^l} \\ \Delta b_i^l &= -\gamma \frac{\partial E}{\partial b_i^l} = -\gamma \frac{\partial E}{\partial Y'} \frac{\partial Y'}{\partial b_i^l} \end{aligned} \quad (3-4)$$

Definiendo γ como la **tasa de aprendizaje** que establece la libertad del reajuste en cada iteración de $w_{p,i}^l(n)$ y $b_i^l(n)$, que representan, respectivamente, el peso y el bias de la neurona i de la capa l en la iteración n . Debido a la naturaleza de Y' es posible realizar un cálculo recursivo para obtener $\Delta w_{p,i}^l$ y Δb_i^l mediante:

$$\begin{aligned} \Delta w_{p,i}^l &= -\gamma \frac{\partial E}{\partial w_{p,i}^l} = -\gamma \delta^l \frac{\partial H^l}{\partial w_{p,i}^l} \\ \Delta b_i^l &= -\gamma \frac{\partial E}{\partial b_i^l} = -\gamma \delta^l \frac{\partial H^l}{\partial b_i^l} \end{aligned} \quad (3-5)$$

Donde definimos δ^l como la retropropagación del error de la capa l :

$$\delta^l = \begin{cases} \delta^{l+1} \frac{\partial H^{l+1}}{\partial H^l}, & \text{si } l < L-1 \\ \frac{\partial E}{\partial Y'} \frac{\partial Y'}{\partial H^l}, & \text{si } l = L-1 \end{cases} \quad (3-6)$$

Como podemos observar, el cálculo del reajuste de los pesos y los bias es exactamente el mismo por lo que en adelante solo se expondrán los reajustes para los pesos, y bastará sustituir $w_{p,i}^l$ por b_i^l para obtener el reajuste equivalente de los bias.

Esta es la base para el algoritmo de Retropropagación del Error y tras un proceso de aprendizaje donde el sistema toma miles de muestras, mide el error cometido y reajusta los pesos en la dirección opuesta al gradiente obtenido, se espera minimizar el error y aproximar la función que calcula la red a la función real.

3.2 Redes Neuronales Convolucionales (CNN).

Las CNN son un tipo de ANN inspiradas en la corteza visual primaria del cerebro, especializada en el reconocimiento de patrones procesando información visual de objetos estáticos y en movimiento. Desde un punto de vista computacional, la principal ventaja de este nuevo tipo de redes reside en operar sobre los elementos de un tensor de entrada teniendo en cuenta el valor, la posición y las vecindades de cada uno. Estas operaciones son convoluciones entre el tensor de entrada y tensores propios de la red, llamados filtros. A continuación, y con el fin de exponer su funcionamiento vamos a definir más detalladamente esta operación.

3.2.1 Convolución entre tensores.

En este tipo de convolución, un tensor, llamado **núcleo de convolución**, realiza operaciones sobre porciones de un tensor entrada. Ambos tensores comparten el mismo orden y dichas porciones son subtensores de las mismas dimensiones que el núcleo de convolución. Por lo tanto, para esta convolución es necesaria una entrada, un núcleo de convolución y algún parámetro que dicte cómo se toman los subtensores.

Para formalizar la operación llamaremos X a la entrada y K al núcleo, ambos de orden N y dimensiones $d_0^X \times d_1^X \times \dots \times d_{N-1}^X$ y $d_0^K \times d_1^K \times \dots \times d_{N-1}^K$, respectivamente. Y como último parámetro que establezca la selección de los subtensores, declaramos un **tensor de paso** P , de orden 1, dimensión N y valores $[p_0, p_1, \dots, p_{N-1}]$ enteros y positivos.

Para poder aplicar una convolución es necesario que se cumpla la siguiente relación entre las dimensiones de la entrada y del núcleo:

$$d_i^K \leq d_i^X, \quad \forall i \in [0, 1, \dots, N-1] \quad (3-7)$$

Si esta condición no se cumple en alguna de las dimensiones, se añaden **elementos de relleno** a X aumentando dicha dimensión d_i^X hasta igualar o superar d_i^K . Este relleno se añade en ambos sentidos de la dimensión d_i^X , dejando el tensor original en el “centro”. Estos elementos de relleno suelen tomar el valor 0.

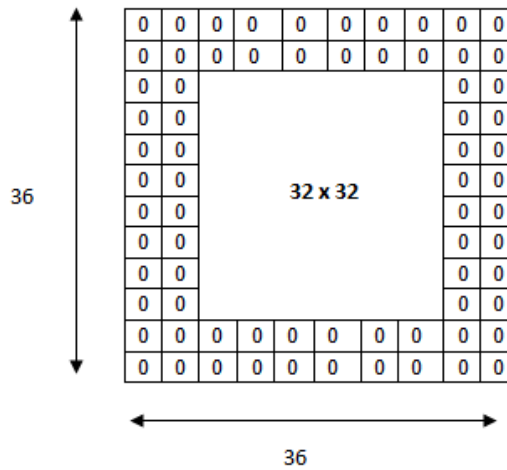


Figura 3-4. Ejemplo de tensor de orden 2 y dimensiones 32x32 extendiendo hasta 36x36.

Cabe destacar que esta operación de relleno no solo es útil para ajustar la relación entre dimensiones, sino que es frecuentemente usada para transformar el tensor X y operar sobre más subconjuntos, extrayendo más información de la entrada.

Tras cumplir este requisito, podemos calcular la convolución $X * K$ y obtener sus elementos como:

$$Y = X * K \quad (3-8)$$

$$y_{i_0, i_1, \dots, i_{N-1}} = \sum_{j_0=0}^{d_0^k-1} \sum_{j_1=0}^{d_1^k-1} \dots \sum_{j_{N-1}=0}^{d_{N-1}^k-1} k_{j_0, j_1, \dots, j_{N-1}} x_{i_0 p_0 + j_0, i_1 p_1 + j_1, \dots, i_{N-1} p_{N-1} + j_{N-1}}$$

Se puede observar que los elementos de Y son el resultado de operaciones sobre porciones de la entrada completa, logrando extraer información local. Tratamos los valores i_k , j_k y p_k como índices para los tensores, donde estos últimos forman el tensor de paso P que dicta cómo se opera sobre la entrada.

El tensor resultante de la convolución mantiene el mismo orden que X y K , pero sus dimensiones vienen determinadas por:

$$d_i^{X*K} = Ent\left(\frac{d_i^X - d_i^K}{p_i} + 1\right) \quad (3-9)$$

Donde $Ent(\cdot)$ es una función que extrae la parte entera y se define como:

$$Ent(x) = \max\{y \in \mathbb{Z} \mid y \leq x\} \quad (3-10)$$

Finalmente, como resultado de una convolución entre dos tensores, tenemos un nuevo tensor del mismo orden que la entrada y los núcleos, cuyos elementos quedan definidos por 3-8 y sus dimensiones por 3-9.

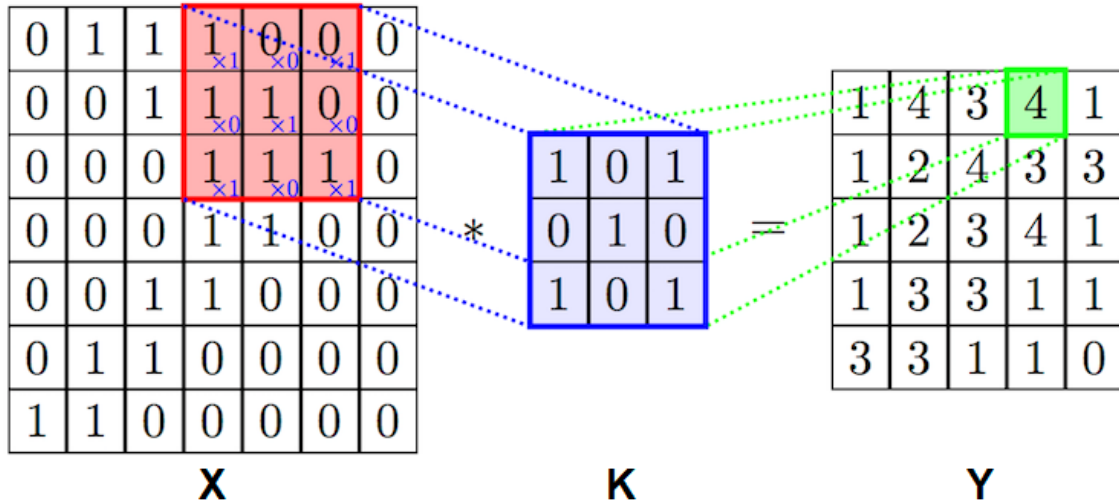


Figura 3-5. Ejemplo de convolución entre tensores de orden 2, con $P = [1,1]$ y sin relleno en X .

Tras definir cómo se realiza la convolución entre tensores, estamos ya en condiciones de exponer el funcionamiento de la capa CNN.

3.2.2 Capa CNN.

Las capas CNN toman como entrada un tensor, realizan la convolución con un conjunto de **filtros**, que actúan como núcleos de convolución, y al resultado le aplica una función de activación. Son generalmente usadas para el procesamiento de imágenes, donde se toma la entrada como un tensor orden 3 para, a través de varias capas, generar un nuevo tensor del mismo orden, pero con más información que la imagen original.

Haciendo una analogía con las ANNs tradicionales, podemos pensar en los filtros de una CNN como tensores formados por los pesos asociados a cada neurona de la capa, pero que procesan la entrada al completo operando por secciones.

Definimos H^{l-1} como el tensor de entrada a la capa l de la CNN, cuya dimensión es $d_0^{H^{l-1}} \times d_1^{H^{l-1}} \times d_2^{H^{l-1}}$, y sobre el cual se aplicarán las convoluciones. Por otro lado, definimos K_i^l como el tensor filtro correspondiente a la neurona i , de dimensión $d_0^{K_i^l} \times d_1^{K_i^l} \times d_2^{K_i^l}$, donde el conjunto de los N^l filtros, concatenados en una nueva dimensión, es tratado como un nuevo tensor K^l de un orden mayor y dimensión $d_0^{K^l} \times d_1^{K^l} \times d_2^{K^l} \times N^l$.

Estas neuronas operan con los filtros de manera independiente sobre la entrada y aplican una función de activación, obteniendo una salida propia H_i^l mediante:

$$H_i^l = f_a(H^{l-1} * K_i^l + b_i^l) \quad (3-11)$$

Donde el valor b_i^l es el bias, ajustándolo se da más o menos relevancia a la salida de la neurona i , y los elementos de $H^{l-1} * K_i^l$ son el resultado de la convolución de K sobre X con un vector de paso P particular de valores $[p_0, p_1, 1]$.

Dado que la dimensión de todas las salidas independientes de la capa l es $d_0^{H^l} \times d_1^{H^l} \times 1$, podemos concatenarlas a lo largo de la tercera dimensión para obtener la salida H^l , cuya dimensión final es $d_0^{H^l} \times d_1^{H^l} \times N^l$. Dicho esto, definimos el cálculo de los elementos de H^l mediante:

$$h_{u,v,i}^l = f_a \left(\sum_{j_0=0}^{d_0^K-1} \sum_{j_1=0}^{d_1^K-1} \sum_{j_2=0}^{d_2^{H^{l-1}}-1} k_{j_0,j_1,j_2,i}^l \cdot h_{u \cdot p_0 + j_0, v \cdot p_1 + j_1, j_2}^{l-1} + b_i^l \right) \quad (3-12)$$

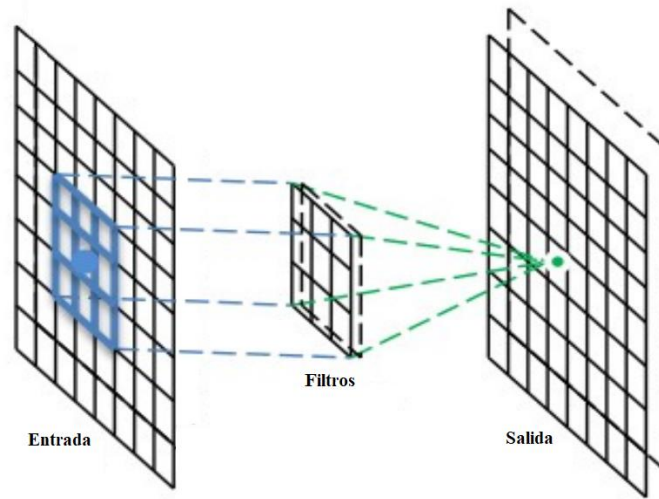


Figura 3-6. Abstracción de una capa aislada de la red CNN.

Aplicando los mismos algoritmos de aprendizaje que para las ANNs clásicas es posible realizar un proceso iterativo que ajusta los elementos de cada filtro y minimiza el error E en cada iteración, aproximando la función propia del sistema a la función real. Definimos el reajuste de cada elemento $k_{j_0,j_1,j_2,i}^l$ de la capa l como:

$$k_{j_0,j_1,j_2,i}^l(n+1) = k_{j_0,j_1,j_2,i}^l(n) + \Delta k_{j_0,j_1,j_2,i}^l(n) \quad (3-13)$$

Donde:

$$\Delta k_{j_0,j_1,j_2,i}^l = -\gamma \frac{\partial E}{\partial k_{j_0,j_1,j_2,i}^l} = -\gamma \delta^l \frac{\partial H^l}{\partial k_{j_0,j_1,j_2,i}^l} \quad (3-14)$$

En este tipo de redes se puede aplicar el algoritmo de Retropropagación del Error de forma análoga a la vista anteriormente, haciendo posible el cálculo recursivo de $\Delta k_{j_0,j_1,j_2,i}^l$ y Δb_i^l de las distintas capas, tal y como ocurre con las ANNs tradicionales.

3.3 Redes Neuronales Recurrentes (RNN).

Las RNN son redes neuronales que presentan **ciclos** por medio de las interconexiones de sus neuronas. La existencia de estos ciclos les permite trabajar de forma natural con secuencias, generalmente temporales, siendo capaces de descubrir regularidades y patrones en ellas.

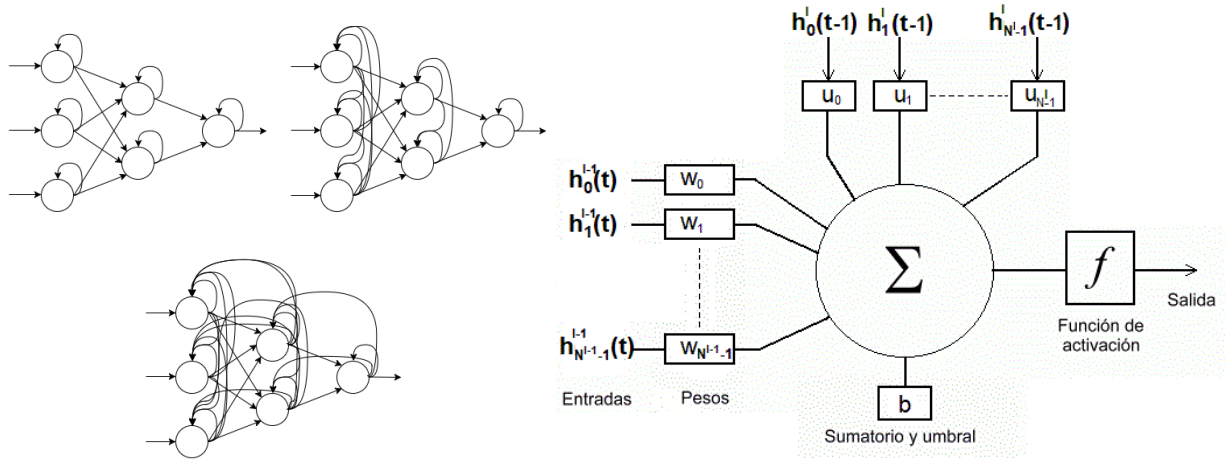


Figura 3-7. Ejemplos de topologías recurrentes y representación de una neurona aislada.

Existen distintas clases de RNNs, según como toma las entradas y genera las salidas, en este caso concreto hablamos de una red *Many to many* donde por cada secuencia de entrada X_{sec} se genera una secuencia de salida Y_{sec} , ambos del mismo tamaño T , y donde cada capa se retroalimenta de sus propias salidas. Podemos pensar que X_{sec} y Y_{sec} están ordenadas cronológicamente y de esta forma tratarlas como una cadena de eventos que tienen correlación en el tiempo.

Estas redes tienen un comportamiento cíclico y en cada ciclo, las neuronas de la capa l reciben la entrada de la capa anterior $H^{l-1}(t)$ y sus propias salidas en el instante anterior $H^l(t-1)$ para generar una nueva salida $H^l(t)$. Descomponiendo $H^{l-1}(t)$ y $H^l(t)$ podemos formalizar la función de transferencia para una neurona i como:

$$h_i^l(t) = f(H^{l-1}(t), H^l(t-1), U_i^l, W_i^l, B_i^l) \quad (3-15)$$

$$h_i^l(t) = f_a \left(\sum_{p=0}^{N^{l-1}-1} w_{p,i}^l h_p^{l-1}(t) + \sum_{q=0}^{N^l-1} u_{q,i}^l h_q^l(t-1) + b_i^l \right)$$

Donde aparecen los pesos $w_{p,i}^l$ y $u_{q,i}^l$, asociados a la entrada habitual $h_p^{l-1}(t)$ y a las nuevas conexiones recurrentes $h_q^l(t-1)$. Como en toda ANN, las neuronas trabajan en paralelo formando **capas recurrentes** y el conjunto de todas las $h_i^l(t)$ forman la salida $H^l(t)$ de toda la red. Desarrollando este proceso para toda la secuencia obtenemos:

$$H^l(t) = f(H^{l-1}(t), H^l(t-1), W^l, U^l, B^l) \quad (3-16)$$

$$H^l(t-1) = f(H^{l-1}(t-1), H^l(t-2), W^l, U^l, B^l)$$

$$H^l(t-2) = f(H^{l-1}(t-2), H^l(t-3), W^l, U^l, B^l)$$

$$H^l(t-3) = f(H^{l-1}(t-3), H^l(t-4), W^l, U^l, B^l)$$

$$\vdots$$

Se observa que la salida de la RNN en el instante t depende de su comportamiento en los instantes anteriores, dando como resultado una función dinámica en el tiempo.

