UNIVERSIDAD DE VALLADOLID

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS DE TELECOMUNICACIÓN

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA DE TECNOLOGÍAS DE TELECOMUNICACIÓN

Reconocimiento de actividades físicas con sensores inerciales y Redes Neuronales de Aprendizaje Profundo

Autor:

D. Sergio Sáez Bombín

Tutor:

Dr. D. Mario Martínez Zarzuela

Valladolid, 4 de julio de 2018

TÍTULO:	**Reconocimiento de actividades físicas con sensores inerciales y Redes Neuronales de Aprendizaje Profundo**
AUTOR:	**D. Sergio Sáez Bombín**
TUTOR:	**Dr. D. Mario Martínez Zarzuela**
DEPARTAMENTO:	**Teoría de la Señal y Comunicaciones e Ingeniería Telemática**

TRIBUNAL

PRESIDENTE:	**Dr. D. Francisco J. Díaz Pernas**
SECRETARIO:	**Dr. D. Mario Martínez Zarzuela**
VOCAL:	**Dr. D. David González Ortega**
SUPLENTE 1:	**Dra. Dª. Miriam Antón Rodríguez**
SUPLENTE 2:	**Dr. D. Carlos Gómez Peña**

FECHA:	**4 de julio de 2018**
CALIFICACIÓN:	

Agradecimientos

Me gustaría manifestar mi agradecimiento al Dr. D. Mario Martínez Zarzuela por sus consejos y ayuda en la realización de este Trabajo Fin de Grado.

A mis amigos y a mi pareja, dispuestos siempre a escucharme y a darme su punto de vista, así como a apoyarme en mis decisiones.

Por último, a mis padres, mi hermano y mi hermana, por su interés, ánimo y ayuda en todo momento.

Resumen

El objetivo de este Trabajo Fin de Grado consiste en el desarrollo de una red neuronal vía software capaz de clasificar actividades físicas a partir de un set limitado de ellas, teniendo que generalizar, la propia red, patrones o características propias de cada actividad para poder reconocerlas sin importar el sujeto que las esté realizando.

En primer lugar, estudiaremos las posibles utilizaciones que se dan a las redes neuronales y su uso con datos provenientes de sensores inerciales. A continuación, se expondrá la situación en la que se encuentra hoy en día el reconocimiento de actividades físicas mediante el uso de la Inteligencia Artificial y más en concreto, del Deep Learning. Tras este estudio, se presentarán los fundamentos matemáticos y teóricos en los que se basa el diseño de redes neuronales, con el objetivo de justificar las decisiones de diseño que se han llevado a cabo. Finalmente, se describirán las redes neuronales diseñadas y se presentarán sus resultados, terminando con las conclusiones sacadas y el planteamiento de las posibles líneas futuras a seguir raíz de este Trabajo Fin de Grado.

Palabras clave

Reconocimiento de actividades humanas, sensores inerciales, Inteligencia Artificial, Deep Learning, Redes Neuronales.

Abstract

The goal of this End-of-Degree Project is the development of a neural network via software capable of classifying physical activities from a limited set, having to generalize patterns or features from each activity to be to recognize them regardless the subject that performs them.

First of all, we will study the posible use given to the neural networks and its use with data from inertial sensors. Then, the situation in which the acticity recognition through the use of the Artficial Intelligence and more specifically, of Deep Learning will be exposed. After this study, the mathematical and theoretical fundamentals in which the design of neural networks is based, with the aim of justifying the design decisions that have been carried out, will be presented. Finally, the designed neural networks will be described and their results presented, ending with the conclusions drawn and the approach of the posible future lines to follow root of this End-of-Degree Project.

Keywords

Human activity recognition, inertial sensors, Artificial Intelligence, Deep Learning, Neural Networks.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1 Introducción

En este capítulo se ubica el marco de trabajo de este Trabajo Fin de Grado, describiendo los objetivos de éste.

Este Trabajo de Fin de Grado se ha desarrollado en el Grupo de Telemática e Imagen (GTI) de la Universidad de Valladolid, el cual, utiliza desde hace tiempo en sus trabajos los sensores inerciales (IMUs, *Inertial Measurement Units*) para determinar la correcta realización de un movimiento en juegos serios interactivos con el objetivo de la recuperación de la movilidad de usuarios (rehabilitación) mediante su utilización. En este contexto, la recogida y procesamiento de los datos provenientes de estos sensores conlleva un gran trabajo de programación con el objetivo de tener en cuenta todas las posibles combinaciones de los movimientos.

En este sentido, este Trabajo de Fin de Grado busca sentar las bases para evitar esa tediosa programación y, en el mejor de los casos, mejorarla, mediante la realización de una red neuronal profunda (DNN, *Deep Neural Network*) que sea capaz de "aprender" los distintos tipos de actividades y movimientos con los que se desee trabajar y así comprobar de manera más sencilla y eficaz la correcta realización de estos por parte del usuario.

Estas redes neuronales se ubican en el ámbito del Deep Learning, el cual se puede considerar una rama del Machine Learning, y, por lo tanto, de la Inteligencia Artificial. Esto hace que, el principal objetivo de este Trabajo de Fin de Grado sea ver la capacidad del Deep Learning para el reconocimiento de actividades físicas a partir de los datos de sensores inerciales.

1.1 Motivación y objetivos

El reconocimiento automático de actividades físicas humanas es conocido como Reconocimiento de la Actividad Humana (HAR, *Human Activity Recognition*), y ha surgido como un área de gran interés en diversos sectores, como son los deportes, el entretenimiento y, el que más se acerca al ámbito de este Trabajo Fin de Grado, el sector de la salud (San *et al.*, 2017, pp. 186-204).

Este creciente interés viene favorecido por el desarrollo de sistemas multi-sensoriales, que utilizan la combinación de sensores inerciales sobre el cuerpo (*wearable on-body sensors*), como bien pueden ser los acelerómetros, giroscopios o sensores de campo magnético.

El problema del HAR está basado en la premisa de que los movimientos del cuerpo se trasladan a patrones específicos de señales provenientes de sensores, que pueden ser

detectados y clasificados mediante algoritmos de aprendizaje de Machine Learning (Ordóñez, F. J. y Roggen, D., 2016).

Es por ello por lo que, en los últimos años, el HAR basado en sensores sobre el cuerpo humano ha sufrido un prometedor progreso en aplicaciones como videoconsolas, entrenamiento fitness personal, toma de medicamentos y monitorización de la salud., ya que la utilización de sensores sobre el cuerpo permite el reconocimiento de las actividades y de su contexto sin importar la ubicación del usuario (Ordóñez, F. J. y Roggen, D., 2016).

Sin embargo, anteriormente, las cámaras de vídeo eran las más utilizadas para el reconocimiento de actividades, ya que pueden tomar la imagen de las actividades (Hassan, M. M. *et al*, 2018), pero, como se indica en (San *et al.*, 2017, pp. 186-204), según (Bulling, A. *et al.*, 2014), las señales procedentes de los sensores son más favorables para el problema HAR que las obtenidas de una cámara de vídeo, debido principalmente a 3 razones:

- Los sensores sobre el cuerpo reducen las limitaciones del entorno y los marcos estacionarios que sufren las cámaras de vídeo.
- Utilizar varios sensores sobre el cuerpo permite una captura más precisa y efectiva de la señal.
- Los sensores sobre el cuerpo disfrutan de las ventajas de la privacidad de la información, ya que las señales son adquiridas de un objetivo concreto, mientras que en las señales captadas por las cámaras de vídeo también pueden contener información de otros sujetos de la escena que no son objetivo de estudio.

En contra de los sensores sobre el cuerpo está la variabilidad entre clases, ya que, distintos sujetos realizarán la misma actividad con movimientos distintos, así como la semejanza entre clases, por ejemplo, entre el *jogging* y el *running*, el movimiento del cuerpo es semejante, pero son actividades distintas (San *et al.*, 2017, pp. 186-204).

El factor clave atribuido al éxito del HAR es encontrar una representación efectiva de las series temporales recogidas de sensores sobre el cuerpo. Convencionalmente, el problema HAR se considera como una aplicación específica del análisis de series temporales.

Recientemente, el Deep Learning (las redes neuronales de aprendizaje profundo) ha aparecido como una familia de modelos de aprendizaje que ayudan a modelar abstracciones de alto nivel sobre los datos (Bengio Y., 2009, citado en San *et al.*, 2017, pp. 188), ya que procesan información no lineal para la extracción de características y su clasificación, organizadas de forma jerárquica, en la que cada capa de la red procesa los datos de la capa anterior. Además, permite utilizar unas capas de la red para la extracción de características de los datos, algo que en otros algoritmos de Machine Learning no es posible y debe hacerse como procesado previo al algoritmo utilizado.

Tras la revisión bibliográfica, en búsqueda de un dataset idóneo para trabajar con él, y tras un estudio de diferentes trabajos que tratan el problema HAR de diferentes maneras, que se verá en el apartado 3.4, se llegó a la conclusión de que no había una gran cantidad de trabajos que utilizaran técnicas de Deep Learning para resolver el problema HAR con el dataset que se escogió para este trabajo, viendo en ese hecho una oportunidad para realizar un estudio en esa dirección.

Por lo tanto, **el objetivo principal de este Trabajo Fin de Grado es explorar mediante el estudio y diseño de modelos de redes neuronales de aprendizaje profundo el reconocimiento de actividades físicas de un dataset de movimientos capturado con sensores inerciales**, con la intención de servir de base para una red neuronal que permita, mediante los sensores del Grupo de Telemática e Imagen, el reconocimiento de movimientos o actividades físicas para los juegos serios que en él se desarrollan y en definitiva, para resolver el problema HAR.

1.2 Fases y métodos

Las fases en las que se ha dividido este Trabajo Fin de Grado se exponen brevemente a continuación:

- **Búsqueda de datasets**: Focalizada en encontrar un dataset que contuviera los datos de actividades lo más semejantes posible a las que se realizarían en la rehabilitación de un paciente.
- **Revisión bibliográfica de trabajos HAR**: Para realizar un sistema de Deep Learning para resolver el problema HAR, es necesario conocer el Estado del Arte del problema, y ver cómo se afronta tanto con técnicas de Deep Learning como con técnicas de Machine Learning en general.
- **Instalación del entorno de trabajo**: Para poder realizar cualquier sistema de Deep Learning ha sido necesario el acondicionamiento de un servidor. Para ello se han instalado una serie de herramientas software desarrolladas para realizar trabajos de Deep Learning. Esta fase de trabajo se puede ver en el ANEXO I.
- **Estudio de Deep Learning y Tensorflow**: Para poder realizar un sistema de Deep Learning, primero es necesario conocer los conceptos teóricos subyacentes a esta técnica. Una vez estudiados los conceptos teóricos, hay que estudiar cómo implementarlos, para lo que se ha utilizado, esencialmente Tensorflow, que dispone de una serie de bibliotecas de funciones diseñadas para el Deep Learning. Esta fase de trabajo se expone en el ANEXO II.
- **Acondicionamiento del dataset**: Tras el cumplimiento de las fases anteriores, ya se está en condiciones de comenzar a diseñar el sistema de Deep Learning. Para ello, en primer lugar, se debe realizar un preprocesado sobre los datos del dataset siguiendo algunas ideas de otros trabajos.
- **Propuesta de redes**: Una vez listos los datos, ya se puede comenzar a tomar decisiones de diseño de lase redes neuronales.
- **Pruebas realizadas**: Para cada decisión de diseño, se realizan pruebas y un estudio de los resultados, con el objetivo de ajustar los parámetros de la red para obtener un mejor resultado. Una vez se considere que se ha acanzado el mejor resultado posible con una red, se vuelve a la fase anterior y se proponen más redes para después pasar otra vez a esta última fase, realizándose este proceso de forma iterativa hasta obtener un resultado satisfactorio.

1.3 Medios disponibles

1.3.1 Hardware

Este Trabajo Fin de Grado, siguiendo las fases anteriormente citadas, se ha desarrollado con el siguiente equipamiento hardware disponible en el servidor configurado para el trabajo (dilbert), disponible en el Laboratorio de GTI:

- Dos procesadores Intel® Xeon® X5650 con las siguientes características cada uno (Intel Corporation, n.d.):
    - Velocidad de reoj: 2.67 GHz
    - Número de núcleos: 6
    - Cantidad de subprocesos: 12
- Memoria RAM: 24 GB
- Dos tarjetas gráficas:
    - GeForce GTX 1080 Ti (NVIDIA Corporation, 2017)
        - CUDA *Cores*: 3584
        - Frecuencia de reloj normal: 1480 MHz
        - Frecuencia de reloj acelerada: 1582 MHz
        - Velocidad de la memoria: 11 Gb/s
        - Configuración de memoria estándar: 11GB GDDR5X
        - Ancho de banda de memoria: 484 GB/s
    - GeForce GTX 970 (NVIDIA Corporation, 2014)
        - CUDA *Cores*: 1664
        - Frecuencia de reloj normal: 1050 MHz
        - Frecuencia de reloj acelerada: 1178 MHz
        - Velocidad de la memoria: 7 Gb/s
        - Configuración de memoria estándar: 4 GB GDDR5
        - Ancho de banda de memoria: 224 GB/s

Estas características son suficientes para el trabajo a realizar. Solamente comentar que, los tiempos de entrenamiento que se expondrán más adelante no serán completamente reales, ya que la mayoría del tiempo, este servidor ha sido compartido por varios usuarios, y, por lo tanto, los recursos han tenido que ser repartidos, no pudiendo disfrutar de toda la capacidad del servidor.

1.3.2 Software

En cuanto al software utilizado en este Trabajo Fin de Grado cabe destacar lo siguiente:

Python

El lenguaje de programación utilizado en este Trabajo Fin de Grado ha sido Python.

Python es un lenguaje de programación interpretado y orientado a objetos, caracterizado por su sintaxis clara y legible. Es relativamente fácil de aprender y es portable, es decir, puede ser interpretado por un gran número de Sistemas Operativos, como pueden ser los sistemas basado en UNIX, Mac OS, MS-DOS, 0S/2 y Windows (Python, 2018).

Fue creado por Guido van Rossum, procedente de Holanda, y el código es de libre acceso.

Una característica notable de Python es su indentado en las sentencias de fuente para hacer más sencilla la lectura. Python ofrece tipos de datos dinámicos, clases de lectura e interfaces para varias llamadas al sistema y bibliotecas. Puede ser extendido, usando el lenguaje C o C++.

En la realización de este Trabajo Fin de Grado se ha hecho uso de la versión 2.7 de este lenguaje.

Figura 1. Logotipo de Python.

Tensorflow

Tensorflow es una biblioteca software de código abierto para computación numérica utilizando grafos de flujos de datos. Fue originalmente desarrollada por Google Brain Team en la organización de investigación de Machine Intelligence de Google para la investigación en Machine Learning y Redes Neuronales Profundas, pero el sistema es suficientemente aplicable a una amplia variedad de dominios (Tensorflow, 2018).

Es una plataforma cruzada y funciona sobre casi todo: GPUs y CPUs, incluyendo móviles y plataformas embebidas e incluso en unidades de procesamiento de tensores (TPUs), que son hardware especializado sobre el que realizar matemática de tensores.

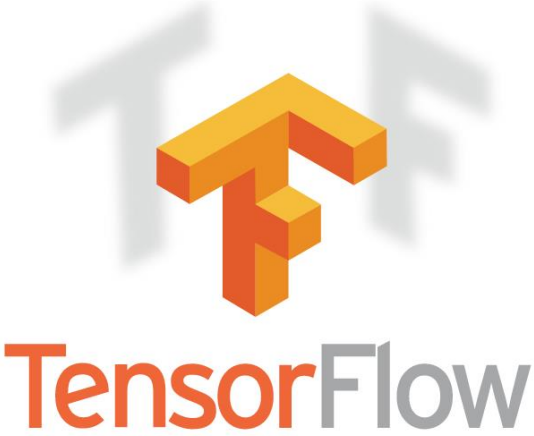


Figura 2. Logotipo de Tensorflow.

Jupyter

El Proyecto Jupyter existe para desarrollar software de código abierto, estándares abiertos y servicios para computación interactiva a través de docenas de lenguajes de programación, en concreto más de 40, entre los que se incluyen algunos como Python, R, Julia y Scala.

El uso que se ha hecho de este Proyecto Jupyter ha sido a través del Notebook de Jupyter, el cual es una aplicación web de código abierto que permite crear y compartir documentos (vía email, Dropbox, GitHub y Jupyter Notebook Viewer) que contienen código *"vivo"*, ecuaciones, visualizaciones y texto narrativo. Las salidas de estos Notebooks son interactivas y son muy ricas en cuanto a variedad, ya que permiten salidas del tipo HTML, imágenes, vídeos, LaTeX y tipos personalizados de MIME (Jupyter, 2018).

Entre otros usos caben destacar: limpieza de datos y transformaciones, simulación numérica, modelado estadístico, visualización de datos y muchos más, pero de cara a este Trabajo Fin de Grado importa destacar su ventaja de permitir hacer uso de herramientas y bibliotecas del mundo del Big Data, como pueden ser: pandas, scikit-learn y Tensorflow, todas ellas utilizadas durante este Trabajo Fin de Grado.

Para terminar con esta breve explicación acerca del Proyecto Jupyter, cabe mencionar que la versión que se ha utilizado para este Trabajo Finde Grado ha sido JupyterHub, que es una versión multi-usuario diseñada para compañías, clases y laboratorios de investigación. Esta versión permite el manejo de los usuarios de forma unificada y un acceso a los datos con una autentificación con PAM, OAuth o un sistema propio, además permite la distribución de Jupyter Notebook a los usuarios de la organización en una infraestructura centralizada y es compatible con Docker y Kubernetes para escalar mediante contenedores esta distribución, aislar los procesos de cada usuario y simplificar la instalación de software (Jupyter, 2018).

Figura 3. Logotipos de Jupyter y JupyterHub.

Docker

Docker es la compañía puntera en el movimiento de los contenedores y la única plataforma proveedora de contenedores que permite el uso de todas las aplicaciones a través de los distintos sistemas.

Docker permite una independencia verdadera entre aplicaciones e infraestructura y los desarrolladores liberan su potencial para crear un modelo para una mejor innovación y colaboración. Docker se caracteriza por: su agilidad (acelera el desarrollo de software en un factor de 13 y responde instantáneamente a lo que el diseñador necesita), su portabilidad (elimina el trabajo en la propia máquina de una vez por todas, gana independencia entre los distintos ambientes), su seguridad (distribuye aplicaciones más segura mediante capacidades de seguridad propias y configuraciones *"out of the box"* y su capacidad de reducir costes (optimiza el uso de los recursos de las infraestructuras y las operaciones de optimización para reducir en un 50% los costes) (Docker, 2018).

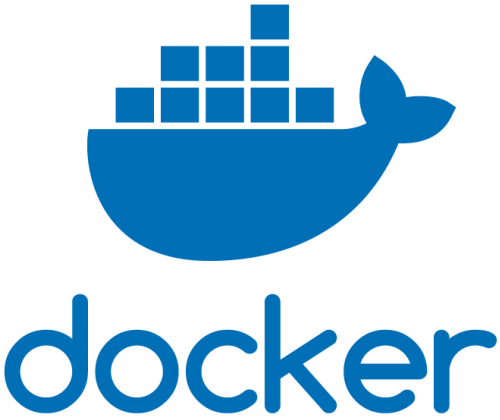


Figura 4. Logotipo de Docker.

DIGITS

Esta plataforma es el Sistema de Entrenamiento con GPUs de Deep Learning perteneciente a NVIDIA y pone al alcance de los ingenieros y *data sicentists* el poder del Deep Learning.

DIGITS puede ser utilizado para entrenar la mayoría de las Redes Neuronales Profundas para tareas de clasificación de imágenes, segmentación y detección de objetos.

Simplifica las tareas comunes del Deep Learning como el manejo de datos, diseño y entrenamiento de redes neuronales en sistemas multi-GPU, monitorear el rendimiento en tiempo real con visualizaciones avanzadas y seleccionar el mejor modelo en cuanto a rendimiento a partir de los resultados. DIGITS es completamente interactivo con lo que los *data scientists* pueden centrarse en el diseño y entrenamiento de redes más que en programarlas y depurarlas (NVIDIA Corporation, 2018a).

Figura 5. Logotipo de DIGITS.

Preparación de un servidor para Deep Learning

La utilización de Jupyter, Tensorflow, Docker y NVIDIA DIGITS se puede ver en el ANEXO I, en el que se explica esta preparación del servidor, que fue necesaria para poder realizar el Trabajo de Fin de Grado.

1.4 Estructura de la memoria

La memoria está estructurada en capítulos, en los que se abordan los diferentes pasos y acciones realizados para la consecución del objetivo de este Trabajo Fin de Grado:

En el **primer capítulo** se ha realizado una introducción al ámbito en el que se enmarca este trabajo, el reconocimiento de actividades físicas, además de definir las motivaciones y objetivos de este.

En el **segundo capítulo**, se estudiarán los principios básicos de los sensores inerciales, fuente de los datos usados en todo problema de reconocimiento de actividades físicas, así como los fundamentos matemáticos de los cuaterniones, ya que son los datos que más utilizaremos en este trabajo.

El **tercer capítulo** versará sobre el Deep Learning. Se realizará una introducción y se comentará su historia para poder mostrar los hitos conseguidos y su potencial. Posteriormente, se explicará la base teórica del Deep Learning referente a este trabajo, se mostrarán algunos de sus fundamentos matemáticos para poder entender mejor el porqué de ciertos aspectos y se pondrán ejemplos para afianzar y aclarar las explicaciones más teóricas. Finalmente, se comentarán algunos trabajos en los que se ha basado este Trabajo Fin de Grado.

El **cuarto capítulo** muestra el estudio y diseño de distintos tipos de redes neuronales de aprendizaje profundo para el reconocimiento de actividades físicas, así como los resultados obtenidos para cada tipo de red.

Finalmente, en el **quinto capítulo**, se mostrarán las conclusiones sacadas de este Trabajo Fin de Grado y las líneas futuras que se pueden seguir en trabajos sucesivos que sirvan de complemento a este.