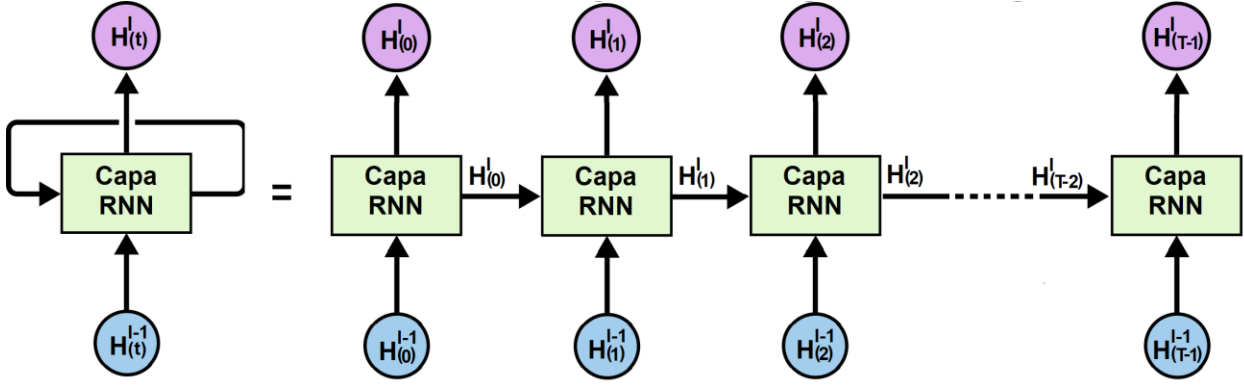


Figura 3-8. Abstracción de una RNN extendida en el tiempo.

Como en todos los tipos de ANNs que se han planteado, tratamos de reducir el error entre la salida real, Y_{sec} , y la salida proporcionada por la red, Y'_{sec} , para cada muestra s_n , que en este caso es un par de secuencias (X_{sec}, Y_{sec}) , dentro de un proceso iterativo. En este caso, definimos un error E_{total} como la suma de los errores cometidos en los distintos instantes t y lo formalizamos como:

$$E_{total} = \sum_{t=0}^{T-1} E(t) = \sum_{t=0}^{T-1} f_{err}(Y(t), Y'(t)) \quad (3-17)$$

Para el cálculo recursivo del reajuste en cada capa es necesario modificar el algoritmo de Retropropagación del Error para convertirlo en lo que se conoce como el algoritmo de **Retropropagación del Error a través del Tiempo** [25][19].

Para definir el reajuste de los pesos $w_{p,i}^l$ y $u_{q,i}^l$ se sigue exactamente el mismo procedimiento por lo que se expone el cálculo únicamente para los pesos w de una capa recurrente l :

$$w_{p,i}^l(n+1) = w_{p,i}^l(n) + \Delta w_{p,i}^l(n) \quad (3-18)$$

Siendo:

$$\Delta w_{p,i}^l = -\gamma \frac{\partial E_{total}}{\partial w_{p,i}^l} = -\gamma \sum_{t=0}^{T-1} \frac{\partial E(t)}{\partial Y'(t)} \sum_{\tau=0}^t \delta^l(\tau) \frac{\partial H^l(\tau)}{\partial w_{p,i}^l} \quad (3-19)$$

Donde declaramos un nuevo $\delta^l(\tau)$ como la retropropagación del error de la capa l para el instante τ :

$$\delta^l(\tau) = \begin{cases} \delta^{l+1}(\tau) \frac{\partial H^{l+1}(\tau)}{\partial H^l(\tau)} + \delta^l(\tau+1) \frac{\partial H^l(\tau+1)}{\partial H^l(\tau)}, & \text{si } l < L \text{ y } \tau < t \\ \delta^l(\tau+1) \frac{\partial H^l(\tau+1)}{\partial H^l(\tau)}, & \text{si } l = L \text{ y } \tau < t \\ \delta^{l+1}(\tau) \frac{\partial H^{l+1}(\tau)}{\partial H^l(\tau)}, & \text{si } l < L \text{ y } \tau = t \\ \frac{\partial Y'(\tau)}{\partial H^l(\tau)}, & \text{si } l = L \text{ y } \tau = t \end{cases} \quad (3-20)$$

3.4 Redes Long Short-Term Memory (LSTM).

Ya hemos visto que las RNN clásicas pueden trabajar como sistemas dinámicos y procesar secuencias, pero cuando las secuencias tienen una longitud elevada se presentan problemas en la propagación de la información. Como ya se ha visto, las capas de una RNN toman como entrada la salida de la capa anterior y su propia salida en el instante anterior, lo que provoca que la relevancia de lo ocurrido en instantes anteriores se pierda con el tiempo.

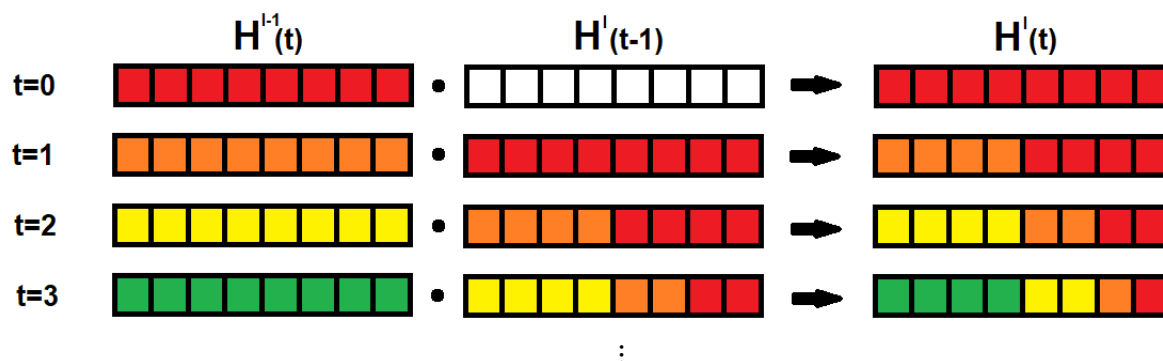


Figura 3-9. Representación simbólica de los efectos del Desvanecimiento del Gradiente.

Este problema se conoce como **Desvanecimiento del Gradiente** y junto al de **Explosión del Gradiente**, con el efecto contrario, frenaron el uso de estas redes hasta 1997, cuando aparecen las primeras redes LSTM que, si bien no consiguen solventar el problema, permiten trabajar con secuencias de una longitud mayor.

La idea es cambiar las neuronas tradicionales por un modelo más complejo, llamado **celda LSTM**, que realiza un mayor número de operaciones en cada iteración. Existen distintos tipos de celdas con diferentes estructuras, pero en este apartado nos centraremos en uno de los modelos más simples de celdas LSTM, ya que han sido las aplicadas en este proyecto.

A diferencia de las neuronas clásicas, en cada iteración las celdas LSTM generan una salida y un **estado interno**, dando lugar a dos tipos de conexiones recurrentes. Existe una recurrencia externa, donde aparecen bucles entre las distintas celdas, y una recurrencia interna, donde cada celda se realimenta con su propio estado interno. Por otro lado, dentro de las celdas se incluyen unas nuevas funciones no lineales, llamadas **puertas**, que regulan el flujo de información de la entrada, la salida y el estado interno de la celda, evitando que el gradiente tienda a desaparecer cuando se entrena con el algoritmo de Retropropagación del Error a través del Tiempo visto anteriormente.

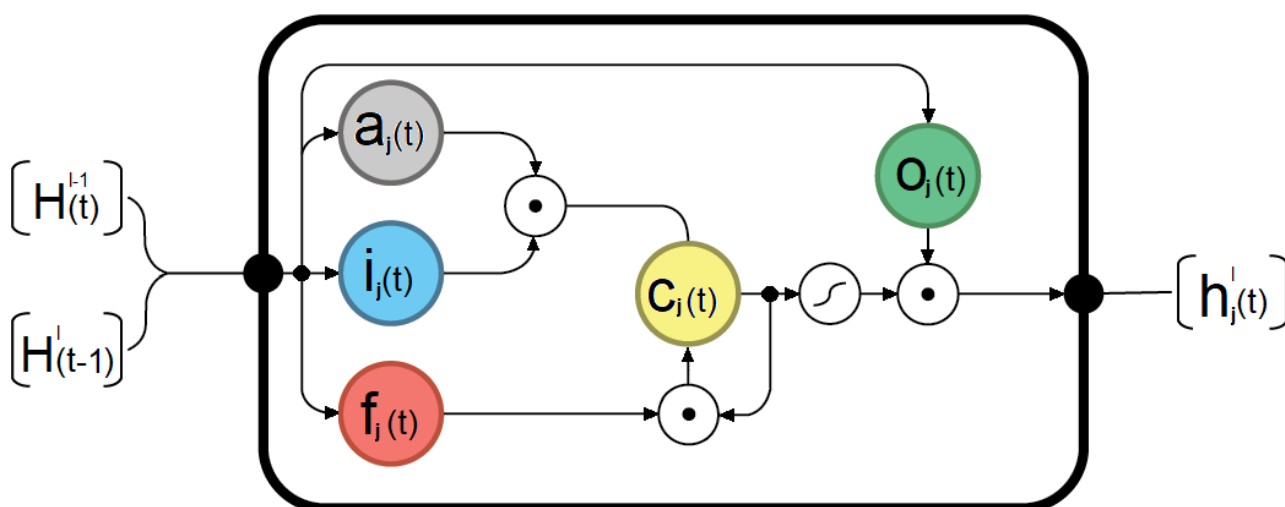


Figura 3-10. Estructura de una celda Modern LSTM.

Estas puertas se implementan utilizando una función no lineal para permitir o negar parcialmente que la información fluya dentro o fuera de la memoria.

Debido a la estructura de las celdas LSTM, la función de transferencia y el reajuste de los pesos tiene una complejidad elevada. Para exponerlo con claridad vamos a dividir la función de transferencia en las funciones relacionadas con las puertas, el estado interno y, finalmente, la salida. Por lo tanto, para una celda j de una capa l podemos definir:

- Activación de Entrada: Función de activación clásica en una red ANNs.

$$a_j^l(t) = f_{a_1}(z_j^{a,l}(t)) \quad (3-21)$$

- Puerta de Entrada: Función que regula el flujo de entrada.

$$i_j^l(t) = f_{a_2}(z_j^{i,l}(t)) \quad (3-22)$$

- Puerta de Olvido: Función que regula el flujo del estado interno pasado.

$$f_j^l(t) = f_{a_2}(z_j^{f,l}(t)) \quad (3-23)$$

- Puerta de Salida: Función que regula el flujo de la salida.

$$o_j^l(t) = f_{a_2}(z_j^{o,l}(t)) \quad (3-24)$$

- Estado Interno: Función que establece el estado para la celda.

$$c_j^l(t) = i_j^l(t)a_j^l(t) + f_j^l(t)c_j^l(t-1) \quad (3-25)$$

- Salida: Salida de la celda en función del estado interno y la puerta de salida

$$h_j^l(t) = f_{a_1}(c_j^l(t))o_j^l(t) \quad (3-26)$$

Donde:

$$z_j^{k,l}(t) = b_j^{k,l} + \sum_{p=0}^{N^{l-1}-1} w_{p,j}^k h_p^{l-1}(t) + \sum_{q=0}^{N^{l-1}-1} u_{q,j}^k h_q^l(t-1), \quad \forall k \in \{a, i, f, o\} \quad (3-27)$$

Y generalmente:

$$f_{a_1}(x) = \tanh(x) \quad (3-28)$$

$$f_{a_2}(x) = \text{sigmoide}(x) \quad (3-29)$$

Como podemos observar en 3-27 aparecen cuatro tipos de pesos y bias, $u_{p,j}^k$, $w_{q,j}^k$ y $b_j^{k,l}$ asociados a las distintas puertas y a la activación de la entrada, y como en toda RNN, se reajustan aplicando la Retropropagación del Error a través del Tiempo. Dada una muestra de secuencias (X_{sec}, Y_{sec}) , definimos el reajuste de w_p^k , $\forall k \in [a, i, f, o]$, de una neurona j de la capa l como:

$$w_{p,j}^{k,l}(n+1) = w_{p,j}^{k,l}(n) + \Delta w_{p,j}^{k,l}(n) \quad (3-30)$$

Donde:

$$\Delta w_{p,j}^{k,l} = -\gamma \frac{\partial E_{total}}{\partial w_{p,j}^{k,l}} = -\gamma \sum_{t=0}^{T-1} \frac{\partial E(t)}{\partial Y'(t)} \sum_{\tau=0}^t \varepsilon_j^{k,l}(\tau) \frac{\partial z_j^{k,l}(\tau)}{\partial w_{p,j}^{k,l}}, \quad \forall k \in \{a, i, f, o\} \quad (3-31)$$

Donde aparece un nuevo $\varepsilon_j^{k,l}(\tau)$ que se define como la retropropagación del error de la puerta k para el instante τ :

$$\varepsilon_j^{k,l}(\tau) = \begin{cases} \varepsilon_j^{c,l}(\tau) \frac{\partial c_j^l(\tau)}{\partial k_j^l(\tau)} \frac{\partial k_j^l(\tau)}{\partial z_j^{k,l}(\tau)}, & \forall k \in \{a, i, f\} \\ \delta^l(\tau) \frac{\partial H^l(\tau)}{\partial k_j^l(\tau)} \frac{\partial k_j^l(\tau)}{\partial z_j^{k,l}(\tau)}, & \text{si } k = o \end{cases} \quad (3-32)$$

Con $\varepsilon_j^{c,l}(\tau)$ como la retropropagación del error del estado interno para el instante τ :

$$\varepsilon_j^{c,l}(\tau) = \begin{cases} \delta^l(\tau) \frac{\partial H^l(\tau)}{\partial c_j^l(\tau)} + \varepsilon_j^{c,l}(\tau + 1) \frac{\partial c_j^l(\tau + 1)}{\partial c_j^l(\tau)}, & \text{si } \tau < t \\ \delta^l(\tau) \frac{\partial H^l(\tau)}{\partial c_j^l(\tau)}, & \text{si } \tau = t \end{cases} \quad (3-33)$$

Como sucedía en el punto anterior, el procedimiento para obtener $u_{p,j}^{k,l}(n + 1)$ y $w_{q,j}^{k,l}(n + 1)$ es el mismo, solo basta con sustituir $u_{p,j}^{k,l}$ por $w_{q,j}^{k,l}$.

3.5 Connectionist Temporal Classification (CTC).

Estas últimas capas son especialmente interesantes ya que permiten encontrar relaciones entre dos secuencias de distinta longitud, logrando clasificar una secuencia de entrada como una secuencia de etiquetas sin necesidad de segmentación. Una peculiaridad de estas capas es que no contiene parámetros, por lo tanto, necesitan de una o varias capas de ANNs previas que operen sobre la secuencia de entrada y permitan optimizar la función objetivo.

La idea es transformar la respuesta de la ANNs en una secuencia de distribuciones de probabilidad sobre las posibles etiquetas, ambas de la misma longitud, y con ella encontrar la secuencia de etiquetas más probable, de longitud menor o igual a la secuencia de entrada.

Gracias a estas distribuciones de probabilidad condicionadas a la entrada de la ANNs, se puede definir una función objetivo derivable para maximizar la probabilidad de la secuencia correcta, de manera que pueda ser entrenada con algoritmos basados en el Descenso del Gradiente.

Veamos los diversos componentes que necesitamos para entender el funcionamiento de este nuevo tipo de redes.

3.5.1 Softmax.

El propósito de la función de activación **Softmax** es transformar las salidas de una red para que sean interpretables como una distribución de probabilidad condicionada sobre un conjunto de variables categóricas. Si tomamos la salida de la última capa H^L como entrada de la capa Softmax, obtenemos la salida Y' mediante:

$$y'_i = \text{Softmax}(h_i^L) = \frac{e^{h_i^L}}{\sum_{k=0}^{N^L-1} e^{h_k^L}} \quad (3-34)$$

Donde N^L expresa el numero de neuronas de la última capa, que corresponde con el número de salidas de la capa Softmax.

3.5.2 De Distribuciones de Probabilidad a Etiquetas.

En una ANN para una tarea de etiquetado de secuencias donde las etiquetas se extraen de un **alfabeto** Λ de tamaño N^A , la última capa H^L debe tener una neurona más que el número de etiquetas que contiene Λ , $N^L = N^A + 1$, ya que esta salida se tomará como entrada de la capa Softmax.

Las primeras N^A activaciones de la capa Softmax son usadas para estimar las probabilidades de observar las etiquetas correspondientes en cada instante, condicionadas a la secuencia de entrada actual, y la activación de la neurona extra estima la probabilidad de observar un “blanco”, o ninguna etiqueta.

En conjunto, las salidas de la capa Softmax representan la probabilidad condicionada de todas las etiquetas en todos los instantes y con este planteamiento, la probabilidad condicionada de cualquier secuencia de etiquetas puede ser encontrada multiplicando las probabilidades de las etiquetas que componen la secuencia.

Formalmente, la activación $y'_k(t)$ de la neurona de salida k en el tiempo t se interpreta como la probabilidad de observar la etiqueta k (o en blanco si $k = N^A + 1$) en el instante t , condicionada por la secuencia de entrada a la ANN. Por lo tanto, dada una secuencia de entrada X_{sec}^T de longitud T podemos estimar:

$$p(\pi | X_{sec}^T) = \prod_{t=0}^{T-1} y'_{\pi_t}(t), \quad \forall \pi \in \Lambda'^T \quad (3-35)$$

Donde Λ'^T es el conjunto de todas las posibles secuencias de longitud T sobre el alfabeto $\Lambda' = \Lambda \cup \{\text{blanco}\}$ y cuyos elementos π quedan definidos como **rut**as.



Figura 3-11. Salida de la capa Softmax, representación simbólica.

Durante este planteamiento hemos supuesto que las salidas de la red en diferentes momentos son condicionalmente independientes, dado el estado interno de la red. Esto se garantiza exigiendo que no existan conexiones de retroalimentación desde la capa de salida a sí misma o a la red.

El siguiente paso es definir una **función de mapeo Many to one** $\mathcal{B}: \Lambda'^T \rightarrow \Lambda^{\leq T}$, para transformar el conjunto de rutas en un conjunto de posibles secuencias de etiquetas $\Lambda^{\leq T}$. Para hacer esto eliminamos primero las etiquetas repetidas consecutivas y luego sólo las etiquetas en ‘blanco’ de las rutas. Por ejemplo, $\mathcal{B}(a - ab -) = \mathcal{B}(-aa - -abb) = aab$.

Finalmente, utilizamos \mathcal{B} para definir la probabilidad condicional de una determinada secuencia de etiquetas $\lambda \in \Lambda^{\leq T}$ como la suma de las probabilidades de todas las rutas donde $\mathcal{B}(\pi) = \lambda$, expresado como:

$$p(\lambda | X_{sec}^T) = \sum_{\pi \in \mathcal{B}^{-1}(\lambda)} p(\pi | X_{sec}^T) \quad (3-36)$$

Este mapeo de diferentes rutas sobre la misma secuencia de etiquetas es lo que permite usar datos no segmentados, ya que elimina el requisito de saber dónde se encuentran los patrones a clasificar en la secuencia de entrada.

3.5.3 Algoritmo CTC forward-backward.

Llegados a este punto, requerimos de una forma eficiente de calcular las probabilidades condicionales $p(\lambda | X_{sec}^T)$ de una secuencia de etiquetas λ . A primera vista esto resultará problemático: es necesaria la suma de $p(\pi | X_{sec}^T)$ sobre todas las rutas correspondientes a λ . Afortunadamente, el problema se puede resolver con un algoritmo iterativo.

La clave es que la suma de las rutas correspondientes a una secuencia de etiquetas puede desglosarse como una suma iterativa sobre las rutas correspondientes a los prefijos de dicha secuencia de etiquetas. Las iteraciones pueden entonces ser calculadas eficientemente con variables recursivas hacia adelante y hacia atrás.

Para cualquier secuencia q de longitud r , $q_0 q_1 \dots q_{r-1}$, denotamos $q_{0:p-1}$ y $q_{r-p:r-1}$ como sus primeros y últimos p elementos, respectivamente. Para un etiquetado λ , definimos la variable recursiva **forward** $\alpha_p(t)$ como la probabilidad total de $\lambda_{0:p-1}$ en el momento t :

$$\alpha_p(t) = \sum_{\substack{\pi \in \Lambda'^T: \\ \mathbb{B}(\pi_{0:t}) = \lambda_{0:p-1}}} \prod_{\tau=0}^t y'_\pi(\tau) \quad (3-37)$$

Como podemos ver, $\alpha_p(t)$ puede ser calculado recursivamente desde $\alpha_{p-1}(t-1)$ y $\alpha_p(t-1)$.

Para permitir etiquetas en “blanco” en las rutas de salida, consideramos una secuencia de etiquetas modificada λ' , con etiquetas en blanco añadidos al principio y al final e insertados entre cada par de etiquetas. La longitud de λ' es por lo tanto $2|\lambda| + 1$. En el cálculo de las probabilidades de los prefijos de λ' , permitimos todas las transiciones entre etiquetas en blanco y reales, y también entre cualquier par de etiquetas reales. Forzamos todos los prefijos para que empiecen con una etiqueta en blanco (b) o el primer símbolo en λ , λ_0 . Esto nos fija las siguientes reglas para la inicialización:

$$\begin{aligned} \alpha_0(0) &= y'_b(0) \\ \alpha_1(0) &= y'_{\lambda_0}(0) \\ \alpha_p(0) &= 0, \quad \forall p > 1 \end{aligned} \quad (3-38)$$

Para la recurrencia:

$$\alpha_p(t) = \begin{cases} \alpha'_p(t) y'_{\lambda'_p}(t), & \text{si } \lambda'_p = b \text{ o } \lambda'_{p-2} = \lambda'_p \\ (\alpha'_p(t) + \alpha_{p-2}(t-1)) y'_{\lambda'_p}(t), & e. o. c. \end{cases} \quad (3-39)$$

Donde:

$$\alpha'_p(t) = \alpha_p(t-1) + \alpha_{p-1}(t-1) \quad (3-40)$$

La probabilidad de λ es entonces la suma de las probabilidades totales de λ' con y sin la etiqueta en “blanco” final en el instante $T-1$.

$$p(\lambda | X_{sec}^T) = \alpha_{|\lambda'|-1}(T-1) + \alpha_{|\lambda'|-2}(T-1) \quad (3-41)$$

Del mismo modo, definimos la variable recursiva **backward** $\beta_p(t)$ como la probabilidad total de $\lambda_{p:|\lambda|-1}$ en el momento t :

$$\beta_p(t) = \sum_{\substack{\pi \in \Lambda'^T: \\ \mathbb{B}(\pi_{t:T-1}) = \lambda_{p:|\lambda|-1}}} \prod_{\tau=t}^{T-1} y'_\pi(\tau) \quad (3-42)$$

$$\begin{aligned} \beta_{|\lambda'|-1}(T-1) &= y'_b(T-1) \\ \beta_{|\lambda'|-2}(T-1) &= y'_{\lambda'_{|\lambda'|-1}}(T-1) \\ \beta_p(T-1) &= 0, \quad \forall p < |\lambda'|-2 \end{aligned} \quad (3-43)$$

Para la recurrencia:

$$\beta_p(t) = \begin{cases} \beta'_p(t)y'_{\lambda'_p}(t), & \text{si } \lambda' = b \text{ o } \lambda'_{p+2} = \lambda'_p \\ \left(\beta'_p(t) + \beta_{p+2}(t+1) \right) y'_{\lambda'_p}(t), & \text{e. o. c.} \end{cases} \quad (3-44)$$

Donde:

$$\beta'_p(t) = \beta_p(t+1) + \beta_{p+1}(t+1) \quad (3-45)$$

Y análogamente:

$$p(\lambda|X_{sec}^T) = \beta_0(0) + \beta_1(0) \quad (3-46)$$

3.5.4 Medida de Error y Función objetivo.

Hasta ahora hemos descrito cómo una ANN con una capa de Softmax y una función de mapeo se puede utilizar para la clasificación en secuencias de etiquetas, pero es necesaria una función objetivo que permita optimizar los parámetros de la red.

Generalmente para entrenar redes con algoritmos basados en Descenso del Gradiente se utilizan funciones derivables que miden el error entre la salida dada por el sistema y la salida ideal, como hemos ido observando a lo largo del trabajo. Pero en nuestro caso concreto esta medida no se obtiene mediante una función derivable y hace imposible la aplicación de este procedimiento.

3.5.4.1 Label Error Rate (LER).

Para los objetivos que perseguimos en este proyecto nos interesa la siguiente medida de error: dado una secuencia objetivo $Y_{SEC}^{T_1}$ y una secuencia de salida $Y_{SEC}'^{T_2}$ generada por el sistema, donde las longitudes T_1 y T_2 pueden ser iguales o no, definimos la **tasa de error de etiquetas** (LER) entre dos secuencias como:

$$LER(Y_{SEC}^{T_1}, Y_{SEC}'^{T_2}) = \frac{ED(Y_{SEC}^{T_1}, Y_{SEC}'^{T_2})}{T_1} \quad (3-47)$$

Donde $ED(p, q)$ es la distancia de edición, o distancia de Levenshtein, entre dos secuencias p y q (que se define como el número mínimo de inserciones, sustituciones y eliminaciones necesarias para transformar p en q). Se trata de una medida frecuente para este tipo de tareas (como el reconocimiento del habla o la escritura a mano, en las que hay involucradas cadenas de símbolos), ya que permite medir la tasa de errores de transcripción.

En este tipo de problemas, es evidente que este error ofrece una medida de la eficiencia del sistema, pero tiene el inconveniente de que no se obtiene mediante una función derivable, lo que imposibilita que pueda ser usada como función objetivo para el entrenamiento del sistema siguiendo los mecanismos apuntados en las secciones anteriores.

3.5.4.2 Función de Máxima Probabilidad.

Como ya hemos comentado, necesitamos alguna medida obtenida mediante una función derivable que contenga información sobre la eficiencia del sistema y, simultáneamente, permita aplicar algoritmos de entrenamiento basados en el Descenso del Gradiente. Podríamos pensar en una función objetivo que permita maximizar la probabilidad de la secuencia correcta $Y_{SEC}^{\leq T}$ dada una secuencia de entrada X_{SEC}^T .

$$\text{Maximizar } p(Y_{SEC}^{\leq T} | X_{SEC}^T) \quad (3-48)$$

Que es lo que se conoce como **función de verosimilitud**, y su maximización permite optimizar los parámetros de la ANN que hemos situado antes de la capa CTC en nuestra arquitectura.

Para poder aplicar algoritmos de entrenamiento basados en el Descenso del Gradiente es necesario que se persiga minimizar una determinada función, ya que se realiza una búsqueda de mínimos locales y por lo tanto necesitamos negar la función de verosimilitud. La maximización anterior se puede convertir en una minimización si trabajamos con:

$$\text{Minimizar } -\ln(p(Y_{SEC}^{\leq T}|X_{SEC}^T)) \quad (3-49)$$

Que, además de convertir los máximos en mínimos, ofrece la ventaja de que trabajar con logaritmos reduce el riesgo de desbordamiento en el análisis numérico, permite interpretar su resultado como un “error” tomando el valor 0 para una probabilidad absoluta de secuencia correcta y, por último y quizás más importante, convierte un producto de factores en una suma de factores.

Para una secuencia de etiquetas $\lambda = Y_{SEC}^{\leq T}$ podemos establecer:

$$\ln(p(\lambda|X_{SEC}^T)) = \sum_{t=0}^{T-1} \ln \left(\sum_{p=0}^{|\lambda'|-1} \alpha_p(t) \right) \quad (3-50)$$

El punto clave es que para una determinada secuencia de etiquetas λ , el producto de las variables forward y backward en un determinado p y t es la probabilidad de todas las rutas π correspondientes a λ que pasan por la etiqueta p en el tiempo t . Formalmente, tenemos:

$$\alpha_p(t)\beta_p(t) = \sum_{\substack{\pi \in B^{-1}(\lambda): \\ \pi_t = \lambda'_p}} y'_{\lambda'_p}(t) \prod_{t=0}^{T-1} y'_{\pi_t}(t) \quad (3-51)$$

Manipulando adecuadamente la expresión anterior y reorganizando y sustituyendo a partir de 3-35 obtenemos:

$$\frac{\alpha_p(t)\beta_p(t)}{y'_{\lambda'_p}(t)} = \sum_{\substack{\pi \in B^{-1}(\lambda): \\ \pi_t = \lambda'_p}} p(\pi|X_{SEC}^T) \quad (3-52)$$

Que se corresponden con la porción de la probabilidad total $p(\lambda|X_{SEC}^T)$ debido a las rutas que pasan por λ'_p en el instante t . Por lo tanto, para cualquier t , podemos sumar sobre todos los p para obtener:

$$p(\lambda|X_{SEC}^T) = \sum_{p=0}^{|\lambda'|-1} \frac{\alpha_p(t)\beta_p(t)}{y'_{\lambda'_p}(t)} \quad (3-53)$$

Para derivar $p(\lambda|X_{SEC}^T)$ con respecto a $y'_k(t)$ sólo tenemos que considerar aquellas trayectorias que pasan por la etiqueta k en el instante t . Debido a que la misma etiqueta puede repetirse varias veces para una sola secuencia de etiquetas λ , definimos el conjunto de posiciones donde la etiqueta k aparece como $lab(\lambda, k) = \{p: \lambda'_p = k\}$, que puede estar vacío. Luego derivamos para obtener:

$$\frac{\partial p(\lambda|X_{SEC}^T)}{\partial y'_k(t)} = \frac{1}{\partial y'_k(t)^2} \sum_{p \in lab(\lambda, k)} \alpha_p(t)\beta_p(t) \quad (3-54)$$

Observando que:

$$\frac{\partial \ln(p(\lambda|X_{SEC}^T))}{\partial y'_k(t)} = \frac{1}{p(\lambda|X_{SEC}^T)} \frac{\partial p(\lambda|X_{SEC}^T)}{\partial y'_k(t)} \quad (3-55)$$

Por último, para la retropropagación del gradiente a través de la capa de Softmax, definido previamente como $\delta^L(t)$, necesitamos las derivadas de la función objetivo con respecto a las salidas $h_k^L(t)$ de la ANN previa.

Si nos ayudamos del algoritmo CTC forward-backward, obtenemos:

$$\frac{\partial \ln(p(\lambda|X_{SEC}^T))}{\partial h_k^L(t)} = y'_k(t) - \frac{1}{y'_k(t)Z_t} \sum_{p \in lab(z,k)} A_p(t)B_p(t) \quad (3-56)$$

Donde:

$$\begin{aligned} A_p(t) &= \frac{\alpha_p(t)}{\sum_{i=0}^{|\lambda'|-1} \alpha_i(t)} \\ B_p(t) &= \frac{\beta_p(t)}{\sum_{i=0}^{|\lambda'|-1} \beta_i(t)} \\ Z_t &= \sum_{p=0}^{|\lambda'|-1} \frac{A_p(t)B_p(t)}{y'_{\lambda'_p}(t)} \end{aligned} \quad (3-57)$$

La ecuación 3-56 es la retropropagación del “error” de la última capa de la ANN y será recibida por la red durante el entrenamiento.

4 MODELO ANN

El sistema DL propuesto en este proyecto está compuesto por un conjunto de diferentes arquitecturas de redes neuronales, cada una cumpliendo una función específica, y que junto con la capa CTC consiguen transcribir palabras manuscritas con una cierta eficiencia. Todo el sistema está implementado en Python haciendo uso de la librería TensorFlow, y su código queda expuesto en el Anexo I y subido a un repositorio público [26].

La arquitectura global del sistema está basada en dos ideas principales: (i) obtener la mayor cantidad de información posible de la imagen, y (ii) ayudarnos de esta información para calcular la secuencia de etiquetas más probable.

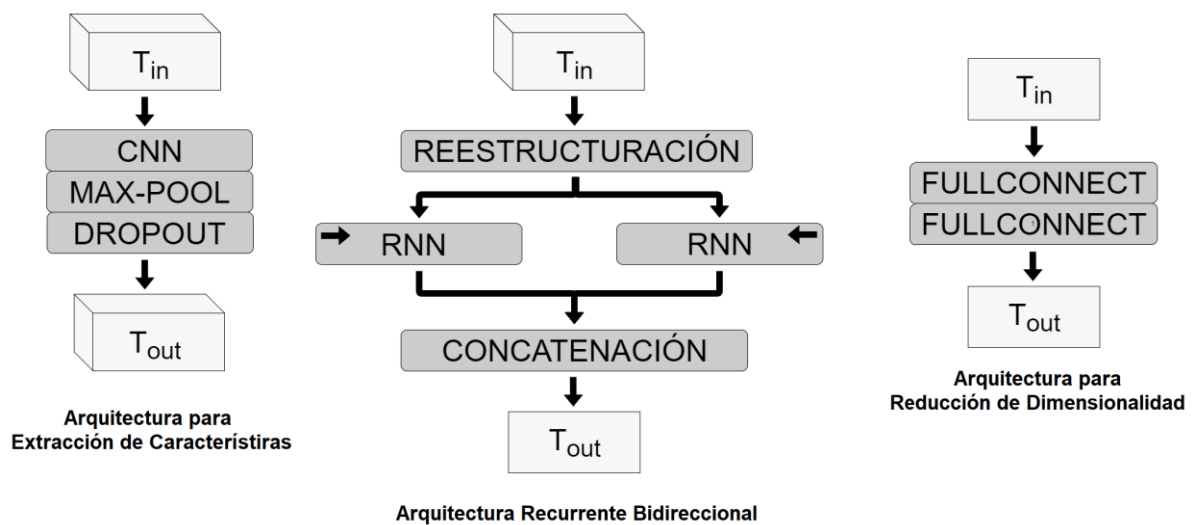


Figura 4-1. Arquitecturas que componen el modelo.

Para cumplir el primer objetivo y procesar la imagen en busca de información se han propuesto 3 arquitecturas que trabajan en serie sobre la imagen.

4.1 Arquitectura para Extracción de Características (AEC).

Esta arquitectura consigue establecer una relación entre los elementos de la salida y la información local de los elementos de la entrada. Está formada por 3 capas:

- CNN: Capa convolucional que permite establecer una relación entrada-salida fijada por unos parámetros ajustables.
- Max-Pool [27]: Capa para reducir la dimensionalidad basada en el muestreo de la entrada. El objetivo es muestrear la entrada tomando subconjuntos de elementos y seleccionando como salida el valor más elevado. Gráficamente lo podemos ver como:

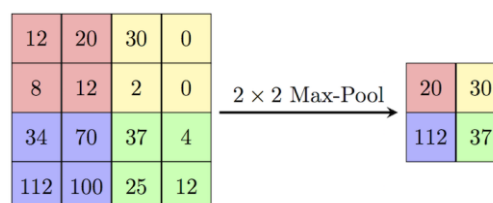


Figura 4-2. Max-Pool.

- Dropout [28]: El objetivo de esta última capa es “añadir ruido” durante el entrenamiento de la ANN, anulando, bajo una cierta probabilidad, las salidas de la capa anterior. En un primer momento este procedimiento parece ser perjudicial para el aprendizaje de la red, pero evita en cierta medida el

problema de sobreentrenamiento, cuando el sistema se adapta muy bien a las muestras que ya ha procesado, pero responde muy mal ante muestras desconocidas (evitando el objetivo principal de todo aprendizaje que es la capacidad de generalización del modelo obtenido a partir de la muestra parcial con la que se entrena).

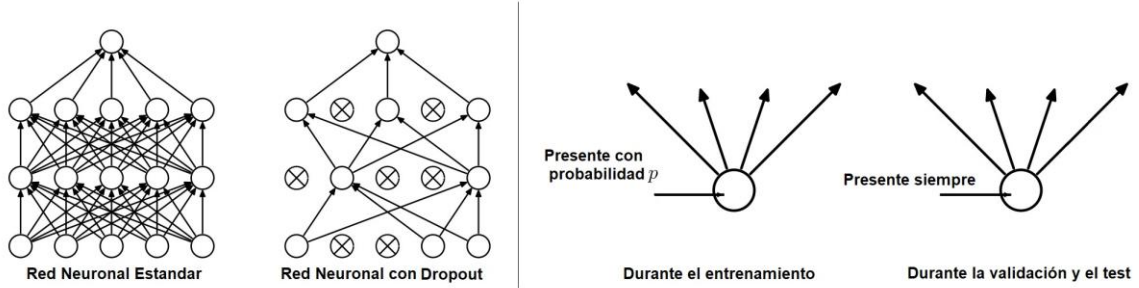


Figura 4-3. Dropout.

Esta arquitectura es frecuentemente usada para el procesamiento de imágenes y generalmente obtiene buenos resultados en tareas de visión artificial.