Además, se incluyen dos anexos:

- **ANEXO I**: En este anexo se explica cómo configurar un servidor de cara a realizar este Trabajo Fin de Grado, para permitir el manejo de la gran cantidad de datos necesarios en cualquier red de Deep Learning, así como la gestión de forma cómoda del trabajo simultáneo en Deep Learning de varios usuarios.
- **ANEXO II**: Aquí se comentan algunos cursos de interés que se realizaron de cara a preparar este Trabajo Fin de Grado y que pueden resultar de gran interés a modo de introducción al mundo de la Inteligencia Artificial y el Deep Learning.

Capítulo 2 Reconocimiento de movimientos con sensores inerciales

Como se ha expuesto en la introducción, los sensores inerciales son parte fundamental en el problema del Reconocimiento de Actividades Humanas. Es por ello, que se le dedica un capítulo en esta memoria, para explicar su principio de funcionamiento y cómo combinar los datos obtenidos para obtener los datos de nuestro interés, que, en este caso, son los cuaterniones.

2.1 IMUs

Antes de hablar sobre el reconocimiento de movimientos en sí, vamos a hablar de qué es un sensor inercial o Unidad de Medida Inercial (IMU):

Los sensores inerciales son dispositivos electrónicos de medida que permiten estimar la orientación de un cuerpo a partir de las fuerzas inerciales que experimenta este. Su principio de funcionamiento se basa en la medida de las fuerzas de aceleración (con el acelerómetro) y velocidad angular (con el giróscopo o giroscopio), además, son capaces de medir otras fuerzas, como la orientación del campo magnético (si disponen de un magnetómetro) y los cuaterniones.

Es el componente principal de sistemas de guía inercial utilizados en vehículos aéreos, espaciales, marinos y aplicaciones robóticas (González Alonso, J., 2017, pp. 103-106).

Las IMUs están compuestas, en su forma más básica, por un acelerómetro y un giroscopio o giróscopo, que generalmente serán triaxiales, para estimar su orientación en el espacio mediante la captura de una aceleración y una velocidad angular concretas (González Alonso, J., 2017, pp. 103-106).
En este sentido, los componentes que nos interesan para este Trabajo Fin de Grado, de los que vamos a obtener los datos son:

- **Acelerómetro**: Es un instrumento capaz de medir la aceleración en uno, dos o tres ejes. Un acelerómetro triaxial básico consiste en una masa suspendida por un muelle en un receptáculo, y puede moverse en la dirección de medida del acelerómetro. Por lo tanto, un acelerómetro triaxial está formado por tres acelerómetros triaxiales orientados ortogonalmente entre sí para ofrecer información de aceleración en el espacio tridimensional. Sin embargo, las IMUs incorporan acelerómetros integrados en silicio, utilizando la tecnología llamada

MEMS (*Micro-machined Electro Mechanical Sysrem*), que calculan la aceleración mediante el voltaje obtenido entre dos placas, variando una de ellas su posición dependiendo del movimiento del acelerómetro, con lo que son de carácter capacitivo (González Alonso, J., 2017, pp. 103-106).

- **Giroscopio o giróscopo**: Este instrumento es capaz de medir la velocidad angular de rotación del cuerpo con respecto al sistema de referencia inercial. La construcción de un giroscopio puede estar basada en diferentes diseños, pero, en el caso de las IMUS, están diseñados, al igual que los acelerómetros, con la tecnología MEMS. De esta forma, la salida del giroscopio será un voltaje que varía indicando, en grados por segundo (V/º/s), la velocidad angular sufrida por el sensor (González Alonso, J., 2017, pp. 103-106). Integrando esta velocidad angular obtenida, se puede obtener el ángulo de rotación del cuerpo sobre un eje. Al igual que antes, para obtener un giroscopio triaxial, se pueden montar tres giroscopios uniaxiales orientados ortogonalmente entre sí.
- **Magnetómetro**: Este es un elemento sensible al campo magnético y está incluido en algunas IMUs. Miden la fuerza y/o dirección del campo magnético con respecto al campo magnético terrestre (González Alonso, J., 2017, pp.103-106).

En los últimos años, las IMUs se han ido miniaturizando y siendo cada vez más accesibles a nivel económico, lo que permite disponer de sistemas de medida que pueden ser colocados por toco el cuerpo (*wearable on-body sensors*) y poder conocer su posición y movimiento en cualquier entorno.

Estas ventajas han llevado, como se ha indicado en la anterior sección de esta memoria, a su uso cada vez mayor en todo tipo de aplicaciones para la monitorización o reconocimiento de actividades físicas o movimientos. De esta forma, esta tecnología permite extraer patrones cinemáticos del movimiento humano de una forma más sencilla, sin la utilización de algoritmos complejos como en el caso de la captura del movimiento mediante cámaras de vídeo, también comentado en la sección anterior, ya que ene se caso se debe realizar un procesado de la imagen y un modelado, muy condicionados por el entorno, como pueden ser las condiciones de iluminación.

2.2 Cuaterniones

2.2.1 Definición

Los cuaterniones son números hiper-complejos (número complejo de rango superior a 2) de rango 4, que tienen la forma:

$$q = q_0 + \mathbf{q} = q_0 + \mathbf{i}q_1 + \mathbf{j}q_2 + \mathbf{k}q_3$$

Y teniendo en cuenta que:

$$\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1$$

2.2.2 Notación

Se utiliza **i**, **j** y **k** para denotar las bases ortonormales del espacio tridimensional $\mathbb{R}^3$. Estas bases las podemos escribir como:

$$\mathbf{i} = (1,0,0) \qquad \mathbf{j} = (0,1,0) \qquad \mathbf{k} = (0,0,1)$$

Por lo tanto, un cuaternión define un elemento en $\mathbb{R}^4$:

$$q = (q_0, q_1, q_2, q_3)$$

Siendo q_0, q_1, q_2 y q_3 números reales.

Por lo tanto, en la anterior ecuación:

$$q = q_0 + \mathbf{q} = q_0 + \mathbf{i}q_1 + \mathbf{j}q_2 + \mathbf{k}q_3$$

Tenemos que, q_0 es la parte escalar del cuaternión mientras que q es la parte vector del cuaternión, siendo q_0, q_1, q_2 y q_3 las componentes del cuaternión.

2.2.3 Rotación de cuaterniones

El interés de los cuaterniones en el estudio de la actividad y el movimiento mediante IMUs es que permiten obtener la orientación del sensor de una forma más sencilla y efectiva que cualquier otra medida de la orientación.

Para ello, es interesante conocer cómo se pueden obtener los cuaterniones de las medidas del acelerómetro, giroscopio y magnetómetro de las observaciones del campo terrestre.

Con respecto a esto, siguiendo la explicación de (Valenti, R. G. *et al*., 2015, pp. 19308-19314), tomamos esta notación:

- Para referirnos al sensor utilizamos el superíndice L (de local), mientras que, para referirnos a la tierra, utilizamos el superíndice G (de global).
- Para el acelerómetro, la aceleración medida se denota como $\boldsymbol{a}^{\mathrm{L}}$, y la aceleración gravitacional como $\boldsymbol{g}^{\mathrm{G}}$ y se definen como:

$$\boldsymbol{a}^{\mathrm{L}} = \begin{bmatrix} a_x & a_y & a_z \end{bmatrix}^T, \quad \|\boldsymbol{a}\| = 1$$

$$\boldsymbol{g}^{\mathrm{G}} = [0 \;\; 0 \;\; 1]^T$$

- Para el giroscopio, la medida de la velocidad angular $\boldsymbol{w}^{\mathrm{L}}$ será:

$$\boldsymbol{w}^{\mathrm{L}} = \begin{bmatrix} w_x & w_y & w_z \end{bmatrix}^T$$

- Finalmente, la medida del campo magnético por el magnetómetro será $\boldsymbol{m}^{\mathrm{L}}$, mientras que el verdadero campo magnético es $\boldsymbol{h}^{\mathrm{G}}$:

$$\boldsymbol{m}^{\mathrm{L}} = \begin{bmatrix} m_x & m_y & m_z \end{bmatrix}^T, \;\; \|\boldsymbol{m}\| = 1$$

$$\boldsymbol{h}^{\mathrm{G}} = \begin{bmatrix} h_x & h_y & h_z \end{bmatrix}^T, \;\; \|\boldsymbol{h}\| = 1$$

Además, hay que tener en cuenta que la ecuación de rotación de un vector entre dos marcos A y B, mediante un cuaternión, es:

$$\boldsymbol{v}^{\mathrm{B}} = \mathrm{R}(\boldsymbol{q}_{\mathrm{A}}^{\mathrm{B}})\boldsymbol{v}^{\mathrm{A}}$$

Con todo esto, en (Valenti, R. G. *et al*., 2015, pp. 19308-19314), se presenta la derivación algebraica del cuaternión de orientación $\boldsymbol{q}_{\mathrm{G}}^{\mathrm{L}}$, del marco global (G, la tierra) con respecto al marco local (L, el sensor), como una función de $\boldsymbol{a}^{\mathrm{L}}$ y $\boldsymbol{m}^{\mathrm{L}}$, generando el siguiente sistema de ecuaciones a partir de las observaciones de cada sensor y haciendo uso de la ecuación de rotación expuesta anteriormente:

$$\begin{cases} \mathrm{R}^T(\boldsymbol{q}_{\mathrm{G}}^{\mathrm{L}})\boldsymbol{a}^{\mathrm{L}} = \boldsymbol{g}^{\mathrm{G}} \\ \mathrm{R}^T(\boldsymbol{q}_{\mathrm{G}}^{\mathrm{L}})\boldsymbol{m}^{\mathrm{L}} = \boldsymbol{h}^{\mathrm{G}} \end{cases}$$

Una solución algebraica posible para hallar $\boldsymbol{q}_{\mathrm{G}}^{\mathrm{L}}$ es mediante su descomposición en dos cuaterniones auxiliares, referentes al acelerómetro $\boldsymbol{q}_{acc}$ y magnetómetro $\boldsymbol{q}_{mag}$:

$$\boldsymbol{q}_{\mathrm{G}}^{\mathrm{L}} = \boldsymbol{q}_{acc} \otimes \boldsymbol{q}_{mag}$$

Con:

$$\mathrm{R}(\boldsymbol{q}_{\mathrm{G}}^{\mathrm{L}}) = R(\boldsymbol{q}_{acc})R(\boldsymbol{q}_{mag})$$

Cuaternión del acelerómetro

El cuaternión auxiliar del acelerómetro ($\boldsymbol{q}_{acc}$) (Valenti, R. G. *et al*., 2015, pp. 19308-19314) realiza la transformación entre las dos observaciones del vector de gravedad en los dos marcos de referencia. De esta forma:

$$\mathrm{R}(\boldsymbol{q}_{\mathrm{G}}^{\mathrm{L}})\, \boldsymbol{g}^{\mathrm{G}} = \boldsymbol{a}^{\mathrm{L}}$$

Lo que lleva a que, utilizando las dos últimas ecuaciones vistas:

$$R(\boldsymbol{q}_{acc})R(\boldsymbol{q}_{mag})\begin{bmatrix}0\\0\\1\end{bmatrix}=\begin{bmatrix}a_x\\a_y\\a_z\end{bmatrix}$$

El desarrollo de esta ecuación, en el que no se va a entrar, genera un sistema totalmente determinado, y, tras unas consideraciones, se llega a que se tienen dos soluciones reales:

$$\boldsymbol{q}_{acc}=\begin{cases}\begin{bmatrix}\sqrt{\dfrac{a_z+1}{2}} & -\dfrac{a_y}{\sqrt{2(a_z+1)}} & \dfrac{a_x}{\sqrt{2(a_z+1)}} & 0\end{bmatrix}^T, a_z\geq 0\\ \begin{bmatrix}-\dfrac{a_y}{\sqrt{2(1-a_z)}} & \sqrt{\dfrac{1-a_z}{2}} & 0 & \dfrac{a_x}{\sqrt{2(1-a_z)}}\end{bmatrix}^T, a_z<0\end{cases}$$

Cuaternión del magnetómetro

En esta subsección se muestra el cálculo del cuaternión auxiliar del magnetómetro ($\boldsymbol{q}_{mag}$) (Valenti, R. G. *et al*., 2015, pp. 19308-19314). Primero se utiliza el cuaternión $\boldsymbol{q}_{acc}$ para rotar $\boldsymbol{m}^{\mathrm{L}}$ en un marco intermedio cuyo eje z sea el mismo que el del marco global (G). De esta forma:

$$\mathrm{R}^T(\boldsymbol{q}_{acc})^{\mathrm{L}}\boldsymbol{m}=\boldsymbol{l}$$

Lo que lleva a que:

$$\mathrm{R}^T(\boldsymbol{q}_{mag})\begin{bmatrix}l_x\\l_y\\l_z\end{bmatrix}=\begin{bmatrix}{l_x}^2+{l_y}^2\\0\\l_z\end{bmatrix}$$

Posteriormente, tras resolver el sistema de ecuaciones y, al igual que antes, hacer algunas consideraciones en las que no se van a entrar, se obtiene el resultado de

El desarrollo de esta ecuación, en el que no se va a entrar, genera un sistema totalmente determinado, y, tras unas consideraciones, se llega a que se tienen dos soluciones reales:

$$\boldsymbol{q}_{mag}=\begin{cases}\begin{bmatrix}\dfrac{\sqrt{\Gamma+l_x\sqrt{\Gamma}}}{\sqrt{2\Gamma}} & 0 & 0 & \dfrac{l_y}{\sqrt{2}\sqrt{\Gamma+l_x\sqrt{\Gamma}}}\end{bmatrix}^T, l_x\geq 0\\ \begin{bmatrix}\dfrac{l_y}{\sqrt{2}\sqrt{\Gamma-l_x\sqrt{\Gamma}}} & 0 & 0 & \dfrac{\sqrt{\Gamma-l_x\sqrt{\Gamma}}}{\sqrt{2\Gamma}}\end{bmatrix}^T, l_x<0\end{cases}$$

Con $\Gamma={l_x}^2+{l_y}^2$

Finalmente, el cuaternión de orientación del marco global (G, la tierra) con respecto al marco local (L, el sensor) quedará, como se ha indicado antes:

$$\boldsymbol{q}_{\mathrm{G}}^{\mathrm{L}} = \boldsymbol{q}_{acc} \otimes \boldsymbol{q}_{mag}$$

2.3 Búsqueda de datasets

Una vez conocidas los tipos de datos que se van a manejar, se puede hablar ya de los datasets utilizados.

Antes de comenzar el diseño y programación de una red neuronal, es necesario la obtención de los datos. Tras una revisión bibliográfica enfocada al reconocimiento de actividades humanas, se pudieron encontrar una serie de bases de datos o datasets, ampliamente utilizados en los trabajos de este problema y que tienen suficientes datos e información para ser la base de un sistema de Deep Learning.

En esta revisión bibliográfica, se buscó un dataset que se asemejara lo máximo posible al objetivo de este Trabajo Fin de Grado, es decir, que contuviera muchas actividades distintas, además de que esas actividades se asemejaran a las que se tendrían que aprender para su utilización en un juego serio del grupo GTI y que proporcionara los datos de forma semejante a como los proporcionan los sensores del grupo GTI, es decir, que además de los datos del acelerómetro, giroscopio y magnetómetro, estuvieran en ese dataset los cuaterniones calculados.

En este sentido, se encontró un dataset que cumplía todos los requisitos, el dataset REALDISP (Baños, O. *et al*., 2012a) y es el que se ha utilizado más a fondo. Sin embargo, también se presenta otro dataset que se ha utilizado como introducción al reconocimiento de movimientos con Deep Learning.

A continuación, se muestran y explican los dos datasets principales con los que se ha trabajado en este Trabajo Fin de Grado.

2.3.1 Dataset para Reconocimiento de Movimientos usando Smartphones

Este dataset es un dataset público (Anguita, D. *et al.*, 2013), cuyos datos están recogidos a partir de los sensores de los smartphones.

En este dataset se distinguen entre 15 actividades (7 estáticas y 8 dinámicas), las cuales se muestran a continuación:

No.	Static	Time (sec)	No.	Dynamic	Time (sec)
0	Start (Standing Pos)	0	7	Walk (1)	15
1	Stand (1)	15	8	Walk (2)	15
2	Sit (1)	15	9	Walk Downstairs (1)	12
3	Stand (2)	15	10	Walk Upstairs (2)	12
4	Lay Down (1)	15	11	Walk Downstairs (1)	12
5	Sit (2)	15	12	Walk Upstairs (2)	12
6	Lay Down (2)	15	13	Walk Downstairs (3)	12
			14	Walk Upstairs (3)	12
			15	Stop	0
				Total	**192**

Figura 6. Actividades en el dataset para reconocimiento de movimientos utilizando smartphones.

La metodología de obtención de este dataset es la siguiente:

Se seleccionó a un grupo de voluntarios de entre 19 y 48 años. A cada persona se le indicó que siguiera una serie de actividades mientras llevaban el smartphone Samsung Galaxy S II. Cada sujeto realizó la serie de actividades de la Figura 6 dos veces: primero con el smartphone en el lado izquierdo y luego con él donde el sujeto prefiriera. Las pruebas fueron realizadas en condiciones de laboratorio, pero se les pidió a los voluntarios que realizaran las actividades de forma libre para obtener un dataset más natural (Anguita, D. *et al.*, 2013).

Los datos de este dataset son medidas triaxiales, es decir, en los ejes X, Y y Z, de los acelerómetros y de los giroscopios de los móviles, con lo que se obtienen tanto aceleraciones como velocidades angulares, con una tasa de muestreo de 50 Hz:

Name	Time	Freq.
Body Acc	1	1
Gravity Acc	1	0
Body Acc Jerk	1	1
Body Angular Speed	1	1
Body Angular Acc	1	0
Body Acc Magnitude	1	1
Gravity Acc Mag	1	0
Body Acc Jerk Mag	1	1
Body Angular Speed Mag	1	1
Body Angular Acc Mag	1	1

Figura 7. Features en el dataset para reconocimiento de movimientos utilizando smartphones.

Este dataset se ha utilizado en este Trabajo Fin de Grado para realizar una prueba previa al mismo, como introducción al Reconocimiento de Actividades Humanas, debido a que los datos, aunque más simples, son similares a lo que buscábamos y sirve como una pequeña introducción y primera toma de contacto con el problema HAR.

2.3.2 REALDISP

El dataset REALDISP es un dataset abierto (Baños, O. *et al*., 2012a), disponible en un repositorio de datasets (UCI Machine Learning Repository, n.d) con el que se puede investigar los efectos de la mala colocación de los sensores en el reconocimiento de actividades (AR, Activity Recognition).

En este dataset se consideran 33 actividades fitness diferentes, grabadas usando 9 sensores inerciales (del tipo Xsens) colocados en 17 participantes.

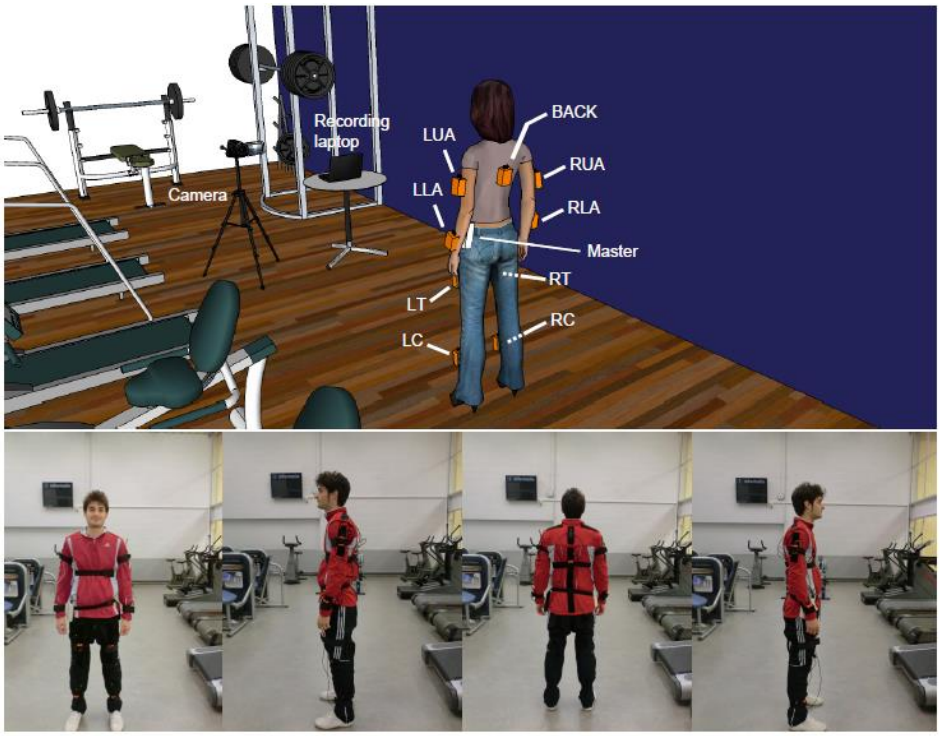


Figura 8. Disposición de los sensores en el dataset REALDISP.

El set de actividades utilizado es el siguiente:

Activity set		
L1: Walking (1 min)	L12: Waist rotation (20x)	L23: Shoulders high amplitude rotation (20x)
L2: Jogging (1 min)	L13: Waist bends (reach foot with opposite hand) (20x)	L24: Shoulders low amplitude rotation (20x)
L3: Running (1 min)	L14: Reach heels backwards (20x)	L25: Arms inner rotation (20x)
L4: Jump up (20x)	L15: Lateral bend (10x to the left + 10x to the right)	L26: Knees (alternatively) to the breast (20x)
L5: Jump front & back (20x)	L16: Lateral bend arm up (10x to the left + 10x to the right)	L27: Heels (alternatively) to the backside (20x)
L6: Jump sideways (20x)	L17: Repetitive forward stretching (20x)	L28: Knees bending (crouching) (20x)
L7: Jump leg/arms open/closed (20x)	L18: Upper trunk and lower body opposite twist (20x)	L29: Knees (alternatively) bend forward (20x)
L8: Jump rope (20x)	L19: Arms lateral elevation (20x)	L30: Rotation on the knees (20x)
L9: Trunk twist (arms outstretched) (20x)	L20: Arms frontal elevation (20x)	L31: Rowing (1 min)
L10: Trunk twist (elbows bended) (20x)	L21: Frontal hand claps (20x)	L32: Elliptic bike (1 min)
L11: Waist bends forward (20x)	L22: Arms frontal crossing (20x)	L33: Cycling (1 min)

Figura 9. Actividades en el dataset REALDISP.