4.2 Arquitectura Recurrente Bidireccional (ARB).

La segunda arquitectura considerada para este proyecto recorre un tensor entrada a lo largo de una única dimensión, pero en ambos sentidos, para generar una salida que contenga información sobre la relación que existe entre los elementos de la entrada a lo largo de dicha dimensión. Los elementos que componen esta arquitectura son:

- **Reestructuración:** Función que reorganiza un tensor de entrada suprimiendo una dimensión. El tensor de entrada es el resultado devuelto por la AEC expuesta en la sección anterior, por tanto, por cada imagen, tenemos como entrada un tensor de orden 3 cuyas dimensiones son *altura* × *anchura* × *nº de filtros* y se reestructura en un tensor de orden 2 con dimensiones *altura* · *nº de filtros* × *anchura*.
- **Capas Recurrentes:** Dos capas RNN que toman el tensor resultado de la reestructuración y lo recorren a lo largo de la dimensión que corresponde a la *anchura*. Cada capa recorre el tensor en una dirección dando como resultado dos tensores.
- **Concatenación:** Función para concatenar las salidas de las capas RNN. Esta concatenación se hace a lo largo de la dimensión que corresponde a la *altura* · *nº de filtros*

Con este planteamiento se trata de “leer” el tensor resultado de la AEC, que contiene más información que la imagen de entrada, como lo haríamos los humanos con la imagen de entrada. Esta lectura se realiza en ambas direcciones para extraer la relación de los elementos en los dos sentidos.

4.3 Arquitectura para Reducción de Dimensionalidad (ARD).

Tras las capas anteriores se añaden dos capas Fullconect (o densas, capas ANN tradicionales) para reducir la dimensionalidad y dando como resultado un tensor de orden 2 con dimensiones $d_0 \times d_1$:

$$d_0 = \frac{\text{anchura de la imagen original}}{2^5} \quad (4-1)$$

$$d_1 = (n^\circ \text{ de etiquetas} + 1)$$

De manera que la capa CTC toma como entrada una secuencia de longitud d_0 con una variable más que el número de etiquetas existentes, $d_1 = N^A + 1$.

Esta última capa nos permite operar con una función objetivo y entrenar la red completa para encontrar la

relación entre la imagen de entrada y la palabra escrita o, dicho de otro modo, entre la matriz de entrada y la secuencia de salida.

4.4 Arquitectura Global.

El sistema global propuesto está compuesto por la siguiente arquitectura en serie de redes neuronales:

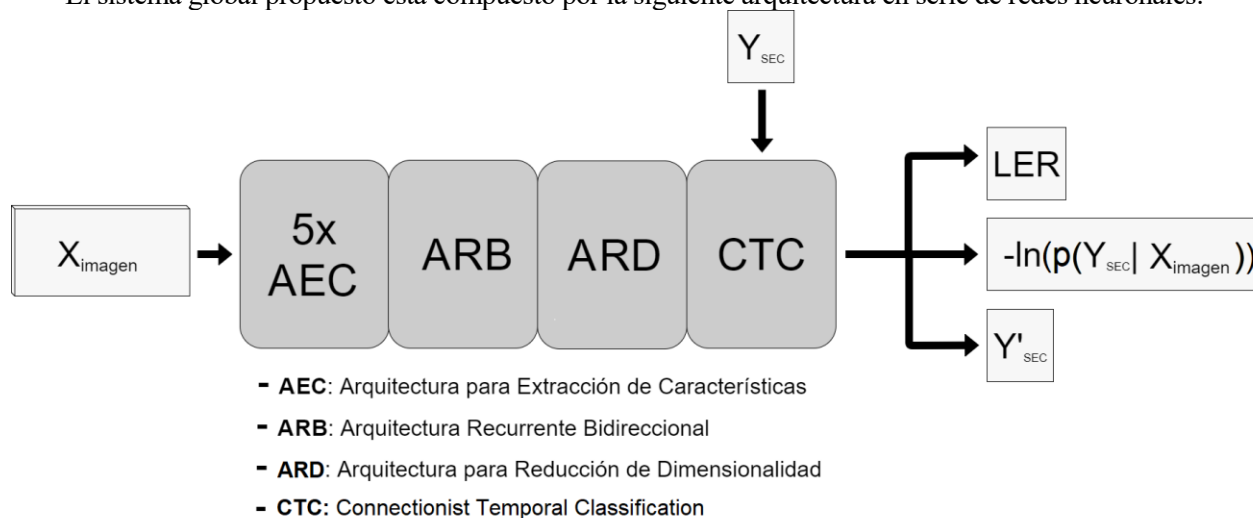


Figura 4-4. Arquitectura Global.

- 5xAEC en serie: Procesan la imagen de entrada extrayendo la mayor cantidad de información posible.
- ARB: Lee el tensor resultante de la etapa anterior extrayendo información a lo largo de la dimensión que corresponde a la anchura de la imagen.
- ARD: Transforma toda la información en una secuencia que será procesada por la capa CTC.
- CTC: Calcula el error cometido respecto a la palabra esperada, el valor de la función objetivo para la muestra dada y la secuencia de salida más probable, que tomaremos como salida del sistema.

Debemos indicar que en el presente capítulo, y con el fin de facilitar la comprensión de la tarea que lleva a cabo esta arquitectura, nos hemos centrado en el tratamiento de una única imagen, pero en la implementación se trabaja en paralelo con lotes de muestras, calculando el error para cada una de ellas y ajustando los pesos de la red en consecuencia.

Por lo tanto, el tensor de entrada al sistema es de orden 4 y dimensiones $n^{\circ} \text{ de imágenes} \times \text{altura} \times \text{anchura} \times 1$ y el resto de tensores son de un orden mayor, reservando la dimensión extra para el número de muestras.

La siguiente tabla muestra los valores de los hiperparámetros con los que se ha evaluado el sistema:

Tabla 4-1. Hiperparámetros del modelo.

Hiperparámetro	Valor	Hiperparámetro	Valor	Hiperparámetro	Valor
Nº de Filtos CNN-1	20	Nº de Filtos CNN-5	400	Neuronas de las RNNs	200
Nº de Filtos CNN-2	50	Dimensión de los filtros	5x5	Neuronas primera capa de la ARD	200
Nº de Filtos CNN-3	100	Ventana Max-Pool	2x2	Número de clases	63
Nº de Filtos CNN-4	200	Probabilidad de Dropout	0.5	Longitud secuencia entrada a la CTC	32

5 PRUEBAS Y RESULTADOS

Sobre el sistema se han realizado dos pruebas para evaluar su rendimiento en la transcripción de palabras manuscritas: (1) una validación cruzada, y (2) el test propuesto por el equipo que construyó la IAM Handwriting Database. En las siguientes secciones mostraremos un extracto de los resultados obtenidos en estas pruebas. Al igual que el código desarrollado para este proyecto, los resultados están disponibles en el repositorio público asociado a este trabajo.

5.1 Validación Cruzada (Cross-Validation).

Como suele ser habitual, para la validación cruzada se ha tomado el dataset proporcionado por la IAM Handwriting Database y se han realizado 10 evaluaciones independientes, con distintas particiones para entrenamiento y validación. En cada evaluación, se ha entrenado el sistema con el 90% de las muestras del dataset y se ha validado con el 10% restante (por supuesto, los datasets de entrenamiento y validación son disjuntos).

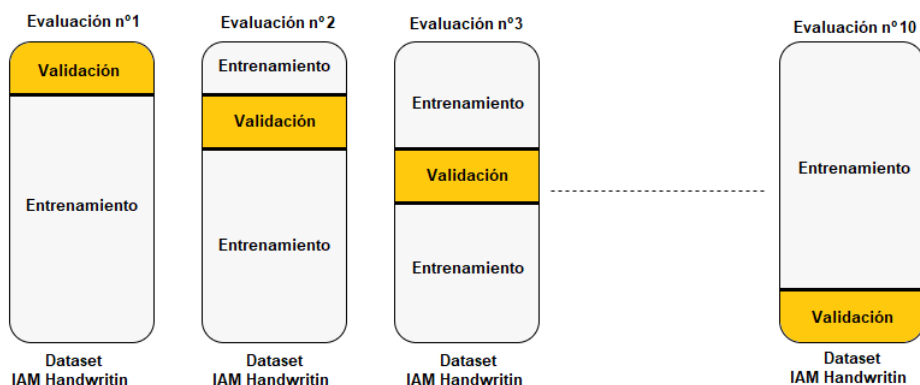


Figura 5-1. Representación del dataset en las distintas evaluaciones.

El entrenamiento del sistema se ha realizado para 20.000 épocas, donde en cada una se ha tomado un lote de 500 muestras del conjunto de entrenamiento para el ajuste de los pesos, intentando disminuir el error cometido. Cada 50 épocas se ha calculado el valor medio del LER y de la función objetivo, llamado Coste, para las 500 muestras del entrenamiento y para el conjunto de validación al completo.

El resultado de cada evaluación se ha exportado como CSV y tras las 10 evaluaciones se pueden realizar algunas observaciones. Para representarlo gráficamente se han unificado todos los CSV, calculando la media aritmética de estas medidas en cada época.

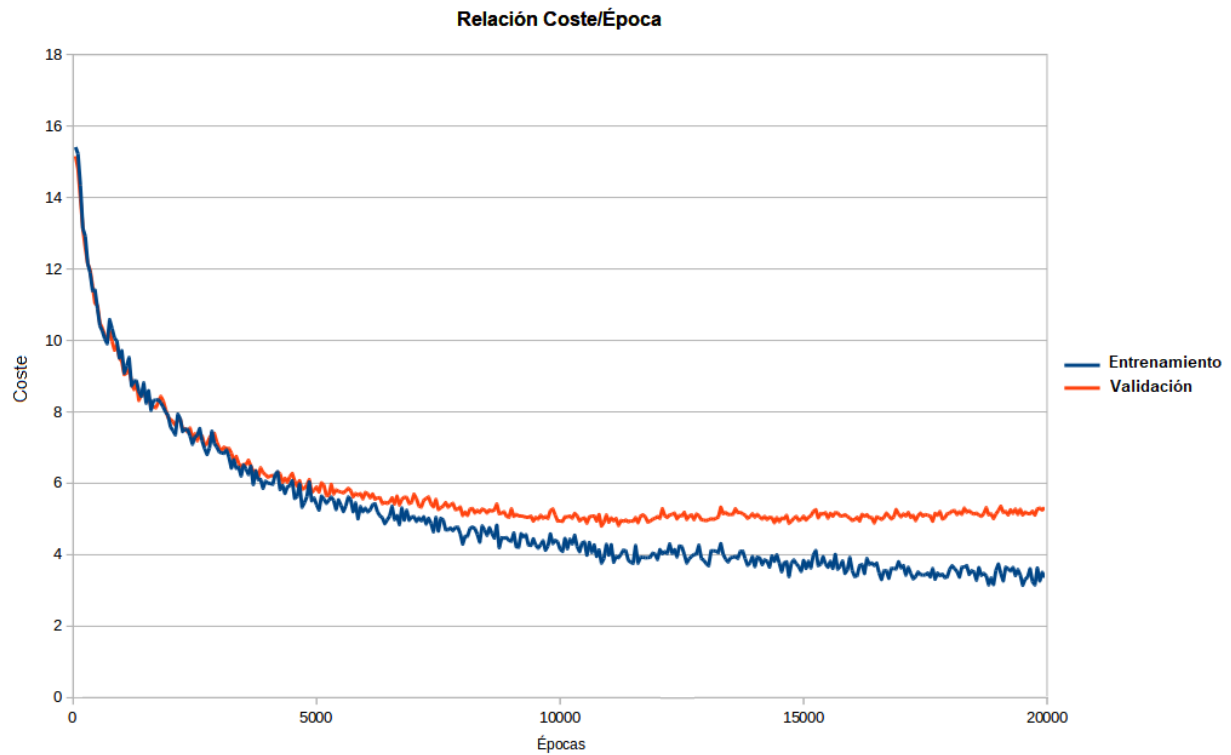


Figura 5-2. Resultados del Coste para la validación cruzada, representados en media.

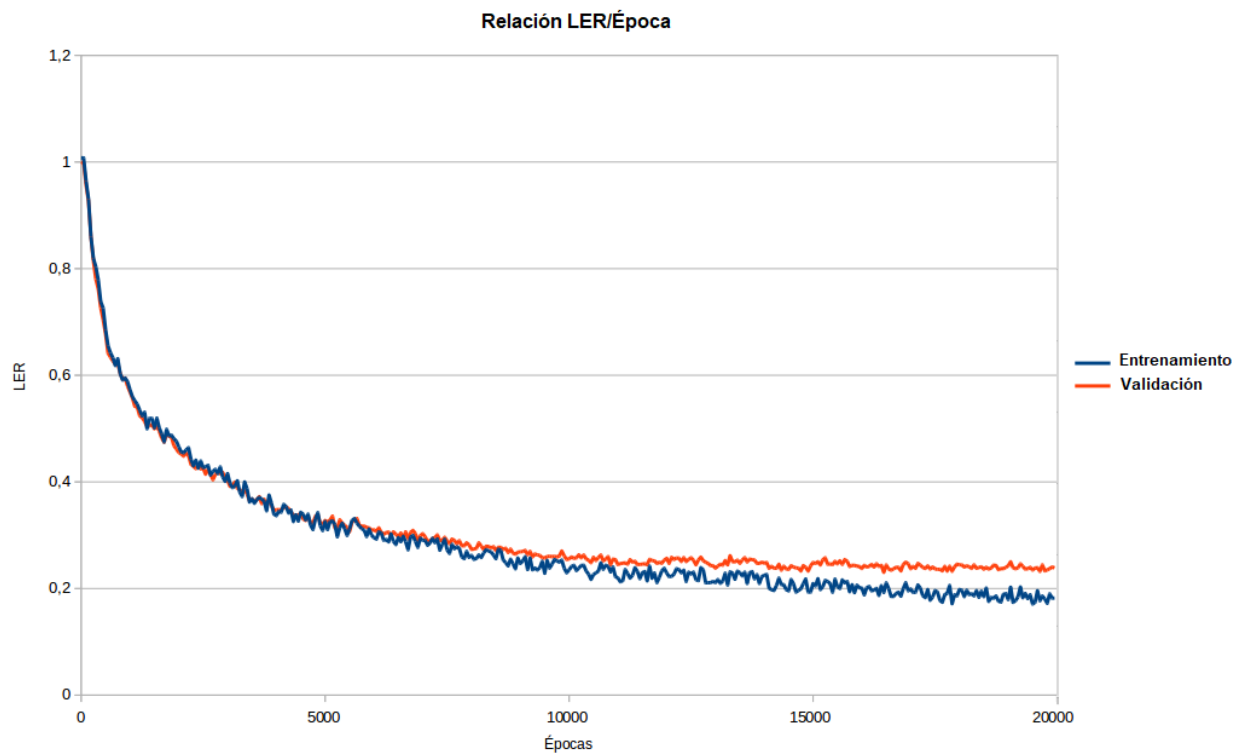


Figura 5-3. Resultados del LER para la validación cruzada, representados en media.

Por un lado, se observa una relación entre el LER y el Coste, ambas disminuyen de forma parecida durante el transcurso de las épocas, sin embargo, no podemos afirmar que los mínimos de ambos valores coincidan en la misma época.

Tabla 5–1. Valores mínimos para el LER y el Coste.

Época	LER	Coste
Coste mínimo en validación. Época 10850	Validación: 0.245 Entrenamiento: 0.231	Validación: 4.804 Entrenamiento: 3.147
LER mínimo en validación. Época 16450	Validación: 0.230 Entrenamiento: 0.193	Validación: 4.890 Entrenamiento: 3.700

También podemos apreciar cómo el sistema responde mejor ante muestras conocidas que ante muestras desconocidas, y cómo poco a poco se produce un sobreentrenamiento, obteniendo peores resultados para el conjunto de validación. Este suceso se hace más evidente para la medida del Coste.

Este tipo de técnica es un proceso costoso, pero es ampliamente utilizada para evaluar los resultados de un sistema e intentar generalizarlos a todo el conjunto de posibles casos.

5.2 Test IAM.

En esta segunda prueba se ha realizado un test propuesto por el equipo que trabajó en la construcción de la *IAM Handwriting Database*, en el que se define una distribución específica del dataset para poder comparar distintos modelos bajo las mismas condiciones. Se proporciona un conjunto de entrenamiento, dos de validación y uno último para el testeo del sistema.

Sin embargo, esta prueba ha tenido que ser adaptada ya que está diseñada para modelos de reconocimiento de líneas completas, que no es nuestro caso. Para ello, se han extraído las palabras de cada línea en los distintos conjuntos y se ha respetado la misma distribución para el dataset de palabras aisladas.

Para realizar este test se ha entrenado el sistema durante 50.000 épocas, con lotes de 500 muestras, y se han realizado validaciones cada 50 épocas, exportando el resultado en CSV como hacíamos en el punto anterior. Las validaciones se producen sobre los dos conjuntos de validación, calculando la media aritmética del LER y tratando de encontrar un mínimo.

Al comenzar el entrenamiento se establece un LER Mínimo base de 1, se podría interpretar como un 100% de error, y en cada validación se compara con el LER obtenido por el sistema, si el LER del sistema es menor se guardan los parámetros del sistema y se actualiza el LER Mínimo.

Tras realizar todo el entrenamiento tenemos como resultado 3 ficheros CSV que contienen los valores del LER y el Coste para los conjuntos de entrenamiento y validación. Por otro lado, también se han guardado los valores de los parámetros que han demostrado un mejor rendimiento para el LER, lo que llamaríamos “la máquina entrenada”. Unificamos nuevamente el resultado de las validaciones para representarlo en una gráfica y facilitar la extracción de algunas conclusiones.