En este Trabajo Fin de Grado se ha decidido añadir otra actividad, la actividad L0, correspondiente al periodo de inactividad, siendo: "L0: No actividad", llevando a aumentar este set de actividades hasta las 34.

En el dataset se diferencia entre:

- Datos de sensores colocados idealmente, es decir, por un profesional (*"ideal"*). En (Baños, O. *et al*., 2014a) se indica que estos datos pueden considerarse como el set de entrenamiento para sistemas de reconocimiento de actividades.
- Datos de sensores colocados por el propio sujeto, es decir, sin la supervisión de un profesional (*"self"*), con lo que estarán descolocados con respecto a la posición ideal. Este escenario intenta simular la variabilidad que puede ocurrir en el uso del día a día en un sistema de reconocimiento de actividades. En este caso hay 3 sensores descolocados. (Baños, O. *et al*., 2014a)
- Datos de sensores descolocados tanto por el sujeto como por el instructor intencionadamente (*"mutual"*) en cuanto a rotaciones y traslaciones con respecto a la colocación ideal. En este caso hay hasta 7 sensores descolocados (Baños, O. *et al*., 2014a).

En cuanto a los sensores, cada nodo (sensor) proporciona medidas de aceleración, velocidad angular y campo magnético en las 3 dimensiones además de una estimación de la orientación en 4D. Así que cada sensor da 13 valores a la salida en cada medida, lo que lleva a un set de 117 señales, recogidas a 50 Hz, es decir, cada 0,02 segundos.

De esta forma el dataset consta de 46 ficheros *.log* con los datos antes indicados, es decir, en el fichero "subjectX_ideal.log" sustituyendo X por el número del sujeto correspondiente, aparecerán los datos referentes al sujeto X con los sensores colocados por el profesional, en el fichero "subjectX_self.log" aparecerán los datos referentes al sujeto X con algunos sensores colocados por él mismo y en el fichero "subjectX_mutualY.log" sustituyendo la Y por el número 4,5,6 ó 7 (lo que indica el número de sensores descolocados o mal colocados) aparecerán los datos referentes al sujeto X con algunos sensores mal colocados tanto por él mismo como por el profesional.

Hay que indicar que, de estos 46 ficheros, tras una inspección de cada uno de ellos, y como se indica en (Baños, O. *et al*., 2014a), hay varios ficheros corruptos, en concreto 3: "subject6_self.log", "subject13_self.log" y "subject15_mutual4.log", con lo que el dataset se queda con 43 ficheros de datos.

La forma en la que están distribuidos los datos dentro de cada fichero es en forma de matriz, más en concreto de la siguiente forma:

- La 1ª y la 2ª columna son los timestamps (tiempos de captura de los sensores) con la parte entera en segundos (en la 1ª columna) y el resto en microsegundos (en la 2ª columna).
- De 3ª columna a 119ª son las medidas del acelerómetro (ACC), giroscopio (GYR), campo magnético (MAG) y cuaterniones (QUAT) como se ha indicado anteriormente. En este orden están: ACC (X, Y, Z), GYR (X, Y, Z), MAG (X, Y, Z) y QUAT (Re, i, j, k), estas 13 columnas para cada uno de los sensores, los cuales van en el siguiente orden: RLA, RUA, BACK, LUA, LLA, RC, RT, LT, LC.
- La 120ª columna son las etiquetas de actividad.

Se ha elegido este dataset para la realización de este Trabajo de Fin de Grado ya que las características de este se adecúan a lo que se buscaba: tiene datos de movimientos fitness de la persona y una de las formas en que dan los datos los sensores son en cuaterniones, al igual que los sensores de los que se dispone en el Laboratorio de GTI.

Capítulo 3 Deep Learning

El Deep Learning es la técnica a seguir en este Trabajo Fin de Grado. Durante el desarrollo de este capítulo se desglosarán las distintas opciones de diseño, mostrando sus fundamentos teóricos, analizados previamente al comienzo de la toma de decisiones y el inicio del estudio de las redes.

3.1 Introducción

Este Trabajo de Fin de Grado de Reconocimiento de actividades físicas con sensores inerciales y Redes Neuronales de Aprendizaje Profundo no podría entenderse sin una explicación sobre la Inteligencia Artificial (AI, *Artificial Intelligence*), el Machine Learning y el Deep Learning.

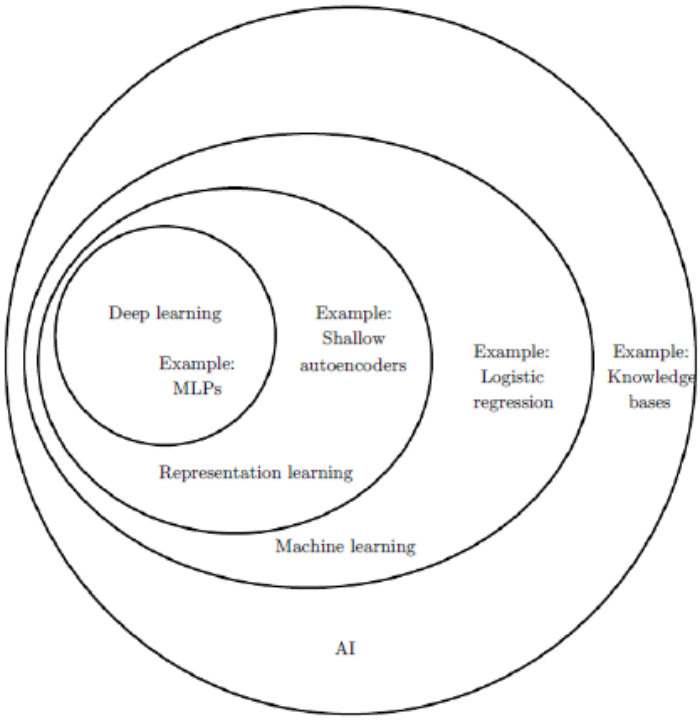


Figura 10. Representación en de la Inteligencia Artificial.

En la Figura 10, de (Goodfellow, I. *et al*., 2016), se representa un diagrama de Venn, en el que se puede ver cómo el Deep Learning es un subconjunto del Machine Learning, y,

por lo tanto, de la Inteligencia Artificial. Por ello, es necesario hablar de la Inteligencia Artificial para después hacerlo del Deep Learning.

La Inteligencia Artificial es aquel conjunto de técnicas a través del cual se realiza software que aprende de modelos matemáticos, sin necesidad de preprogramar reglas que abarquen todas las infinitas combinaciones de posibilidades dentro de un problema.

En los comienzos de la Inteligencia Artificial, se atacaron y solventaron problemas rápidamente que eran intelectualmente difíciles para los humanos, pero relativamente sencillos para los ordenadores, problemas que pueden describirse mediante una serie de reglas matemáticas. El verdadero reto de la Inteligencia Artificial es resolver tareas que son fáciles de realizar para las personas, pero difíciles de describir formalmente por ellas, problemas que resolvemos de forma intuitiva, como reconocer ciertas palabras cuando las escuchamos o caras e imágenes cuando las vemos (Goodfellow, I. *et al*., 2016).

Hoy en día, la Inteligencia Artificial es un campo próspero con muchas aplicaciones prácticas y temas de investigación activos. El software inteligente se busca para automatizar labores rutinarias, entender el lenguaje o imágenes, hacer diagnósticos en medicina y apoyar la investigación científica, algo solamente posible si, como se ha indicado antes, se consiguen abarcar las infinitas posibilidades dentro de todas estas tareas (Goodfellow, I. *et al*., 2016).

Esta cantidad de posibilidades nos llevan a la necesidad de que las máquinas realicen un aprendizaje automático, es decir, que sean capaces de extraer patrones de los datos que les proporcionamos. Este aprendizaje automático es lo que se define como Machine Learning. Por este motivo se utiliza software de este estilo para que la máquina, por ejemplo, automatice labores, entienda lo que dice una persona o lo que se muestra en una imagen, haga diagnósticos médicos y sirva de apoyo en la investigación científica, entre otras muchas aplicaciones (Yoshua Bengio *et al*., 2016, citado en Goodfellow, I. *et al*., 2016).

El Machine Learning hace uso de un conjunto de técnicas y algoritmos que le permiten conseguir sus objetivos, y una de estas técnicas son las redes neuronales. Las redes neuronales son sistemas artificiales formados por "neuronas" (esto lo veremos en la sección 3.3) que están inspiradas en el funcionamiento bilógico del cerebro humano, compuesto por la interconexión entre las neuronas. Las redes neuronales están compuestas por distintas capas, conexiones y una dirección en la que se propagan los datos atravesando cada capa, realizando cada una de ellas una función concreta sobre los datos.

Por esta razón se requiere de una gran cantidad de datos para proporcionarlos a las neuronas de la red y así ser capaz de reconocer patrones, hacer clasificaciones de los datos y categorizar resultados con más precisión que si se usa una cantidad escasa de datos.

Del desarrollo de estas redes neuronales, surgió el Deep Learning:

El Deep Learning es una rama del Machine Learning que utiliza una gran cantidad de datos y que permite al ordenador tener una percepción de las cosas lo más cercana hasta ahora a la de los seres humanos, proporcionando una alta adaptabilidad y una estructura y lenguaje comunes para describir los problemas a resolver (Udacity, 2018).

Es una solución que permite a los ordenadores aprender de la experiencia y entender el mundo en términos de una jerarquía de conceptos, definiendo cada uno de esos conceptos a partir de sus relaciones con otros más simples. Mediante el aprendizaje a través de la experiencia, esta aproximación evita la necesidad de especificar de forma precisa, por parte de los humanos, todo el conocimiento que el ordenador necesita. La jerarquía de conceptos permite al ordenador aprender conceptos complicados construyéndolos a partir de conceptos más sencillos (Goodfellow, I. *et al*., 2016).

El Deep Learning se ha ido desarrollando y convirtiendo en la técnica líder del Machine Learning en varios ámbitos, pero sobre todo destaca en los campos de *Computer Vision* (aquel en el que se pretende reconocer objetos, animales o personas de una imagen o vídeo) y *Speech Recognition* (aquel en el que se pretende reconocer lo que una persona está diciendo a partir de la grabación de su voz) , y esto ha sido, en gran parte, gracias al desarrollo y a la utilización de las GPUs, ya que permiten un procesamiento mucho más rápido de la gran cantidad de datos que se necesitan para que la red arroje un resultado preciso, llevando, en unos pocos años, a reducir en gran medida los tiempos de procesamiento (LeCun *et al*., 2015, citado en Goodfellow, I. *et al*., 2016).

Como se indica en (Goodfellow, I. *et al*., 2016), si dibujáramos un gráfico mostrando cómo estos conceptos se construyen encima de otros más sencillos, el gráfico sería profundo, con muchas capas. Por esto se llama a esta aproximación de Inteligencia Artificial, Deep Learning.

En este Trabajo de Fin de grado se va a hacer uso del Deep Learning, como el título indica y como se ha indicado en secciones anteriores, para el reconocimiento de actividades físicas o movimientos, es decir, para el Reconocimiento de Actividad Humana (HAR):

Reconocer actividades humanas y el contexto en el ocurren a partir de los datos procedentes de un sensor es el núcleo de las tecnologías inteligentes asistivas (Ordóñez, F. J. y Roggen, D., 2016).

El Reconocimiento de Actividad Humana es un reto debido a la gran variabilidad de motores de movimiento empleados para una determinada acción. Está basado en la asunción de que los movimientos del cuerpo se traducen en unos patrones característicos de las señales de los sensores, que pueden ser detectados por los mismos y clasificados utilizando técnicas de Machine Learning, en concreto el Deep Learning (Ordóñez, F. J. y Roggen, D., 2016).

En este Trabajo de Fin de Grado, por los motivos indicados en la sección 1.1, nos hemos centrado en la utilización de sensores llevados sobre el cuerpo, lo cual nos lleva a un reconocimiento basado en la combinación de varios sensores, sobre cuyas señales aplicaremos técnicas de Deep Learning, ya que parece cumplir los requisitos de este tipo de reconocimiento: el rendimiento se mejora sobre las técnicas existentes de reconocimiento y puede tener potencial para descubrir características que están ligadas a la dinámica del movimiento humano, desde la codificación de movimientos simples en las capas bajas hasta movimientos más complejos en capas superiores, lo cual puede ser útil para escalar el Reconocimiento de Actividad Humana a actividades más complejas (Ordóñez, F. J. y Roggen, D., 2016).

3.2 Historia del Deep Learning

En esta sección se comenta la historia del Deep Learning, su evolución y algunos hitos de carácter general conseguidos hasta la actualidad:

La historia del Deep Learning comienza con la historia de la Inteligencia Artifical, que puede remontarse a 1943, cuando Walter Pitts y Warren McCulloch crearon un modelo de ordenador basado en las redes neuronales del cerebro humano. Utilizaron una combinación de algoritmos y matemáticas que para imitar el proceso de pensar (McCulloch, W., S. y Pitts, W., 1943).

Posteriormente, en 1957, Frank Rosenblatt, comenzó el desarrollo del **Perceptron**, la primera red neuronal que creó gran expectación, pero que en realidad era muy limitada (Rosenblatt, F., 1958). En los siguientes años, se realizaron más redes neuronales basándose en el Perceptron, probando otras reglas de aprendizaje.

Las limitaciones del Perceptron fueron expuestas por Marvin Minsky y Seymour Papert, al demostrar que esta red neuronal no era capaz de aprender a partir de una función no lineal, un tipo de función muy utilizada (Atich, D. J., 2001, pp. 6-7).

La era moderna de las redes neuronales artificiales surge cuando se desarrollaron las bases de un algoritmo de aprendizaje llamado ***backpropagation*** (propagación hacia atrás), en un intento de Rosenblatt de generalizar el algoritmo de aprendizaje del Perceptron a múltiples capas en 1962. En realidad, hubo muchos intentos de generalizar este algoritmo a múltiples capas entre los años 60 y los años 70, como por ejemplo el de Seppo Linnainmaa, en 1970, que escribió su tesis incluyendo un código FORTRAN para la *backpropagation*, pero en una versión básica. La versión moderna y que tuvo éxito de la *backpropagation*, parece tener tres inventores distintos: Paul Werbos desarrolló la idea básica en 1974 en el título *"Beyond Regression"*, mientras que David Parker y David Rumelhart la desarrollaron a la vez, en 1982. Sin embargo, a partir de la publicación del *paper" Parallel distributed processing: Explorations in the microstructure of cognition"* por Rumelhart, Hinton y Williams en 1986, mostrando sus aplicaciones en las redes neuronales y la Inteligencia Artificial, un gran número de investigadores se interesaon por este algoritmo (Chauvin, Y. y Rumelhart, D. E., 1995, pp 1-2).

Los primeros esfuerzos de desarrollo de algoritmos de Deep Learning vinieron de Alexey Grigoryevich Ivakhnenko (desarrolló el *Group Method of Data Handling* (Ivakhnenko, A. G. y Ivakhnenko, G. A., 1995)) y junto a Valentin Grigorevich Lapa (autor de *Cybernetics and Forecasting Techniques*) en 1967.

En los años 80 se realizaron varios trabajos muy importantes para las redes neuronales (Udacity, 2018). Las primeras redes neuronales convolucionales fueron usadas por Kunihiko Fukushima. En 1979 se comenzó a desarrollar una red neuronal artificial, llamada **Neocognitron** (Fukushima, K. y Miyake, S., 1982), la cual utilizaba un diseño multi-capa jerárquico, que permitía al ordenador "aprender" a reconocer patrones visuales.

En 1989, Yann LeCun proporcionó la primera demostración práctica de la *backpropagation* en los Laboratorios Bell (LeCun, Y. *et al*., 1989). Más tarde, crearía la

red LeNet 5, en la que combinó redes neuronales convolucionales con la *backpropagation* para clasificar dígitos escritos a mano (el conocido MNIST).

En 1995, Corinna Cortes y Vladimir Vapnik desarrollaron el ***Support Vector Machine*** (SVM, un sistema para mapear y reconocer datos similares) (Cortes, C. y Vapnik, V., 1995). Las celdas LSTM (*Long short-term memory*) para las redes neuronales recurrentes fue desarrollada en 1997, por Sepp Hochreiter y Juergen Scmidhuber (Hochreiter, S. y Schmidhuber, J., 1997).

Sin embargo, "en los años 90, los ordenadores eran muy lentos, y los datasets eran muy pequeños, además de que la investigación no conseguía encontrar demasiadas aplicaciones en el mundo real" (Udacity, 2018).

Es por esta razón, que, a comienzos de los 2000, las redes neuronales no eran una opción viable ni interesante para el mundo del Machine Learning (Udacity, 2018).

Sin embargo, poco después del comienzo del siglo XXI, ocurrió el siguiente paso revolucionario para el Deep Learning, cuando se comenzó a tener más datos, los ordenadores empezaron a ser más rápidos en el procesamiento de datos y las GPUs se desarrollaron para comenzar a procesar datos no relacionados con los gráficos (Buck, I., 2010). Gracias a estos avances, las redes neuronales compitieron con el *Support Vector Machine*. Mientras una red neuronal podía ser lenta comparada con un SVM, las redes neuronales ofrecían mejores resultados usando los mismos datos, además de ser capaces de mejorar cuantos más datos se utilicen.

Ha sido en los últimos años en los que se han tenido más avances en las redes neuronales. En 2009, las redes neuronales irrumpieron con fuerza en el campo de la *Speech Recognition*, un poco más tarde, en 2012, lo hicieron en el campo de la *Computer Vision*, y ya en 2014, lo hicieron en el campo de *Machine Translation*, convirtiéndose, en los tres casos, en la técnica líder para la resolución de esos problemas (Udacity, 2018).

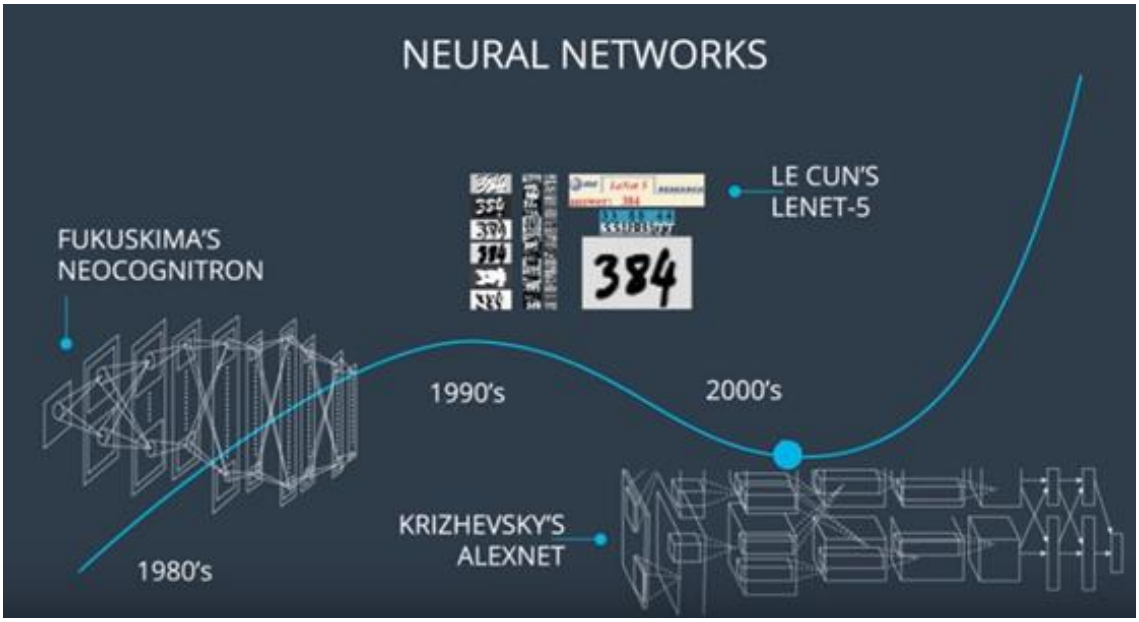


Figura 11. Línea temporal del Deep Learning.

Como se puede ver en la Figura 11 de (Udacity, 2018), destacan como eventos importantes en la historia más actual de las redes neuronales: el **Neocognitron** de Fukushima, la **LeNet-5** de LeCun y la **AlexNet** de Krizhevshy (Krizhesky, A., *et al*., 2012), que ganó varias competiciones con una red convolucional utilizando múltiples GPUs.

Actualmente, el procesamiento de Big Data y la evolución de la Inteligencia Artificial dependen los dos del Deep Learning, que no para de evolucionar (Udacity, 2018).

3.3 Teoría

En este apartado vamos a ver la teoría relacionada con lo que se ha trabajado en este Trabajo Fin de Grado. Iremos desde los modelos más sencillos, tratando de explicar los conceptos más básicos, hasta los más complejos, que son los utilizados y la aplicación de los conceptos vistos a lo largo de esta sección:

3.3.1 Estudio con modelos lineales (Conceptos básicos)

En primer lugar, previo a cualquier aprendizaje por parte de un algoritmo de Deep Learning, o para ser más general, de Machine Learning, nos encontramos con la ***feature extraction*** o extracción de características. El objetivo de este proceso es la extracción o construcción de las características importantes de los datos, es decir, aquellas que tenga un "significado", que permitan un mejor entrenamiento, aprendizaje y entendimiento.

Una vez se tienen las características significativas de los datos, se distinguen dos tipos de aprendizaje, que desembocan en dos formas de entrenamiento:

- Supervisado (***Supervised Learning***): En este tipo de aprendizaje, se indica a la máquina, durante el entrenamiento, qué es cada cosa, mediante el uso de etiquetas (***labels***).
- No supervisado (***Unsupervised Learning***): En este tipo de aprendizaje no se utilizan las etiquetas. Se le proporcionan los datos a la máquina y debe sacar conclusiones sin ayuda humana.

En nuestro caso utilizaremos un aprendizaje supervisado.

El objetivo del aprendizaje es que la máquina sea capaz de clasificar, y a partir de esta clasificación poder realizar otra serie de actividades, como puede ser la detección (por ejemplo, en un semáforo, detectar si hay peatones o no cruzando un paso de cebra a través de un clasificador binario) o el "ranking" (ser capaz de crear una lista de preferencias, por ejemplo, de páginas web, mediante la clasificación del par petición-página web).