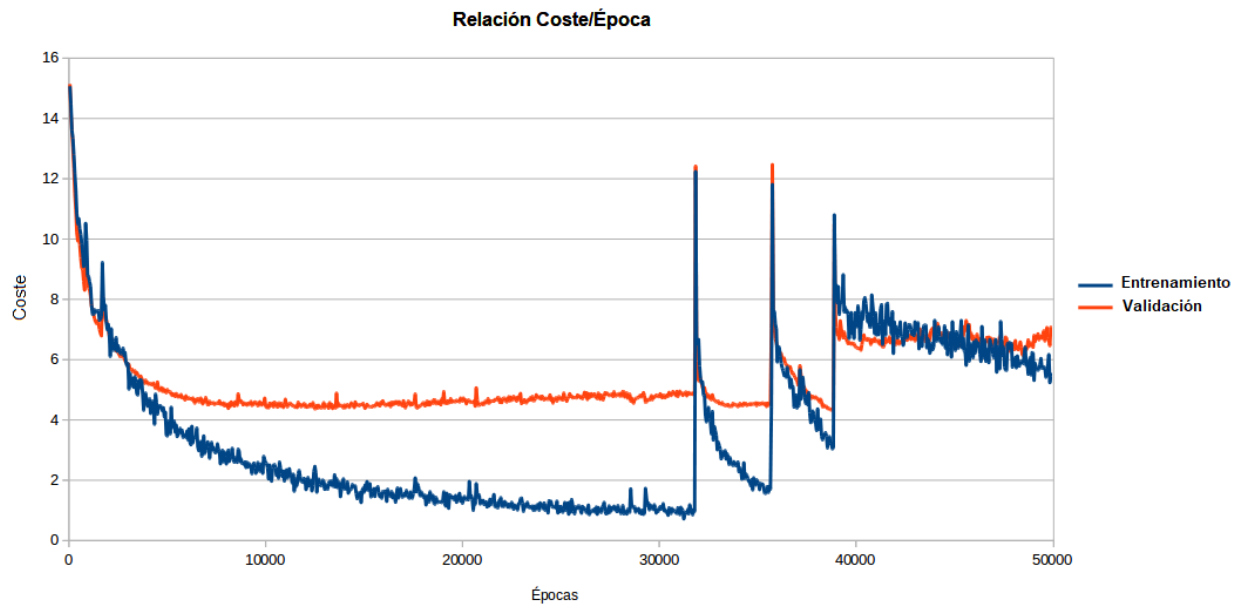


Figura 5-4. Resultados Coste para el test IAM.

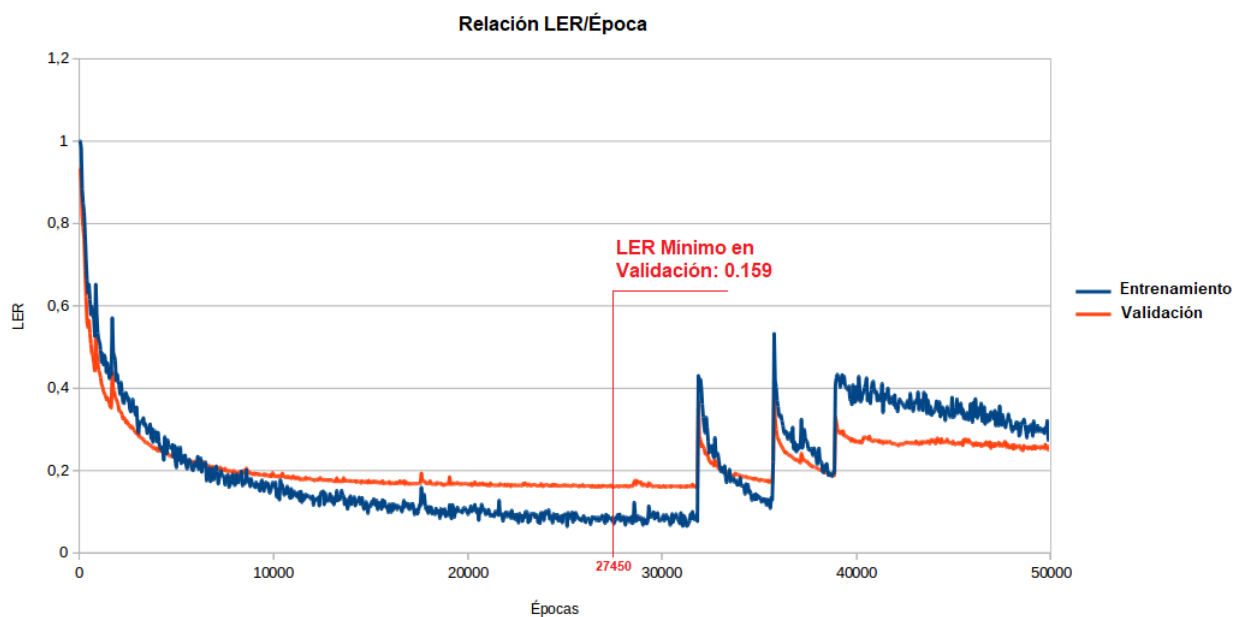


Figura 5-5. Resultados LER para test IAM.

Lo primero que podemos apreciar en estas imágenes es cómo se estabilizan los errores a partir de las 10.000 o 12.000 épocas, donde el sistema se sigue entrenando y mejorando sus resultados sobre el conjunto de entrenamiento, pero apenas se aprecian cambios en los resultados de las validaciones (es decir, se está produciendo un efecto de sobreentrenamiento o sobreajuste).

Otro aspecto a destacar es lo ocurrido entre la época 30.000 y la 40.000, donde aparecen unos picos que son fáciles de explicar, pero que pueden impactar. Durante el entrenamiento se trata de encontrar un mínimo para una determinada función objetivo, pero una vez encontrado un mínimo local no podemos asegurar que sea global. Es por ello que llegados a un punto donde se considera que existe un mínimo local, se realiza un “salto” en el espacio de búsqueda y se vuelve a buscar un nuevo mínimo local, aunque en este caso no ha dado mejores resultados.

En la época 27.450 se logra el LER mínimo de todo el proceso, guardando los parámetros en este punto para realizar el test sobre el conjunto propuesto.

Tras el entrenamiento se realiza el test sobre el conjunto propuesto, obteniendo el valor medio del LER y el Coste, además de generar un fichero CSV que contiene las transcripciones reales y las logradas por el sistema. Los resultados quedan expuestos en la siguiente tabla:

Tabla 5–2. Resultados del Test IAM.

Conjunto	LER	CTC-Loss
Entrenamiento	0.087	1.149
Validación	0.159	4.807
Test	0.175	3.977

En esta prueba se han obtenido mejores resultados que en la validación cruzada debido a que se realiza una única evaluación con una distribución bien establecida. Aun así, se debe aclarar que se han realizado varias pruebas, entrenando el modelo en busca de unos buenos resultados ya que existe cierta aleatoriedad debido a diferentes factores, como pueden ser el valor inicial de los parámetros o la extracción aleatoria de las muestras durante el entrenamiento.

6 CONCLUSIONES Y TRABAJOS FUTUROS

Una vez expuesto el procedimiento seguido para abordar el problema y obtenidos los resultados de las pruebas realizadas es hora de considerar las conclusiones obtenidas y proponer posibles trabajos futuros relacionados, que o bien buscan solventar las carencias y errores que a lo largo de nuestra aproximación hemos podido realizar, o bien proponen vías alternativas para abordar el mismo problema.

6.1 Conclusiones.

Para llegar a las primeras conclusiones, hay que recordar que en el presente proyecto no se ha pretendido construir un sistema que resuelva el problema con una eficiencia del 100%, sino mostrar de una manera clara pero precisa cómo se afrontan este tipo de problemas desde el Aprendizaje Automático y presentar un sistema implementado con Tensoflow, la librería de DL que mayor popularidad y porcentaje de uso tiene actualmente. Es por ello que se ha hecho un especial hincapié en el apartado teórico dejando en un segundo plano el apartado técnico y más relacionado con la implementación real del sistema, que no obstante queda incluido en los Anexos y alojado en un repositorio público.

Durante el diseño del sistema, se han implementado y probado distintos modelos, añadiéndole complejidad poco a poco. Se comenzó con un modelo muy simple para familiarizarnos con la capa CTC, dado que es determinante en la arquitectura, con una única capa Feedforward previa, dando unos resultados mediocres pero que se tomaron como punto de inicio. Dado que las ANNs permiten diseñar topologías libremente, se fueron añadiendo las distintas capas, dándoles sentido a su uso, hasta llegar a la arquitectura final que ha sido presentada.

Con esto quiero remarcar la versatilidad que ofrecen las ANNs, donde podemos comenzar con una arquitectura muy simple e ir añadiendo capas y estudiar su comportamiento. Existen multitud de topologías con funciones específicas y en la combinación de ellas surge la adaptabilidad del algoritmo.

Conforme se iban añadiendo las nuevas capas los resultados fueron mejorando, aunque no se ha hecho un estudio de los hiperparámetros que mejor se ajustaban a este problema, algo que sin duda hubiera mejorado el comportamiento de las arquitecturas, al igual que la inclusión de capas *Multi-dimensional Long Short-Term Memory* (MLSTM).

Las redes MLSTM son un tipo de *RNN Multi-Dimensional* (MDRNN) donde la idea básica es reemplazar la única conexión recurrente encontrada en la RNN estándar por tantas conexiones recurrentes como dimensiones en los datos.

Una RNN clásica opera de forma iterativa sobre una secuencia de elementos que guardan una relación a lo largo de una única dimensión, generalmente el tiempo. Las MDRNN generalizan este planteamiento para tensores de orden N , entradas multidimensionales, donde la entrada se recorre con una dirección establecida operando sobre los elementos de la entrada y las salidas para elementos pasados.

Trabajando en serie con capas CNN, las MLSTM han demostrado mejorar el rendimiento para problemas de Visión Artificial [16, 17] ya que pueden extraer información de la relación entre elementos distantes en el tensor de entrada, a diferencia de las CNN que están limitadas por las dimensiones de los filtros. De hecho, dichas capas han estado planteadas y presentes en el proyecto durante gran parte del tiempo, pero las implementaciones disponibles para TensorFlow no han llegado a convencerme. No responden mal, pero limitan el sistema haciéndolo trabajar siempre con el mismo número de muestras, e introducen un gasto computacional demasiado elevado para poder ser asumido en el proyecto.

Dejando a un lado el rediseño de la ANN con la introducción de nuevos tipos de capas, también sería interesante estudiar la eficiencia del sistema añadiendo varias nuevas ARBs en serie.

Otro punto a destacar han sido las medidas de error usadas y los resultados obtenidos que, desde un punto de vista práctico, no son muy intuitivos. Por un lado, tenemos el resultado obtenido por la función de verosimilitud, función que opera con logaritmos y puede tomar cualquier valor no negativo. Y, por otro lado, tenemos el LER que, aunque es algo más intuitiva, depende de la longitud de la palabra objetivo y no hace distinción entre sustitución, inserción o eliminación de caracteres, ocultando información en la medida.

También es relevante la importancia del dataset en los resultados del proyecto. Las imágenes de la *IAM Handwriting Database* son claras y limpias, perfectas para investigación y pruebas de concepto, pero pienso que no sería el dataset ideal para un sistema en producción, puesto que, por lo general, los archivos manuscritos no se encuentran en tan buenas condiciones y tendríamos un sistema demasiado adaptado a imágenes limpias, un sistema sobreentrenado. Algunas posibles soluciones podrían ser: (1) “ensuciar” los datos, añadiéndoles algún tipo de ruido a las imágenes, (2) añadir métodos que dificulten el sobreentrenamiento, parecidos a las capas Dropout, y (3) realizar un fuerte preprocesado sobre los nuevos manuscritos con el objetivo de hacerlos parecidos a los del dataset de entrenamiento.

Tampoco podemos pasar por alto las limitaciones que marca el idioma del dataset. Al utilizar este dataset estamos ajustando el sistema para transcribir palabras del inglés, dado que ha “aprendido” qué secuencia de letras es más probable en dicho idioma. Este comportamiento en el algoritmo es fruto de las capas CNN y RNN que establecen relaciones entre los elementos de la entrada según su ubicación y sus vecindades.

Habría que sopesar si es más productivo crear modelos por idiomas o crear un gran dataset internacional para entrenar una máquina general.

Otro aspecto relacionado con el idioma sería la inclusión de un corrector ortográfico. Tras implementar y evaluar el modelo, se realizaron pruebas añadiendo un corrector ortográfico a la salida del sistema con intención de mejorar el rendimiento sin modificarlo. Dado que no es objeto del proyecto no se ha incluido en la memoria, pero se consiguió una pequeña mejora en los resultados con un sistema entrenado previamente.

Respecto a las etiquetas, también podríamos comentar algunas mejoras. En las transcripciones obtenidas del sistema se manifiesta un problema propio de las capas CTC, cuando la palabra original tiene alguna letra repetida consecutivamente, como “hello”, “see” o “look”, el sistema las transcribe como una única letra. Estas palabras son muy comunes en el inglés y tienen mucha influencia en los resultados.

Si bien es cierto que la implementación de la capa CTC incluida en la librería TensorFlow tiene parámetros que anulan el descarte de etiquetas repetidas consecutivas, habría que realizar un estudio de la eficiencia, ya que se puede producir el efecto contrario, obteniendo transcripciones con etiquetas repetidas de más.

La segunda mejora que hace referencia a las etiquetas es la inclusión de la etiqueta “espacio” para adaptar el sistema a transcripciones de líneas completas. Para llevar a cabo esta mejora deberíamos trabajar con el dataset de líneas completas proporcionado por la misma *IAM Handwriting Database*, realizar el preprocesado oportuno e incluir una nueva etiqueta para el espacio entre palabras. En consecuencia, no parece un trabajo que requiera más que recursos temporales (y de computación). Sería interesante medir hasta qué punto el trabajar con líneas completas podría mejorar o no la eficiencia del sistema obtenido.

A lo largo de la memoria no se ha hecho grandes referencias a la implementación del sistema puesto que, bajo mi punto de vista, la potencia y el interés de estos sistemas está en los fundamentos matemáticos que los sostienen. Sin embargo, cabe mencionar las facilidades que ofrecen Python3 como lenguaje de programación, con multitud de librerías para casi todo, y TensorFlow como librería para el diseño de ANNs, con entrenamiento usando procesamiento en GPU y una amplia documentación. Sin duda, sin estas herramientas y el equipamiento ofrecido por el Departamento de Teoría de la Señal y Comunicaciones no hubiera sido posible construir el modelo ni llevar a cabo las pruebas, dejando el proyecto como un trabajo teórico.

Como se puede observar este tipo de proyectos no tienen una aplicación práctica directa, es más bien una prueba de concepto algo elaborada, que puede suponer un punto de partida para construir modelos robustos de DL que sean capaces de resolver este tipo de problemas.

Y dejando a un lado los cambios y mejoras posibles para el sistema que concierne al proyecto, también se podrían plantear este tipo de modelos para otros problemas como el reconocimiento de autoría, época, símbolos o señales.

Incluso para trabajar en problemas con patrones en video, como es el de reconocimiento de gestos o de objetos y señales en movimiento. Estos mismos algoritmos son, con más o menos esfuerzo, adaptables para procesar videos tomándolos como tensores de orden 4.

Podríamos pensar que un video no es más que un conjunto de imágenes concatenadas en una dimensión extra y, con este planteamiento, construir un sistema usando capas CNN con convoluciones en 3 dimensiones, donde los filtros serían tensores de orden 4, que extraerían información temporal. Además, podríamos añadir capas

MDLSTM que recorran la entrada en altura, anchura y tiempo, extrayendo información de todas las dimensiones.

Otro uso de las capas RNN y CTC podría ser la detección de secuencias de eventos. Aunque en este proyecto las RNNs se han usado para extraer información de una imagen, su uso más frecuente es frente a problemas dinámicos y junto a las capas CTC es posible construir un sistema que interprete un gran conjunto de sucesos consecutivos como una secuencia de eventos o alarmas.

Por otro lado, y desde un punto de vista más personal, este proyecto me ha obligado a abandonar el círculo de confort que nos ofrece el grado y al que nos vamos acomodando a lo largo de la ejecución de los estudios, ya que no está en absoluto relacionado con el mundo de las Telecomunicaciones y la Telemática, dándome la oportunidad de conocer un mundo nuevo, que me apasiona y del que me quedan muchos años de aprendizaje.

6.2 Trabajo Futuro.

A continuación, se exponen propuestas de mejora y desarrollo, algunas se han quedado en el tintero tras algún intento (abandonado por limitaciones técnicas o temporales) y otras solo han rondado en mi cabeza y no han llegado a proyectarse en un intento real, pero creo que pueden ser de valor para una aproximación a mayor profundidad de este problema.

Algunos de los muchos puntos a mejorar son:

- Inclusión de etiquetas para letras duplicadas: En el planteamiento e implementación del sistema nos hemos ceñido a etiquetas para letras aisladas, a propósito y por simplicidad, pero podría ser interesante comprobar la eficiencia de esta posible mejora.
- Mantener las etiquetas duplicadas consecutivas: Sería una posible solución al problema descrito en el punto anterior. Se trata de una mejora de fácil implementación, activar dicha opción a través de un parámetro en la función CTC, y puede significar una mejora considerable en los resultados.
- Corrector Ortográfico: Añadir un corrector como última capa y estudiar su comportamiento.
- Adaptar el sistema a líneas completas: Esta mejora es relativamente sencilla, pero aumenta el coste computacional del sistema dado que las imágenes son de mayor dimensión.
- Añadir nuevas ARBs: En el diseño original del sistema no se ha planteado incluir varias Arquitecturas Recurrentes Bidireccionales y podría ser interesante su estudio. Compañeros de la Escuela han realizado pruebas con sistemas que cuentan con varias capas de RNNs trabajando en serie y se han obtenido mejoras en los resultados.
- Añadir capas de Multi-dimensional Long Short-Term Memory: Distintas publicaciones avalan su buen rendimiento y la mejora en los resultados sería apreciable.
- Añadir Ruido: El sistema presentado ha sido entrenado con un dataset demasiado limpio y claro. Para un sistema en producción sería necesario añadir ruido al algoritmo o al dataset de entrenamiento.
- Exploración de Hiperparámetros: Sin duda, una exploración adecuada en este espacio incrementaría la precisión del sistema obtenido, a cambio de un consumo de recursos computacionales que puede tener sentido cuando se intente extraer un producto final con aplicaciones concretas.
- Distintos Idiomas: Estudiar cómo afrontar este mismo problema para distintas lenguas.
- Visión Artificial con ANNs: Como se ha comentado en el apartado anterior, estos mismos modelos son adaptables a distintos tipos de problemas relacionados con este ámbito de la Visión Artificial.
- Sistemas dinámicos con ANNs: En el proyecto presentado se exponen algoritmos que generalmente trabajan con funciones que evolucionan en el tiempo. Sería enriquecedor evaluar el rendimiento frente a distintos tipos de problemas dinámicos.

REFERENCIAS











- [1] K. P. Murphy, *Machine Learning: A Probabilistic Perspective*, Cambridge, Massachusetts y London, England: The MIT Press, 2012.
- [2] R. y. S. N. S. Plamondon, «Online and offline handwriting recognition: a comprehensive survey.», *IEEE Transactions*, vol. 22, nº 1, 2000.
- [3] U. Marti y H. Bunke, *Using a statistical language model to improve the performance of an HMM-based cursive handwriting recognition systems*, Bern, Switzerland: Institute of Computer Science of the University of Bern, 2000.
- [4] Yann LeCun, Yoshua Bengio y Geoffrey Hinton, *Deep learning*, vol. 521, Nature Publishing Group, 2015.
- [5] Simon Haykin, *Neural Networks and Learning Machines*, McMaster University, Hamilton, Ontario, Canada: Pearson Prentice Hall, 2009.
- [6] Douglas Reynolds, *Gaussian Mixture Models*, MIT Lincoln Laboratory, Lexington, USA, 2000.
- [7] Zoubin Ghahramani y Michael I. Jordan, *Factorial Hidden Markov Models*, Boston: Kluwer Academic Publishers, 1997.
- [8] Danilo P. Mandic, Jonathon A. Chambers, *Recurrent Neural Networks for Prediction: Learning Algorithms, Architectures and Stability*, Wiley, 2001.
- [9] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho y Yoshua Bengio, *Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling*, Montreal, Canada: Université de Montréal, 2014.
- [10] Yoshua Bengio, Patrice Simard y Paolo Frasconi, *Learning Long-Term Dependencies with Gradient Descent is Difficult*, *IEEE Transactions on Neural Networks*, 1994.
- [11] Sepp Hochreiter y Jürgen Schmidhuber, *Long Short-Term Memory*, München, Germany & Lugano, Switzerland, 1997.
- [12] David Stutz, *Understanding Convolutional Neural Networks*, Aachen, Alemania: Fakultät für Mathematik, Informatik und Naturwissenschaften, 2014.
- [13] Samer Hijazi, Rishi Kumar y Chris Rowen, *Using Convolutional Neural Networks for Image Recognition*, Cadence Design Systems, 2015.
- [14] Alex Krizhevsky, Ilya Sutskever y Geoffrey E. Hinton, *ImageNet Classification with Deep Convolutional Neural Networks*, Toronto, Canadá: Universidad de Toronto, 2012.
- [15] Max Jaderberg, Karen Simonyan, Andrea Vedaldi y Andrew Zisserman, *Reading Text in the Wild with Convolutional Neural Networks*, Universidad de Oxford, Reino Unido: Springer, 2015.

- [16] Alex Graves, Santiago Fernandez y Jürgen Schmidhuber, *Multi-Dimensional Recurrent Neural Networks*, Manno, Switzerland: IDSIA, 2007.
- [17] Alex Graves y Jürgen Schmidhuber, *Offline Handwriting Recognition with Multidimensional Recurrent Neural Networks*, Vancouver, Canada: NIPS 2008, 2008.
- [18] Alex Graves, Santiago Fernández, Faustino Gomez y Jürgen Schmidhuber, *Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks*, Manno-Lugano (Switzerland), Garching & Munich (Germany): Istituto Dalle Molle di Studi sull'Intelligenza Artificiale, Technische Universität München, 2006.
- [19] Jan Snyma, *Practical Mathematical Optimization: An Introduction to Basic Optimization Theory and Classical and New Gradient-Based Algorithms*, Universidad de Pretoria, Sur Africa: Springer, 2005.
- [20] Vu Pham, Theodore Bluche, Christopher Kermorvant y Jérôme Louradour, *Dropout improves Recurrent Neural Networks for Handwriting Recognition*, Orsay, Paris, Francia y Drive, Singapore, 2014.
- [21] *Offline Arabic Handwriting Recognition with Multidimensional Recurrent Neural Networks*, Springer, 2012.
- [22] *IAM Hand Writing Database*, <http://www.fki.inf.unibe.ch/databases/iam-handwriting-database>.
- [23] K. Hornik, *Multilayer Feedforward Networks are Universal Approximators*, Pergamon Press plc, 1989.
- [24] R. Rojas, *Neural Networks*, Berlin, Germany: Springer-Verlag, 1996.
- [25] Paul J. Werbos, *Backpropagation Through Time: What It Does and How to Do It*, Proceeding of the IEE, 1990.
- [26] *Offline Handwriting Recognition with Deep Learning implemented in TensorFlow.*, <https://github.com/hugrubsan/Offline-Handwriting-Recognition-with-TensorFlow>.
- [27] Dominik Scherer, Andreas Müller y Sven Behnke, *Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition*, Thessaloniki, Greece: 20th International Conference on Artificial Neural Networks (ICANN), 2010.
- [28] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever y Ruslan Salakhutdinov, *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*, Journal of Machine Learning Research 15, 2014.

ANEXOS

Anexo A. Código de la Implementación.

Directorio del Proyecto.

 CSV	CSV: Directorio que contiene las transcripciones de las imágenes almacenadas como ficheros CSV distribuidos como datasets.
 Data	Data: Directorio para almacenar las imágenes.
 Results	Results: Directorio donde se almacenan los resultados de las pruebas.
 ANN_model.py	Ficheros *.py: Scripts escritos en Python3 para las distintas partes del proyecto.
 README.md	
 clean_IAM.py	
 config.json	
 cross-validation.py	
 hw_utils.py	Config.json: Fichero de configuración
 test.py	
 train.py	

hw_utils.py.

Fichero con funciones para escalar e invertir las imágenes, extraer los distintos lotes de muestras y realizar las validaciones del modelo. Las librerías de Pandas y PIL para Python facilitan el trabajo para tratar con estos tipos de datos.

```
import os
import sys
from PIL import Image
import pandas as pd
import numpy as np

def scale_invert(raw_path, proc_path,height,width):

    """
    Función que escala e invierte cada imagen para almacenarlas en un directorio común.
    Se conservan las proporciones de la imagen original y se añade un relleno hasta alcanzar
    el ancho objetivo.

    Argumentos:

    - raw_path: Ruta de la imagen original. (String)
    - proc_path: Ruta donde almacenar la imagen procesada. (String)
    - height: Altura de las imágenes. (Int)
    - width: Anchura de la imágenes. (Int)

    """
    # Cargamos la imagen
    im = Image.open(raw_path)

    # Reescalamos
    raw_width, raw_height = im.size
    new_width = int(round(raw_width * (height / raw_height)))
    im = im.resize((new_width, height), Image.NEAREST)
    im_map = list(im.getdata())
    im_map = np.array(im_map)
    im_map = im_map.reshape(height, new_width).astype(np.uint8)

    # Rellenamos e invertimos los valores.
    data = np.full((height, width - new_width + 1), 255)
    im_map = np.concatenate((im_map, data), axis=1)
    im_map = im_map[:, 0:width]
    im_map = (255 - im_map)
    im_map = im_map.astype(np.uint8)
    im = Image.fromarray(im_map)

    # Almacenamos todas las imágenes en directorio común
    im.save(str(proc_path), "png")
    print("Processed image saved: " + str(proc_path))
```

```
def extract_training_batch(ctc_input_len, batch_size, im_path, csv_path):

    """
    Función que extrae un lote de imágenes y sus transcripciones de manera aleatoria para entrenar la ANN.

    Argumentos:

    - ctc_input_len: Longitud de la secuencia de entrada a la capa CTC. (Int)
    - batch_size: Tamaño del lote. (Int)
    - im_path: Ruta al directorio donde se almacenan las imágenes. (String)
    - csv_path: Ruta al dataset de entrenamiento. (Int)

    Salida:

    - batchx: Tensor que contiene las imágenes como matrices de entrada a la ANN.
      (Array de Floats: [batch_size, height, width, 1])
    - sparse: SparseTensor que contiene las etiquetas como valores enteros positivos. (SparseTensor: indice, values, shape)
    - transcriptions: Array con las transcripciones correspondientes a las imágenes de "batchx". (Array de Strings: [batch_size])
    - seq_len: Array con la longitud de la secuencia de entrada a la capa CTC, "ctc_input_len". (Array de Ints: [batch_size])

    """

    # Extraemos aleatoriamente un DataFrame de tamaño "batch_size" del Dataset de entrenamiento.
    df = pd.read_csv(csv_path, sep=";", index_col="index")
    df_sample = df.sample(batch_size).reset_index()

    # Declaramos las variables para la salida.
    batchx = []
    transcriptions = []
    index = []
    values = []
    seq_len = []

    # Creamos el lote a partir del DataFrame de muestras aleatorias.
    for i in range(batch_size):
        im Apt = df_sample.loc[i, ['image']].as_matrix()
        df_y = df_sample.loc[i, ['transcription']].as_matrix()
        for fich in im Apt:

            # Extraemos la imagen y la mapeamos en una matriz normalizada.
            fich = str(fich)
            fich = fich.replace("'", "").replace("'", "")
            im = Image.open(im_path + fich + ".png")
            width, height = im.size
            im_map = list(im.getdata())
            im_map = np.array(im_map)
            im_map = im_map / 255
            result = im_map.reshape(height, width, 1)
            batchx.append(result)
```

```

# Extraemos las etiquetas parseando la transcripción.
original=""
for n in list(str(df_y)):
    if n == n.lower() and n != n.upper():
        if n in "0123456789":
            values.append(int(n))
            original = original + n
        elif n == n.lower():
            values.append(int(ord(n) - 61))
            original = original + n
        elif n == n.upper():
            values.append(int((ord(n) - 55)))
            original = original + n

# Añadimos el índice del SparseTensor.
for j in range(len(str(df_y))-4):
    index.append([i,j])

# Añadimos las transcripciones y la longitud de la secuencia de entrada a la CTC.
transcriptions.append(original)
seq_len.append(ctc_input_len)

# Creamos el Array que contiene todas las imágenes normalizadas el lote, entrada de la ANN.
batchx = np.stack(batchx, axis=0)

# Creamos el SparseTensor con el índice, las etiquetas que representan cada caracter y la longitud máxima de palabra
shape=[batch_size,18]
sparse=index,values,shape

return batchx, sparse, transcriptions, seq_len

def extract_ordered_batch(ctc_input_len,batch_size,im_path, csv_path,cont):
    """
    Función que extrae un lote de imágenes y sus transcripciones de manera ordenada para validar o testear la ANN.

    Argumentos:

    - ctc_input_len: Longitud de la secuencia de entrada a la capa CTC. (Int)
    - batch_size: Tamaño del lote. (Int)
    - im_path: Ruta al directorio donde se almacenan las imágenes. (String)
    - csv_path: Ruta al dataset de validación. (Int)
    - cont: Índice auxiliar que permite la extracción de lotes de manera ordenada. (Int)

    Salida:

    - batchx: Tensor que contiene las imágenes como matrices de entrada a la ANN.
      (Array de Floats: [batch_size, height, width, 1])
    - sparse: SparseTensor que contiene las etiquetas como valores enteros positivos. (SparseTensor: índice,values,shape)
    - transcriptions: Array con las transcripciones correspondientes a las imágenes de "batchx". (Array de Strings: [batch_size])
    - seq_len: Array con la longitud de la secuencia de entrada a la capa CTC, "ctc_input_len". (Array de Ints: [batch_size])
    - num_samples: Número de muestras extraídas. (Int)
    """

    # Extraemos secuencialmente un DataFrame de tamaño "batch_size" del Dataset.
    df = pd.read_csv(csv_path, sep=",", index_col="index")
    df_sample=df.loc[int(cont*batch_size):int((cont+1)*batch_size)-1,:].reset_index()
    num_samples=int(len(df_sample.axes[0]))

```

```

# Declaramos las variables para la salida.
batchx = []
transcriptions = []
index = []
values=[]
seq_len=[]

# Creamos el lote a partir del Dataframe de muestras aleatorias.
if len(df_sample.axes[0]) is not 0:
    for i in range(len(df_sample.axes[0])):
        im Apt = df_sample.loc[i, ['image']].as_matrix()
        df_y =df_sample.loc[i, ['transcription']].as_matrix()
        for fich in im Apt:

            # Extraemos la imagen y la mapeamos en una matriz normalizada.
            fich = str(fich)
            fich = fich.replace("'", "").replace("'", "")
            im = Image.open(im_path + fich + ".png")
            width, height = im.size
            im_map = list(im.getdata())
            im_map = np.array(im_map)
            im_map = im_map / 255
            result=im_map.reshape(height, width,1)
            batchx.append(result)

            # Extraemos las etiquetas parseando la transcripción.
            original=""
            for n in list(str(df_y)):
                if n == n.lower() and n == n.upper():
                    if n in "0123456789":
                        values.append(int(n))
                        original=original+n

                    elif n==n.lower():
                        values.append(int(ord(n)-61))
                        original = original + n
                    elif n==n.upper():
                        values.append(int((ord(n)-55)))
                        original = original + n

            # Añadimos el índice del SparseTensor.
            for j in range(len(str(df_y))-4):
                index.append([i,j])

            # Añadimos las transcripciones y la longitud de la secuencia de entrada a la CTC.
            transcriptions.append(original)
            seq_len.append(ctc_input_len)

# Creamos el Array que contiene todas las imagenes normalizadas el lote, entrada de la ANN.
batchx=np.stack(batchx, axis=0)

# Creamos el SparseTensor con el índice, las etiquetas que representan cada caracter y la longitud máxima de palabra
shape=[batch_size,18]
sparse=index,values,shape

return batchx, sparse, transcriptions, seq_len, num_samples

```

```

def validation(curr_epoch,ctc_input_len, batch_size, im_path, csv_path, inputs, targets, keep_prob, seq_len, session, cost, ler):

    """
    Función que realiza la validación de la ANN sobre un dataset concreto.

    Argumentos:

        - curr_epoch: Época actual. (Int)
        - ctc_input_len: Longitud de la secuencia de entrada a la capa CTC. (Int)
        - batch_size: Tamaño del lote. (Int)
        - im_path: Ruta al directorio donde se almacenan las imágenes. (String)
        - csv_path: Ruta al dataset de validación. (Int)
        - inputs: Placeholder de la entrada del model. (placeholder)
        - targets: Placeholder de las salidas objetivo. (placeholder)
        - keep_prob: Placeholder para la probabilidad de dropout. (placeholder)
        - seq_len: Placeholder para la longitud de la secuencia de entrada a la capa CTC. (placeholder)
        - session: Sesión actual de TensorFlow. (Session)
        - cost: Tensor para la salida del error de la CTC. (Tensor: [1])
        - ler: Tensor para la salida del LER. (Tensor:[1])

    Salida:

        - val_tuple: Resultado de la validación del modelo en una época completa. (Tuple: {'epoch','cost','LER'})
    """

    # Variables auxiliares.
    cont = 0
    total_val_cost = 0
    total_val_ler = 0

    # Bucle para realizar la validación sobre el Dataset completo
    while cont >= 0:
        # Extraemos los lotes de forma secuencial mediante "cont"
        val_inputs, val_targets, val_original, val_seq_len, num_samples = extract_ordered_batch(
            ctc_input_len, batch_size, im_path, csv_path, cont)

        # Si el número de muestras extraídas es igual a "batch_size" se ha extraído un lote completo.
        if num_samples == batch_size:
            val_feed = {inputs: val_inputs,
                        targets: val_targets,
                        keep_prob: 1,
                        seq_len: val_seq_len}
            val_cost, val_ler = session.run([cost, ler], val_feed)
            total_val_cost += val_cost
            total_val_ler += val_ler
            cont += 1

        # No se ha podido extraer ningún lote más y por lo tanto, se calcula la media de "cost" y "ler"
        elif num_samples == 0:
            val_tuple = {'epoch': [curr_epoch], 'val_cost': [total_val_cost / (cont + 1)],
                        'val_ler': [total_val_ler / (cont + 1)]}
            cont = -1

        # No se ha podido extraer el lote completo, no quedan suficientes muestras en el Dataset y por lo tanto,
        # se calcula la media de "cost" y "ler".
        else:
            val_feed = {inputs: val_inputs,
                        targets: val_targets,
                        keep_prob: 1,
                        seq_len: val_seq_len}
            val_cost, val_ler = session.run([cost, ler], val_feed)
            total_val_cost += val_cost
            total_val_ler += val_ler
            val_tuple = {'epoch': [curr_epoch], 'val_cost': [total_val_cost / (cont + 1)],
                        'val_ler': [total_val_ler / (cont + 1)]}
            cont = -1

    return val_tuple

```

clean_IAM.py.

Script para limpieza y el preprocesamiento de las imágenes, dejándolas aptas para el entrenamiento y las evaluaciones.

```
#!/usr/bin/env python
# -*- coding: utf-8

import os
import sys
from PIL import Image
import pandas as pd
import numpy as np
import json
import hw_utils

def main():

    # Ruta del archivo de configuración, pasada por argumentos o por defecto "./config.json".
    if len(sys.argv) == 1:
        print("Execution without arguments, config file by default: ./config.json")
        config_file=str('./config.json')
    elif len(sys.argv) == 2:
        print("Execution with arguments, config file:" +str(sys.argv[1]))
        config_file = str(sys.argv[1])
    else:
        print()
        print("ERROR")
        print("Wrong number of arguments. Execute:")
        print(">> python3 clean_IAM.py [path_config_file]")
        exit(1)

    # Cargamos el archivo de configuración
    try:
        data = json.load(open(config_file))
    except FileNotFoundError:
        print()
        print("ERROR")
        print("No such config file : " + config_file)
        exit(1)

    # Si el directorio destino no existe, se crea.
    if not os.path.exists(str(data["general"]["processed_data_path"])):
        os.mkdir(str(data["general"]["processed_data_path"]))

    # Lista con todos los ficheros del directorio.
    lstDir = os.walk(str(data["general"]["raw_data_path"]))

    # Se leen del csv los nombres de las imagenes aptas.
    df = pd.read_csv(str(data["general"]["csv_path"]), sep=",", index_col="index")
    df = df.loc[:, ['image']]
    lstIm = df.as_matrix()

    # Recorremos el directorio donde estan almacenadas las imagenes de la IAM.
    for root, dirs, files in lstDir:
        for file in files:
            (name, ext) = os.path.splitext(file)
            # Comprobamos si el fichero está en el array de imagenes aptas.
            if name in lstIm:
                # Se ejecuta la función "scale_invert"
                hw_utils.scale_invert(str(root)+str("/") +str(name+ext),
                                      str(data["general"]["processed_data_path"])+str(name+ext),int(data["general"]["height"]),int(data["general"]["width"]))

if __name__ == "__main__":
    main()
```


ANN_model.py.