El clasificador más sencillo que puede haber es un **clasificador lineal**, en el que, a través del ajuste de unos pesos y unos sesgos, la máquina es capaz de determinar qué son o qué significan los datos que le estamos proporcionando. Este clasificador sigue una función lineal tan sencilla como:

$$WX + b = Y$$

En esta función *W* es la matriz de pesos, *b* son los sesgos (ambos son los que se entrenan para después realizar una predicción correcta) y *X* son los datos de entrada. Esta función se puede ver como un multiplicador matricial (Udacity, 2018).

Esta función genera una salida *Y*, que será un vector de números (denominados ***scores***, es decir, resultados o ***logits***, en el ámbito de la regresión lineal), cada uno correspondiente a cada una de las posibilidades de resultado, y se deberá convertir a unas probabilidades entre 0 y 1 a través de la función conocida como ***softmax***:

$$S(y_i) = \frac{e^{y_i}}{\sum_j y_j}$$

De esta forma, se obtiene, para cada posible resultado, una probabilidad, es decir, como salida tenemos un vector de probabilidades tomándose como predicción aquella que tenga la probabilidad más alta y el objetivo es que esa salida se parezca lo más posible a lo que son realmente los datos (a la etiqueta) (Udacity, 2018).

Este vector de probabilidades se comparará con el vector de etiquetas, que estará expresado en el formato ***One-hot encoding***. Este formato trata básicamente en expresar las etiquetas o *labels*, las cuales corresponden a una de las clases, como un vector en el que todos los elementos son 0, excepto el correspondiente a la etiqueta correspondiente a la clase correcta, que pasa a ser 1.

Esta comparación entre los vectores se consigue realizar mediante la utilización de la entropía cruzada, que determina la distancia entre ambos:

$$D(S, L) = -\sum_i L_i \log(S_i)$$

Siendo *S* el vector de probabilidades (la salida de la función *softmax*) y *L* el vector de las etiquetas.

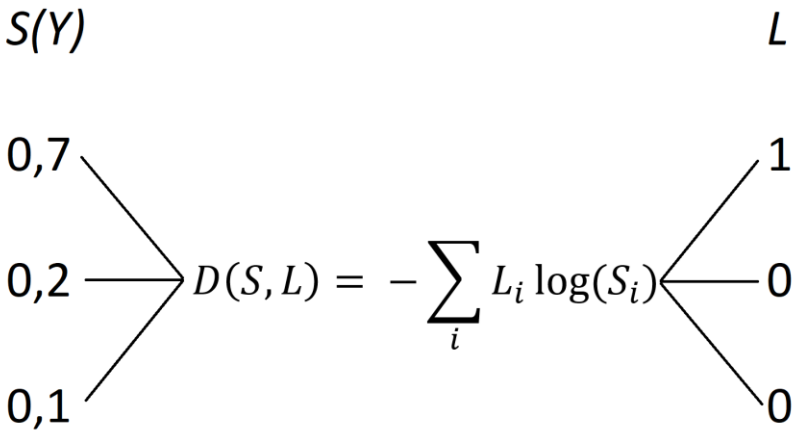


Figura 12. Ejemplo de One-hot Encoding.

Con lo que, el proceso de clasificación lineal constaría de:

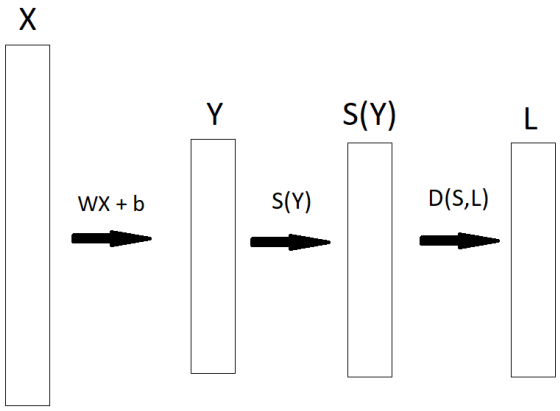


Figura 13. Modelo de clasificación lineal.

Tenemos una entrada *X*, que va a ser convertida en *logits Y* mediante el uso de un modelo lineal, formado por la matriz multiplicadora (pesos) y los sesgos. Después, pasamos los *logits* o *scores* por la función *softmax* para obtener las probabilidades (*S(Y)*), y después comparamos esas probabilidades con el vector de etiquetas (*L*) mediante la entropía cruzada (*D(S, L)*) (Udacity, 2018).

La intención es minimizar el error (también nos podemos referir a él como pérdida o coste) cometido en la predicción, es decir, requerimos de una **optimización** del error calculado. Este error, como se ha indicado, se calcula con la entropía cruzada, y en este proceso de optimización se busca que la distancia del vector de probabilidades sea

pequeña para la clase correcta y muy grande para la clase incorrecta. De esta forma tenemos:

$$\mathcal{E} = \frac{1}{N}\sum_{i} D(S(wx_i + b), L_i)$$

Es decir, el error es el promedio de la entropía cruzada de todo el entrenamiento. Por lo tanto, se deben encontrar los pesos que minimicen esa función de error, convirtiéndose en un problema de optimización. Y la forma más sencilla de realizarlo es mediante la derivada cambiada de signo, tomando esa dirección y sentido hasta alcanzar el mínimo. Así que, para cada iteración en el entrenamiento, los pesos y sesgos se actualizan como sigue:

$$w \leftarrow w - \alpha \Delta w \mathcal{E}$$

$$b \leftarrow b - \alpha \Delta b \mathcal{E}$$

$$\alpha = \boldsymbol{learning\ rate}\ (tasa\ de\ aprendizaje)$$

Este procedimiento de medida y optimización del error se denomina ***Gradient Descent***, sin embargo, tiene una gran dificultad en el escalado, es decir, cuantos más parámetros se tengan que ajustar en el modelo, mayor será el número de operaciones a realizar, llevándolo a no ser eficiente del todo, y es por ello por lo que, el método más conocido y que supera este problema es el ***Stochastic Gradient Descent*** (Udacity, 2018).

Como su propio nombre indica es un *Gradient Descent* estocástico, en el que, en vez de calcular el error con todos los datos, se estima a partir de unos pocos, tomando dicha estimación como buena, llevando a realizar numerosos, pero pequeños y sencillos pasos en el descenso del gradiente en vez de realizar pocos que nos lleven a tener que realizar operaciones muy costosas (Udacity, 2018).

Sin embargo, este no es el método de optimización que vamos a utilizar, si no que utilizaremos el conocido como ***Adam*** (*Adaptive moment estimation*).

Este algoritmo es una extensión del *Stochastic Gradient Descent* en la que, a diferencia de éste, no mantiene una única *learning rate* para todos los pesos, sino que mantiene una *learning rate* diferente para cada parámetro de la red, adaptándolos de manera individual mientras se va desarrollando el aprendizaje de la red (Kingma, D. P. y Lei Ba, M., 2015).

Específicamente, "el algoritmo calcula un movimiento promedio exponencial del gradiente y de su cuadrado, con los hiperparámetros β_1 y β_2 controlando las tasas de decaimiento de estos promedios de movimiento" (Kingma, D. P. y Lei Ba, M., 2015).

Este algoritmo combina las ventajas de otras dos extensiones del *Stochastic Gradient Descent*, como son:

- *Adaptive Gradient Algorithm* (AdaGrad): este algoritmo funciona bien para gradientes dispersos (Duchi, J. *et al*., 2011, citado en Kingma, D. P. y Lei Ba, M., 2015). Este algoritmo corresponde con un algoritmo Adam de parámetros $\beta_1 = 0$, β_2 infinitesimal y otra *learning rate* α modificada (Kingma, D. P. y Lei Ba, M., 2015).

- *Root Mean Square Propagation* (RMSProp): en este caso, el algoritmo realiza las actualizaciones de sus parámetros usando un momento del gradiente, mientras que Adam utiliza los momentos de primer y segundo orden (Kingma, D. P. y Lei Ba, M., 2015).

Los valores iniciales de los promedios de movimiento son 0, llevando a las estimaciones sesgadas de los momentos a cero, especialmente durante los primeros instantes y cuando las tasas de decaimiento son pequeñas, es decir, los parámetros β_1 y β_2 son cercanos a 1). Sin embargo, esto se puede arreglar añadiendo un término ε (Kingma, D. P. y Lei Ba, M., 2015).

A continuación, vemos una comparativa de los costes de entrenamiento en una red multi-capa de este algoritmo con otros realizada por sus autores:

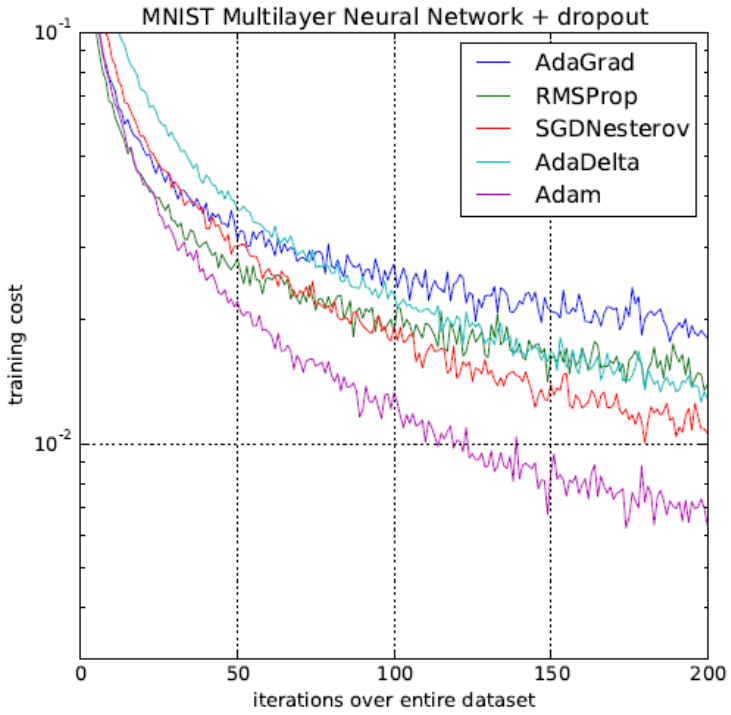


Figura 14. Comparativa del optimizador Adam con otros optimizadores conocidos.

Como se puede observar en la figura 14 (Kingma, D. P. y Lei Ba, M., 2015) el algoritmo *Adam* es el que da un coste más bajo durante el entrenamiento.

En este Trabajo Fin de Grado se utiliza el optimizador *Adam* implementado por Tensorflow, que utiliza los valores recomendados por el *paper* (*learning rate*= 0.001, $\beta_1 = 0.99$, $\beta_2 = 0.999$, $\varepsilon = 1^{-8}$), ya que cada vez se propone más como optimizador base de cualquier aplicación de Deep Learning.

Como comentario adicional, una técnica habitual cuando se diseñan redes neuronales es poner los valores iniciales de los pesos aleatorios, siguiendo una distribución con media cero y varianza pequeña (habitualmente 1), mientras que los sesgos se inicializan a cero, para posteriormente ir actualizándose como se ha indicado anteriormente. Esta inicialización facilita el proceso de la minimización del error (Udacity, 2018).