Script para crear el modelo de ANN.

En este fichero se aloja la estructura del modelo y permite crear distintas versiones del sistema ajustando sus hiperparámetros. Para implementar el modelo se ha hecho uso de la librería de TensorFlow, que agiliza mucho el procedimiento una vez comprendido su funcionamiento, basado en grafos y tensores.

Además hay que añadir que permite el procesamiento por GPU para alcanzar un mayor rendimiento y acortar los tiempos de entrenamiento y evaluación.

```
import tensorflow as tf

def CNN_RNN CTC(kernel_size, num_conv1, num_conv2, num_conv3, num_conv4, num_conv5, num_rnn, num_fc, HEIGHT, WIDTH, num_classes):
    """
    Función que construye el modelo ANNs basado en capas convolucionales, capas recurrentes y una última capa CTC.
    Arquitectura compuesta por 5 Capas CNN-MAXPOOL-DROP, dos RNNs que trabajan en paralelo
    recorriendo una misma dimensión pero en sentidos opuestos, una Fullconnect-CTC que nos devuelve el error cometido.

    Argumentos:

    - kernel_size: Tamaño del kernel para las CNN. (Int)
    - num_conv1: Número de neuronas de la 1ª CNN. (Int)
    - num_conv2: Número de neuronas de la 2ª CNN. (Int)
    - num_conv3: Número de neuronas de la 3ª CNN. (Int)
    - num_conv4: Número de neuronas de la 4ª CNN. (Int)
    - num_conv5: Número de neuronas de la 5ª CNN. (Int)
    - num_rnn: Número de neuronas de la RNN. (Int)
    - num_fc: Número de neuronas de la 1ª Fullconnect. (Int)
    - HEIGHT: Altura de las imágenes. (Int)
    - WIDTH: Anchura de las imágenes. (Int)
    - num_classes: Número de etiquetas, reales + "blanco". (Int)

    Salida:

    - graph: Grafo que contiene la arquitectura del modelo. (Graph)
    - inputs: Placeholder para la entrada. (Placeholder)
    - targets: Placeholder para las salidas objetivo. (Placeholder)
    - keep_prob: Placeholder para la probabilidad de dropout. (Placeholder)
    - seq_len: Placeholder para longitud de la secuencia de entrada a la CTC. (Placeholder)
    - optimizer: Operador para minimizar el error del modelo. (Operation)
    - cost: Operador para obtener el error del modelo. (Operation)
    - ler: Operador para obtener el LER del modelo. (Operation)
    - decoded: Operador para obtener la decodificación de la salida del modelo. (Operation)
```

```

"""

graph = tf.Graph()
with graph.as_default():

    # PLACEHOLDERS
    # Entrada: Tensor [tamaño de batch, altura imagen, anchura imagen, canal]
    inputs = tf.placeholder(tf.float32, [None, HEIGHT, WIDTH, 1])

    # Longitud de la secuencia de entrada a la CTC: Tensor [tamaño de batch]
    seq_len = tf.placeholder(tf.int32, [None])

    # Probabilidad de Dropout: float32
    keep_prob = tf.placeholder(tf.float32)

    # Etiquetas objetivo: SparseTensor (índice, etiquetas, [tamaño de batch, longitud máxima de palabra])
    targets = tf.sparse_placeholder(tf.int32)

    # VARIABLES
    # Filtros y bias de CNNs
    w_conv1 = tf.Variable(tf.random_normal([kernel_size, kernel_size, 1, num_conv1], stddev=0.01))
    b_conv1 = tf.Variable(tf.random_normal([num_conv1], stddev=0.01))
    w_conv2 = tf.Variable(tf.random_normal([kernel_size, kernel_size, num_conv1, num_conv2], stddev=0.01))
    b_conv2 = tf.Variable(tf.random_normal([num_conv2], stddev=0.01))
    w_conv3 = tf.Variable(tf.random_normal([kernel_size, kernel_size, num_conv2, num_conv3], stddev=0.01))
    b_conv3 = tf.Variable(tf.random_normal([num_conv3], stddev=0.01))
    w_conv4 = tf.Variable(tf.random_normal([kernel_size, kernel_size, num_conv3, num_conv4], stddev=0.01))
    b_conv4 = tf.Variable(tf.random_normal([num_conv4], stddev=0.01))
    w_conv5 = tf.Variable(tf.random_normal([kernel_size, kernel_size, num_conv4, num_conv5], stddev=0.01))
    b_conv5 = tf.Variable(tf.random_normal([num_conv5], stddev=0.01))

    # Pesos de RNN
    lstm_fw_cell = tf.contrib.rnn.BasicLSTMCell(num_rnn, forget_bias=1.0)
    lstm_bw_cell = tf.contrib.rnn.BasicLSTMCell(num_rnn, forget_bias=1.0)

    # Pesos de FULLCONNECT
    w1 = tf.Variable(tf.random_normal([2*num_rnn, num_fc], stddev=0.01))
    b1 = tf.Variable(tf.random_normal([num_fc], stddev=0.01))
    w2 = tf.Variable(tf.random_normal([num_fc, num_classes], stddev=0.01))
    b2 = tf.Variable(tf.random_normal([num_classes], stddev=0.01))

    # ARQUITECTURA
    # Normalización de la entrada
    inputs = tf.nn.l2_normalize(inputs, [1, 2])

    #CAPA 1 CNN-MAXPOOL-DROP
    h_conv1 = tf.nn.relu(tf.nn.conv2d(inputs, w_conv1, strides=[1,1,1,1], padding='SAME') + b_conv1)
    h_pool1 = tf.nn.max_pool(h_conv1, ksize=[1,2,2,1], strides=[1,2,2,1], padding='SAME')
    h_pool1=tf.nn.l2_normalize(h_pool1,[1,2])
    h_pool1=tf.nn.dropout(h_pool1,keep_prob=keep_prob)

    # CAPA 2 CNN-MAXPOOL-DROP
    h_conv2 = tf.nn.relu(tf.nn.conv2d(h_pool1, w_conv2, strides=[1,1,1,1], padding='SAME') + b_conv2)
    h_pool2 = tf.nn.max_pool(h_conv2, ksize=[1,2,2,1], strides=[1,2,2,1], padding='SAME')
    h_pool2=tf.nn.l2_normalize(h_pool2,[1,2])
    h_pool2=tf.nn.dropout(h_pool2,keep_prob=keep_prob)

    # CAPA 3 CNN-MAXPOOL-DROP
    h_conv3 = tf.nn.relu(tf.nn.conv2d(h_pool2, w_conv3, strides=[1,1,1,1], padding='SAME') + b_conv3)
    h_pool3 = tf.nn.max_pool(h_conv3, ksize=[1,2,2,1], strides=[1,2,2,1], padding='SAME')
    h_pool3=tf.nn.l2_normalize(h_pool3,[1,2])

```

```

h_pool3=tf.nn.dropout(h_pool3,keep_prob=keep_prob)

# CAPA 4 CNN-MAXPOOL-DROP
h_conv4 = tf.nn.relu(tf.nn.conv2d(h_pool3, w_conv4, strides=[1,1,1,1], padding='SAME') + b_conv4)
h_pool4 = tf.nn.max_pool(h_conv4, ksize=[1,2,2,1], strides=[1,2,2,1], padding='SAME')
h_pool4=tf.nn.l2_normalize(h_pool4,[1,2])
h_pool4=tf.nn.dropout(h_pool4,keep_prob=keep_prob)

# CAPA 5 CNN-MAXPOOL-DROP
h_conv5 = tf.nn.relu(tf.nn.conv2d(h_pool4, w_conv5, strides=[1,1,1,1], padding='SAME') + b_conv5)
h_pool5 = tf.nn.max_pool(h_conv5, ksize=[1,2,2,1], strides=[1,2,2,1], padding='SAME')
h_pool5=tf.nn.l2_normalize(h_pool5,[1,2])
h_pool5=tf.nn.dropout(h_pool5,keep_prob=keep_prob)

# CAPA 6 RNNs
outputs=tf.transpose(h_pool5, (2,0,1,3))
outputs=tf.reshape(outputs, (int(WIDTH/(2*5)), -1, int(HEIGHT*num_conv5/(2*5))))
outputs=tf.transpose(outputs, (1,0,2))
outputs, _ = tf.nn.bidirectional_dynamic_rnn(lstm_fw_cell, lstm_bw_cell, outputs,dtype=tf.float32)
outputs=tf.concat(outputs,2)
outputs=tf.transpose(outputs, (1,0,2))
outputs=tf.reshape(outputs, (-1,2*num_rnn))

# CAPA 7 FULLCONNECT-CTC_Loss
logits = tf.matmul(outputs, w1) + b1
logits = tf.matmul(logits, w2) + b2
logits = tf.reshape(logits, (int(WIDTH/(2*5)), -1, num_classes))
loss = tf.nn.ctc_loss(targets, logits, seq_len, preprocess_collapse_repeated=True)
cost = tf.reduce_mean(loss)

# Optimización para minimizar el error
optimizer = tf.train.AdamOptimizer(learning_rate=0.01).minimize(cost)

# Decodificador para extraer la secuencia de caracteres
decoded, log_prob = tf.nn.ctc_greedy_decoder(logits, seq_len)

# Error: Label Error Rate
ler = tf.reduce_mean(tf.edit_distance(tf.cast(decoded[0], tf.int32), targets))

return graph, inputs, targets, keep_prob, seq_len, optimizer, cost, ler, decoded

```

cross-validation.py.

Script para relizar la validación cruzada con 10 evaluaciones.

Este proceso supone mucha carga computacional ya que se realizan 10 entrenamientos con sus respectivas validaciones sobre el dataset de validación al completo. Esto provoca que la prueba pueda durar varios días y se ha de tener en cuenta.

```
import os
import sys
from random import randint
import tensorflow as tf
import hw_utils
import pandas as pd
import ANN_model
import json
import ast

def run_ctc():

    # Ruta del archivo de configuración, pasada por argumentos o por defecto "./config.json".
    if len(sys.argv) == 1:
        print("Execution without arguments, config file by default: ./config.json")
        config_file = str('./config.json')

    elif len(sys.argv) == 2:
        print("Execution with arguments, config file:" + str(sys.argv[1]))
        config_file = str(sys.argv[1])

    else:
        print()
        print("ERROR")
        print("Wrong number of arguments. Execute:")
        print(">> python3 cross-validation.py [path_config_file]")
        exit(1)

    # Cargamos el archivo de configuración
    try:
        config = json.load(open(config_file))
    except FileNotFoundError:
        print()
        print("ERROR")
        print("No such config file : " + config_file)
        exit(1)

    # Si el directorio destino no existe, se crea.
    if not os.path.exists(str(config["cross-validation"]["results_path"])):
        os.mkdir(str(config["cross-validation"]["results_path"]))

    # Extraemos las variables generales para la cross-validation
    im_path=str(config["general"]["processed_data_path"])
    csv_path=str(config["cross-validation"]["csv_path"])
    results_path=str(config["cross-validation"]["results_path"])
    batch_size = int(config["cross-validation"]["batch_size"])
    num_epochs = int(config["cross-validation"]["num_epochs"])
    val_period = int(config["cross-validation"]["validation_period"])
    print_period = int(config["cross-validation"]["print_period"])
    height = int(config["general"]["height"])
    width = int(config["general"]["width"])
    dct=ast.literal_eval(str(config["general"]["dictionary"]))
```

```

# Extraemos los parametros del modelo a validar
kernel_size=int(config["cnn-rnn-ctc"]["kernel_size"])
num_conv1=int(config["cnn-rnn-ctc"]["num_conv1"])
num_conv2=int(config["cnn-rnn-ctc"]["num_conv2"])
num_conv3=int(config["cnn-rnn-ctc"]["num_conv3"])
num_conv4=int(config["cnn-rnn-ctc"]["num_conv4"])
num_conv5=int(config["cnn-rnn-ctc"]["num_conv5"])
num_rnn=int(config["cnn-rnn-ctc"]["num_rnn"])
num_fc=int(config["cnn-rnn-ctc"]["num_fc"])
num_classes=int(config["cnn-rnn-ctc"]["num_classes"])
ctc_input_len=int(config["cnn-rnn-ctc"]["ctc_input_len"])

# Creamos el modelo ANN
model = ANN_model.CNN_RNN_CTC(kernel_size, num_conv1, num_conv2, num_conv3, num_conv4,
                               num_conv5, num_rnn, num_fc, height, width, num_classes)

graph=model[0]
inputs=model[1]
targets=model[2]
keep_prob=model[3]
seq_len=model[4]
optimizer=model[5]
cost=model[6]
ler=model[7]
decoded=model[8]

# Bucle para realizar la validación sobre los 10 Datasets.
for i in range(10):

    # Declaramos los DataFrames para almacenar el resultado de cada validación
    train_result = pd.DataFrame()
    val_result = pd.DataFrame()

    # Creamos la sesión con el modelo previamente cargado e inicializamos la variables.
    with tf.Session(graph=graph) as session:
        tf.global_variables_initializer().run()

    # Bucle de épocas.
    for curr_epoch in range(num_epochs):

        # Extraemos un lote aleatorio del Dataset de entrenamiento.
        train_inputs, train_targets, original, train_seq_len = hw_utils.extract_training_batch(ctc_input_len,batch_size,
                                                    im_path, csv_path + "train" + str(i + 1) + ".csv")

        feed = {inputs: train_inputs, targets: train_targets, keep_prob: 0.5, seq_len: train_seq_len}

        # Ejecutamos "optimizer", minimizando el error para el lote extraído.
        _ = session.run([optimizer], feed)

        # Comprobamos el periodo de validación para el modelo.
        if curr_epoch % val_period == 0:

            # Calculamos el error de la CTC y el LER para el dataset de entrenamiento y almacenamos los resultados.
            train_cost, train_ler = session.run([cost, ler], feed)
            train_tuple = {'epoch': [curr_epoch], 'train_cost': [train_cost], 'train_ler': [train_ler]}
            train_result = pd.concat([train_result, pd.DataFrame(train_tuple)])

            # Realizamos la validación del modelo y almacenamos los resultados.
            val_tuple=hw_utils.validation(curr_epoch,ctc_input_len, batch_size, im_path, csv_path + "validation" + str(i + 1) + ".csv",
                                         inputs, targets, keep_prob, seq_len, session, cost, ler)
            val_result = pd.concat([val_result, pd.DataFrame(val_tuple)])

```

```

# Comprobamos el periodo de impresión de ejemplos.
if curr_epoch % print_period == 0:

    # Imprimimos la salida del modelo para 10 ejemplos al azar del dataset de validación.
    print("Epoch: " + str(curr_epoch))
    print("Examples:")
    for j in range(10):

        # Extraemos una muestra.
        prob_inputs, prob_targets, prob_original, prob_seq_len, _ = hw_utils.extract_ordered_batch(ctc_input_len,1,
                                                                                               im_path, csv_path + "validation" + str(i + 1) + ".csv",
                                                                                               randint(0,8500))

        prob_feed = {inputs: prob_inputs,
                     targets: prob_targets,
                     keep_prob: 1,
                     seq_len: prob_seq_len}

        # Obtenemos la salida y la mapeamos como una palabra para imprimirla por pantalla.
        prob_d = session.run(decoded[0], feed_dict=prob_feed)
        output = str(list(map(dct.get, list(prob_d.values))))
        for ch in ["'", '"', ',', ' ']:
            output = output.replace(ch, "")
            prob_original=prob_original.replace(ch, "")
        print("Target: " + prob_original + "          Model Output: " + output)

# Cerramos para la siguiente validación.
session.close()

# Almacenamos los resultados
val_result.to_csv(results_path + "validation_result" + str(i + 1) + ".csv", index=False)
train_result.to_csv(results_path + "training_result" + str(i + 1) + ".csv", index=False)
print("Available results number " + str(i + 1) + " in " + results_path)

print("THE CROSS-VALIDATION IS OVER")

if __name__ == '__main__':
    run_ctc()

```

train.py.

Script para entrenar el sistema de cara a realizar el test IAM.

El entrenamiento se realiza mediante épocas y cada cierto número de épocas se valida el sistema. En estas validaciones se obtiene el valor medio del LER para todo el dataset de validación y si es menor al valor medio obtenido en épocas anteriores se guarda el estado del sistema.

Tras realizar todo el entrenamiento se obtienen varios ficheros CSV con las medidas de errores en cada época y el estado del sistema que mejor redimiento a mostrado.

```
import os
import sys
from random import randint
import tensorflow as tf
import hw_utils
import pandas as pd
import ANN_model
import json
import ast

def run_ctc():

    # Ruta del archivo de configuración, pasada por argumentos o por defecto "./config.json".
    if len(sys.argv) == 1:
        print("Execution without arguments, config file by default: ./config.json")
        config_file = str('./config.json')

    elif len(sys.argv) == 2:
        print("Execution with arguments, config file:" + str(sys.argv[1]))
        config_file = str(sys.argv[1])

    else:
        print()
        print("ERROR")
        print("Wrong number of arguments. Execute:")
        print(">> python3 train.py [path_config_file]")
        exit(1)

    # Cargamos del archivo de configuración
    try:
        config = json.load(open(config_file))
    except FileNotFoundError:
        print()
```

```

    print("ERROR")
    print("No such config file : " + config_file)
    exit(1)

# Si el directorio destino no existe, se crea.
if not os.path.exists(str(config["IAM-test"]["results_path"])):
    os.mkdir(str(config["IAM-test"]["results_path"]))
if not os.path.exists(str(config["IAM-test"]["checkpoints_path"])):
    os.mkdir(str(config["IAM-test"]["checkpoints_path"]))

# Extraemos las variables generales para el entrenamiento.
im_path=str(config["general"]["processed_data_path"])
csv_path=str(config["IAM-test"]["csv_path"])
results_path=str(config["IAM-test"]["results_path"])
checkpoints_path=str(config["IAM-test"]["checkpoints_path"])
batch_size = int(config["IAM-test"]["batch_size"])
num_epochs = int(config["IAM-test"]["num_epochs"])
val_period = int(config["IAM-test"]["validation_period"])
print_period = int(config["IAM-test"]["print_period"])
height = int(config["general"]["height"])
width = int(config["general"]["width"])
dct=ast.literal_eval(str(config["general"]["dictionary"]))

# Extraemos los parametros del modelo a validar
kernel_size=int(config["cnn-rnn-ctc"]["kernel_size"])
num_conv1=int(config["cnn-rnn-ctc"]["num_conv1"])
num_conv2=int(config["cnn-rnn-ctc"]["num_conv2"])
num_conv3=int(config["cnn-rnn-ctc"]["num_conv3"])
num_conv4=int(config["cnn-rnn-ctc"]["num_conv4"])
num_conv5=int(config["cnn-rnn-ctc"]["num_conv5"])
num_rnn=int(config["cnn-rnn-ctc"]["num_rnn"])
num_fc=int(config["cnn-rnn-ctc"]["num_fc"])

num_classes=int(config["cnn-rnn-ctc"]["num_classes"])
ctc_input_len=int(config["cnn-rnn-ctc"]["ctc_input_len"])

# Creamos el modelo ANN
model = ANN_model.CNN_RNN CTC(kernel_size, num_conv1, num_conv2, num_conv3, num_conv4,
                               num_conv5, num_rnn, num_fc, height, width, num_classes)

graph=model[0]
inputs=model[1]
targets=model[2]
keep_prob=model[3]
seq_len=model[4]
optimizer=model[5]
cost=model[6]
ler=model[7]
decoded=model[8]

# Declaramos el DataFrames para almacenar el resultado del entrenamiento y la validación.
train_result = pd.DataFrame()
val_result1 = pd.DataFrame()
val_result2 = pd.DataFrame()

# Creamos la sesión con el modelo previamente cargado.
with tf.Session(graph=graph) as session:
    # Guardamos la arquitectura del modelo e inicializamos sus variables
    saver=tf.train.Saver()
    tf.global_variables_initializer().run()

    # Inicializamos el LER a 1.
    LER=1.0

```



```

# Bucle de épocas.
for curr_epoch in range(num_epochs):

    # Extraemos un lote aleatorio del Dataset de entrenamiento.
    train_inputs, train_targets, original, train_seq_len = hw_utils.extract_training_batch(ctc_input_len, batch_size,
                                                                                          im_path, csv_path + "train.csv")

    feed = {inputs: train_inputs, targets: train_targets, keep_prob: 0.5, seq_len: train_seq_len}

    # Ejecutamos "optimizer", minimizando el error para el lote extraído.
    _ = session.run([optimizer], feed)

    # Comprobamos el periodo de validación para el modelo.
    if curr_epoch % val_period == 0:
        # Calculamos el error de la CTC y el LER para el dataset de entrenamiento y almacenamos a los resultados.
        train_cost, train_ler = session.run([cost, ler], feed)
        train_tuple = {'epoch': [curr_epoch], 'train_cost': [train_cost], 'train_ler': [train_ler]}
        train_result = pd.concat([train_result, pd.DataFrame(train_tuple)])

        # Realizamos la validación del modelo para los dos Datasets de validación y almacenamos los resultados
        val_tuple1 = hw_utils.validation(curr_epoch, ctc_input_len, batch_size, im_path,
                                         csv_path + "validation1.csv", inputs, targets, keep_prob, seq_len, session,
                                         cost, ler)
        val_result1 = pd.concat([val_result1, pd.DataFrame(val_tuple1)])

        val_tuple2 = hw_utils.validation(curr_epoch, ctc_input_len, batch_size, im_path, csv_path + "validation2.csv", inputs,
                                         targets, keep_prob, seq_len, session,
                                         cost, ler)
        val_result2 = pd.concat([val_result2, pd.DataFrame(val_tuple2)])

        # Comprobamos si se ha obtenido un LER menor al mínimo obtenido hasta el momento.
        if (float(val_tuple1['val_ler'][0]) + float(val_tuple2['val_ler'][0])) / 2 <= LER:
            # Almacenamos el valor de las variables.
            save_path = saver.save(session, checkpoints_path + "checkpoint_epoch_" +
                                   str(curr_epoch) + "_ler_" + str((float(val_tuple1['val_ler'][0]) + float(val_tuple2['val_ler'][0])) / 2) + ".ckpt")
            print("Model saved in file: " + str(save_path))

            # Actualizamos el valor mínimo de LER.
            LER = (float(val_tuple1['val_ler'][0]) + float(val_tuple2['val_ler'][0])) / 2

    # Comprobamos el periodo de impresión de ejemplos.
    if curr_epoch % print_period == 0:

        # Imprimimos el error cometido para el Dataset de validación en la época actual.
        print("Epoch: " + str(curr_epoch) + " val_cost: " +
              str((float(val_tuple1['val_cost'][0]) + float(val_tuple2['val_cost'][0])) / 2) +
              " val_ler: " + str((float(val_tuple1['val_ler'][0]) + float(val_tuple2['val_ler'][0])) / 2))

        # Imprimimos la salida del modelo para 10 ejemplos al azar del dataset de validación.
        print("Examples:")
        for j in range(10):

            # Extraemos una muestra.
            prob_inputs, prob_targets, prob_original, prob_seq_len, _ = hw_utils.extract_ordered_batch(ctc_input_len, 1,
                                                                                                    im_path, csv_path + "validation1.csv", randint(0, 6086))

            prob_feed = {inputs: prob_inputs,
                         targets: prob_targets,
                         keep_prob: 1,
                         seq_len: prob_seq_len}

            # Obtenemos la salida y la mapeamos como una palabra para imprimirla por pantalla.
            prob_d = session.run(decoded[0], feed_dict=prob_feed)
            output = str(list(map(dct.get, list(prob_d.values))))
            for ch in ["'", '"', "'", '"']:
                output = output.replace(ch, "")
                prob_original = str(prob_original).replace(ch, "")
            print("Target: " + prob_original + " Model Output: " + output)

```

```
# Almacenamos los resultados
val_result1.to_csv(results_path+"validation_result1.csv",index=False)
val_result2.to_csv(results_path+"validation_result2.csv",index=False)
train_result.to_csv(results_path+"training_result.csv",index=False)
print("THE TRAINING IS OVER")

if __name__ == '__main__':
    run_ctc()
```

test.py.

Script para testear un sistema previamente entrenado con el script anterior.

Este proceso carga el estado del sistema que mejores resultados ha dado y realiza una evaluación con el dataset de test. Como resultado se obtiene el LER medio para todas las muestras y un fichero con todas las transcripciones realizadas.

```
import os
import sys
from random import randint
import tensorflow as tf
import hw_utils
import pandas as pd
import ANN_model
import json
import ast
import numpy as np

def run_ctc():

    # Ruta del archivo de configuración, pasada por argumentos o por defecto "./config.json".
    if len(sys.argv) == 1:
        print("Execution without arguments, config file by default: ./config.json")
        config_file = str('./config.json')

    elif len(sys.argv) == 2:
        print("Execution with arguments, config file:" + str(sys.argv[1]))
        config_file = str(sys.argv[1])

    else:
        print()
        print("ERROR")
        print("Wrong number of arguments. Execute:")
        print(">> python3 test.py [path_config_file]")
        exit(1)

    # Cargamos el archivo de configuración
    try:
        config = json.load(open(config_file))
    except FileNotFoundError:
```

```

print()
print("ERROR")
print("No such config file : " + config_file)
exit(1)

# Si el directorio destino no existe, se crea.
if not os.path.exists(str(config["IAM-test"]["results_path"])):
    os.mkdir(str(config["IAM-test"]["results_path"]))

# Extraemos las variables generales para el test.
im_path=str(config["general"]["processed_data_path"])
csv_path=str(config["IAM-test"]["csv_path"])
results_path=str(config["IAM-test"]["results_path"])
checkpoints_path=str(config["IAM-test"]["checkpoints_path"])
height = int(config["general"]["height"])
width = int(config["general"]["width"])
dct=ast.literal_eval(str(config["general"]["dictionary"]))

# Extraemos los parametros del modelo a validar
kernel_size=int(config["cnn-rnn-ctc"]["kernel_size"])
num_conv1=int(config["cnn-rnn-ctc"]["num_conv1"])
num_conv2=int(config["cnn-rnn-ctc"]["num_conv2"])
num_conv3=int(config["cnn-rnn-ctc"]["num_conv3"])
num_conv4=int(config["cnn-rnn-ctc"]["num_conv4"])
num_conv5=int(config["cnn-rnn-ctc"]["num_conv5"])
num_rnn=int(config["cnn-rnn-ctc"]["num_rnn"])
num_fc=int(config["cnn-rnn-ctc"]["num_fc"])
num_classes=int(config["cnn-rnn-ctc"]["num_classes"])
ctc_input_len=int(config["cnn-rnn-ctc"]["ctc_input_len"])

# Creamos el modelo ANN
model = ANN_model.CNN_RNN_CTC(kernel_size, num_conv1, num_conv2, num_conv3, num_conv4,
                               num_conv5, num_rnn, num_fc, height, width, num_classes)

graph=model[0]
inputs=model[1]
targets=model[2]
keep_prob=model[3]
seq_len=model[4]
cost=model[6]
ler=model[7]
decoded=model[8]

# Declaramos el DataFrames para almacenar la salidas del modelo para el Dataset completo.
result_test = pd.DataFrame()

# Creamos la sesión con el modelo previamente cargado.
with tf.Session(graph=graph) as session:

    # Restauramos el valor de las variables del último modelo entrenado.
    saver = tf.train.Saver()
    saver.restore(session, tf.train.latest_checkpoint(checkpoints_path))
    print("Loaded Model")

    # Variables auxiliares.
    cont = 0
    total_test_cost = 0
    total_test_ler = 0

    # Bucle para realizar la validación sobre el Dataset completo
    while cont >= 0:

```

```

# Extraemos las muestras 1 a 1 y de forma secuencial mediante "cont".
test_inputs, test_targets, original, test_seq_len, num_samples = hw_utils.extract_ordered_batch(
    ctc_input_len, 1, im_path, csv_path+ "test.csv", cont)

# Se ha conseguido extraer la muestra para testearla.
if num_samples==1:

    # Calculamos el error de la CTC y el LER para dicha muestra.
    test_feed = {seq_len: test_seq_len,
                  inputs: test_inputs,
                  keep_prob: 1,
                  targets:test_targets}
    test_cost, test_ler= session.run([cost, ler], test_feed)
    total_test_cost += test_cost
    total_test_ler += test_ler

    # Obtenemos la salida del modelo y la mapeamos como una palabra para almacenarla junto a la palabra objetivo.
    dec=session.run(decoded[0],test_feed)
    output = str(list(map(dct.get, list(dec.values))))
    for ch in ["'", '"', ',', '']:
        output = output.replace(ch, "")
        original=str(original).replace(ch, "")
    tuple = {'Target': [original], 'Output': [output]}
    result_test = pd.concat([result_test, pd.DataFrame(tuple)])
    cont += 1

# No quedan más muestras en el Dataset de test.
else:

    # Imprimimos los resultados del test y almacenamos las salidas del modelo.
    print("IAM test result:")
    print("Cost: "+str(total_test_cost / (cont)))
    print("LER: "+str(total_test_ler / (cont)))
    result_test.to_csv(results_path+"test_result.csv",index=False)
    cont = -1

if __name__ == '__main__':
    run_ctc()

```

config.json.

Fichero en formato JSON que contiene los parámetros generales para el funcionamiento de los scripts y los hiperparámetros del modelo a entrenar y evaluar.

```
{
  "general": {
    "raw_data_path": "./Data/words/",
    "processed_data_path": "./Data/Images/",
    "csv_path": "./Data/appropriate_images.csv",
    "height": 64,
    "width": 1024,
    "dictionary": "{0:'0', 1:'1', 2:'2', 3:'3', 4:'4',
                  5:'5', 6:'6', 7:'7', 8:'8', 9:'9', 10:'A',
                  11:'B', 12:'C', 13:'D', 14:'E', 15:'F',
                  16:'G', 17:'H', 18:'I', 19:'J', 20:'K',
                  21:'L', 22:'M', 23:'N', 24:'O', 25:'P',
                  26:'Q', 27:'R', 28:'S', 29:'T', 30:'U',
                  31:'V', 32:'W', 33:'X', 34:'Y', 35:'Z',
                  36:'a', 37:'b', 38:'c', 39:'d', 40:'e',
                  41:'f', 42:'g', 43:'h', 44:'i', 45:'j',
                  46:'k', 47:'l', 48:'m', 49:'n', 50:'o',
                  51:'p', 52:'q', 53:'r', 54:'s', 55:'t',
                  56:'u', 57:'v', 58:'w', 59:'x', 60:'y', 61:'z'}"
  },
  "cnn-rnn-ctc": {
    "kernel_size": 5,
    "num_conv1": 20,
    "num_conv2": 50,
    "num_conv3": 100,
    "num_conv4": 200,
    "num_conv5": 400,
    "num_rnn": 200,
    "num_fc": 200,
    "num_classes": 63,
    "ctc_input_len": 32
  },
  "cross-validation": {
    "csv_path": "./CSV/Cross-Validation/",
    "results_path": "./Results/Cross-Validation/",
    "num_epochs": 20000,
    "validation_period": 50,
    "print_period": 1000,
    "batch_size": 500
  },
  "IAM-test": {
```

```
"csv_path": "./CSV/IAM test/",  
"results_path": "./Results/IAM test/",  
"checkpoints_path": "./Checkpoints/",  
"num_epochs": 50000,  
"validation_period": 50,  
"print_period": 1000,  
"batch_size": 500  
}  
}
```

Anexo B. Manual de Uso.

Sistema de Deep Learning para el Reconocimiento de Palabras Manuscritas implementado en TensorFlow y entrenado con IAM Handwriting Database.

Sobre este sistema se realiza una validación cruzada y el test IAM.

1. Deep Learning para el Reconocimiento de Texto Manuscrito implementado en TensorFlow.

Sistema de Deep Learning para el Reconocimiento de Palabras Manuscritas implementado en TensorFlow y entrenado con IAM Handwriting Database.

Sobre este sistema se realiza una validación cruzada y el test IAM.

1.1. Estructura

1.1.1. Ficheros Python

- *clean_IAM.py*: Script para limpieza y el preprocesamiento de las imágenes.
- *ANN_model.py*: Modelo de la red neuronal implementada en TensorFlow.
- *cross-validation.py*: Script para la validación cruzada.
- *train.py*: Script para entrenar el modelo y almacenar los parámetros que consiguen un mejor resultado.
- *test.py*: Script para testear un modelo previamente entrenado.
- *hw_utils.py*: Funciones útiles en distintas partes del proyecto.

1.1.2. Ficheros CSV

- *CSV/Cross-Validation*: Ficheros que contienen los nombres y las transcripciones de las imágenes de los distintos subconjuntos de entrenamiento y validación.
- *CSV/IAM test*: Ficheros que contienen los nombres y las transcripciones de las imágenes para el test IAM.
- *Data/appropriate_images.csv*: Fichero CSV que contiene el nombre de las imágenes aptas (87.108).

1.1.3. Ficheros de Configuración

Fichero que contiene todos los parámetros del proyecto:

- **general**: Parámetros generales del proyecto.
 - *raw_data_path*: Ruta a las imágenes sin preprocesar.
 - *processed_data_path*: Ruta para las imágenes preprocesadas.
 - *csv_path*: Ruta al CSV de imágenes aptas.
 - *height*: Altura de las imágenes preprocesadas.
 - *width*: Anchura de las imágenes preprocesadas.
 - *dictionary*: Diccionario para parsear las etiquetas.
- **cnn-rnn-etc**: Hiperparámetros del modelo.
 - *kernel_size*: Altura y anchura de los filtros de la CNN, filtros cuadrados.
 - *num_conv1*: Número de filtros de la 1ª capa CNN.

- num_conv2: Número de filtros de la 2ª capa CNN.
- num_conv3: Número de filtros de la 3ª capa CNN.
- num_conv4: Número de filtros de la 4ª capa CNN.
- num_conv5: Número de filtros de la 5ª capa CNN.
- num_rnn: Número de neuronas de las capas RNNs.
- num_fc: Número de neuronas de la 1ª capa Fullconnect.
- num_classes: Número de etiquetas, incluida la etiqueta "blanco".
- ctc_input_len: Longitud de la secuencia de entrada a la CTC.
- cross-validation: Parámetros para la validación cruzada.
 - csv_path: Ruta a los CSVs para la validación.
 - results_path: Ruta para los resultados.
 - num_epochs: Número de épocas.
 - validation_period: Periodo de épocas para realizar la validación del modelo.
 - print_period: Periodo de épocas para la impresión por pantalla.
 - batch_size: Tamaño del lote de muestras.
- IAM-test: Parámetros para el test IAM.
 - csv_path: Ruta a los CSVs para el test.
 - results_path: Ruta para los resultados.
 - checkpoints_path: Ruta para almacenar el modelo entrenado.
 - num_epochs: Número de épocas.
 - validation_period: Periodo de épocas para realizar la validación del modelo.
 - print_period: Periodo de épocas para la impresión por pantalla.
 - batch_size: Tamaño del lote de muestras.

1.2. Primeros Pasos

1.2.1. Requisitos

Python 3.6 y librerías:

- TensorFlow 1.3
- PIL
- Pandas
- Numpy
- Json
- Ast

1.2.2. Instalación y preprocesado de datos

Tras descargar o clonar el repositorio es necesario descargar el dataset de la IAM Handwriting Database y descomprimirlo en el directorio "Offline-Handwriting-Recognition-with-TensorFlow\Data".

Una vez hemos conseguido el dataset ejecutamos:

```
python3 clean_IAM.py [path_config_file]
```

Si no se añade ninguna ruta al archivo de configuración se tomará la ruta por defecto `./config.json`

Este script selecciona las imágenes aptas para las pruebas, las reescala y le añade relleno hasta igualar sus dimensiones.

1.3. Ejecución

1.3.1. Cross-Validation

Para realizar la validación cruzada del modelo solo es necesario ejecutar:

```
python3 cross-validation.py [path_config_file]
```

Este script realiza 10 validaciones con distintas subdivisiones del dataset original y almacena los resultados en formato CSV.

1.3.2. Test IAM

El primer paso es entrenar el modelo con el dataset ofrecido por IAM con unas subdivisiones específicas. Para ello ejecutamos:

```
python3 train.py [path_config_file]
```

Este script realiza un entrenamiento del modelo y almacena los parámetros que mejor resultado han dado para el dataset de validación.

Una vez tenemos el modelo entrenado, obtenemos el resultado del test ejecutando:

```
python3 test.py [path_config_file]
```

El resultado se muestra por pantalla y las salidas del sistema se almacenan en CSV.