Otro factor a tener en cuenta es la cantidad de datos con la que se trabaja. El Deep Learning permite trabajar con una gran cantidad de datos, pero hay que controlar cómo trabajamos en el modelo con esos datos. En este sentido tenemos tres tipos de sets:

- **Set de entrenamiento**: Este set es el que más datos debe contener. Es el que se utilizará para que el modelo aprenda y generalice.
- **Set de validación**: Este set sirve para ir comprobando el rendimiento del entrenamiento. Es un set cuyos datos son mezclados con los del entrenamiento y cada cierto número de iteraciones de entrenamiento se evalúa con ellos el modelo conseguido (es decir, los pesos y sesgos del modelo).
- **Set de test**: Este set debe ser totalmente disjunto a los otros 2. Es el que da el valor de precisión real del modelo.

Si se trabaja mal con alguno de estos sets (o se utilizan mal los parámetros del modelo) puede ocurrir el fenómeno conocido como ***overfitting***, que consiste en que el modelo empleado ha aprendido exactamente los datos de entrenamiento, pero no es capaz de generalizar esos datos para predecir bien con otros. Esto se pone de manifiesto cuando la precisión de entrenamiento alcanza valores muy elevados, sin embargo, en validación y test la precisión es mucho más baja, además, los costes de validación van aumentando cada vez más (Udacity, 2018).

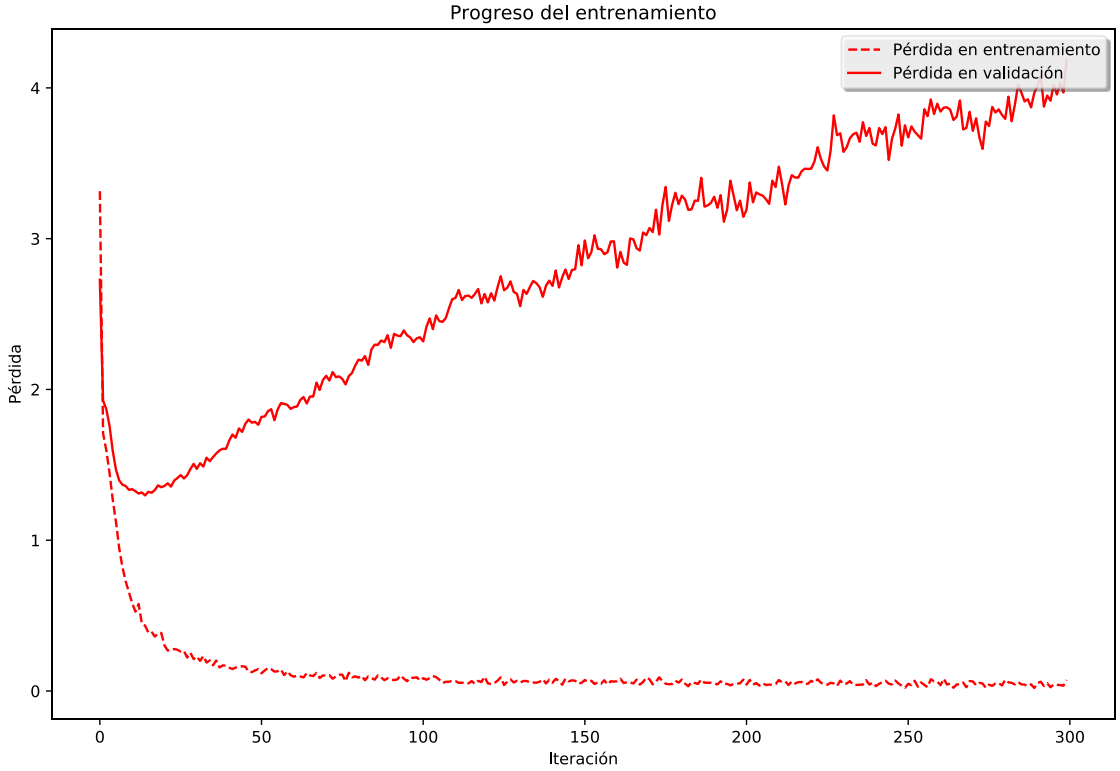


Figura 15. Ejemplo de un caso de overfitting.

Como se puede observar en la figura 15 la pérdida durante el entrenamiento disminuye mientras que la de validación cada vez es mayor.

3.3.2 Redes Neuronales Profundas (DNN)

Las redes neuronales básicas son las ***Feedforward Neural Networks***, también conocidas como Perceptron multi-capa (**MLP**, *multi-layer Perceptron*). Estas redes neuronales serían la extensión a las redes del modelo lineal visto en el apartado anterior, en el que cada **nodo** (agrupados en **capas**) de la red está asociado con el resto de los nodos mediante conexiones con **distintos pesos**. Los distintos nodos de la red transforman los datos mediante operaciones no lineales para crear un umbral de decisión para la entrada, proyectándolo en un espacio donde se convierte linealmente separable (Goodfellow, I. *et al*., 2016).

Se denominan *redes* porque normalmente se representan componiendo diferentes funciones. Siguen una estructura en cadena, cuya longitud determina la ***profundidad*** del modelo (de esta terminología viene Deep Learning):

$$f(\boldsymbol{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\boldsymbol{x})))$$

Como se ha indicado en el apartado anterior, la salida del modelo se espera que se parezca lo más posible a lo que son realmente los datos, es decir, los ejemplos del entrenamiento indican lo que la capa de salida de la red debe hacer, que es producir un valor cercano a la etiqueta a través de la cual indicamos lo que son los datos. El comportamiento del resto de las capas de la red no lo determinan los ejemplos de entrenamiento, sino que es el propio algoritmo de aprendizaje el que determina cómo utilizar esas capas para obtener el resultado deseado. Como no se muestra el resultado de las capas intermedias, estas capas se denominan capas escondidas (***hidden layers***) (Goodfellow, I. *et al*., 2016).

Estas redes se denominan *neuronales* porque están inspiradas por la neurociencia: cada capa escondida trabaja con vectores, la dimensión de estos vectores determina la *anchura* del modelo, pudiéndose interpretar que cada elemento del vector está realizando un papel similar al de una neurona (Goodfellow, I. *et al*., 2016).

Es por ello, que se puede ver cada capa como un conjunto de nodos denominado ***units*** que actúan en paralelo, pareciéndose cada una de estas *units* a una neurona, en el sentido de que reciben a la entrada datos de otras muchas *units* y calcula su propio **valor de activación**, es decir, en el caso de las *Feedforward Neural Networks*, primero se pasará la entrada por el modelo lineal, y al resultado, se le aplicará la **función de activación** (no lineal) de la *unit*. Este valor de activación dependerá de la *unit* o *hidden unit* (al pertenecer a las *hidden layers*) que utilicemos. Hay varios tipos dependiendo de la fución de activación, y no es posible predecir cuál de ellos funcionará mejor. Comentamos solamente los usados en este Trabajo Fin de Grado:

- **ReLU** (*Rectified Linear Unit*): Es la opción por defecto típica. Este tipo de *unit* utiliza la función de activación:

$$g(z) = \max\{0, z\}$$

Las ReLUs son fáciles de optimizar porque son muy similares a una *unit* lineal. La segunda derivada de esta *unit* es 0 para todos los valores, y la primera derivada es 1 para todos los valores en los que la *unit* está activa, lo que hace que la dirección del gradiente sea más útil para el aprendizaje que en aquellas *units* que introducen efectos de segundo orden (Goodfellow, I. *et al*., 2016).

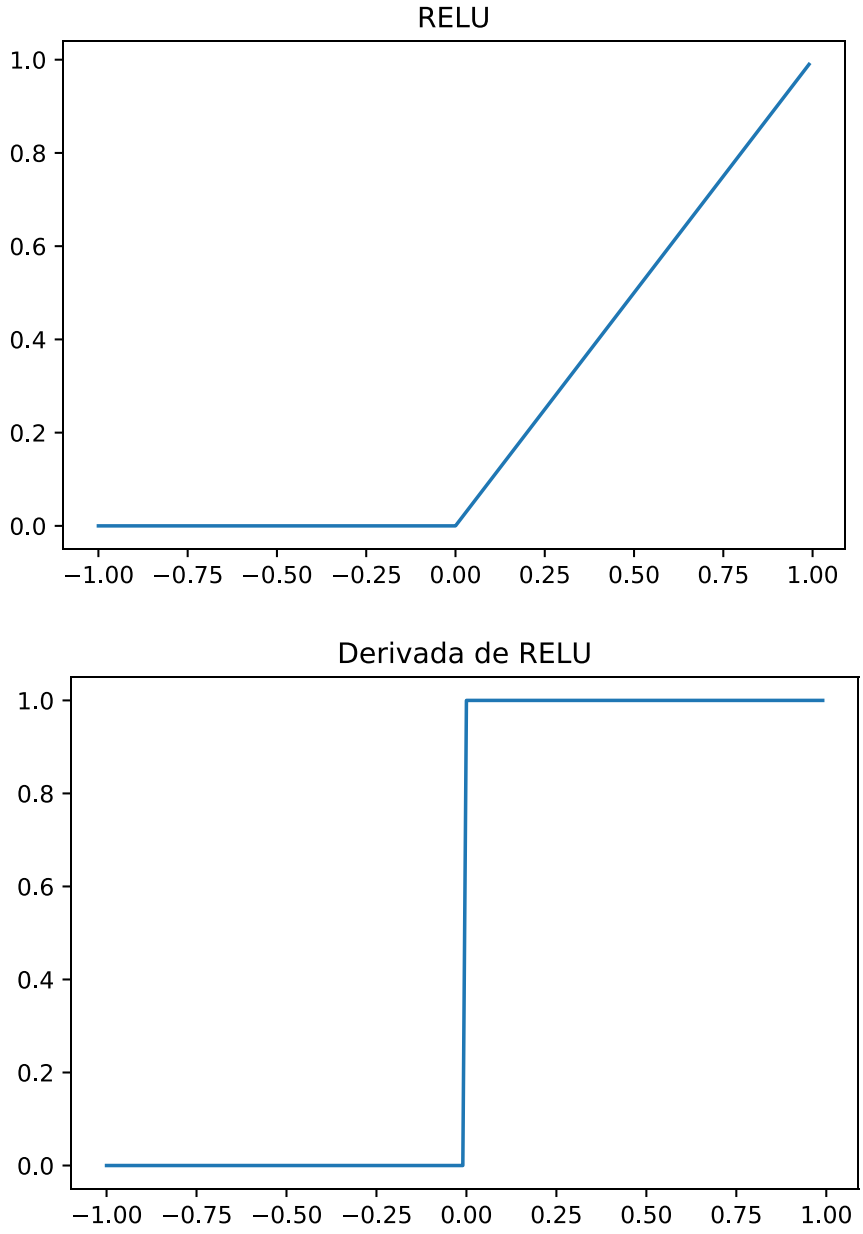


Figura 16. Función de activación de una ReLU y su derivada.

En un modelo lineal, como el visto en el anterior apartado, sería:

$$h = g(WX + b)$$

- **Función sigmoide** (*Logistic Sigmoid*): Utilizadas antes de la introducción de las ReLUs. Su función de activación (sigmoide estándar) es:

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

El problema de estas *units* es que la función toma un valor muy alto para z muy positivas y muy bajos para z muy negativas, solamente siendo realmente sensible para z cercanas a 0. Este hecho hace que el aprendizaje basado en el gradiente sea más difícil. Sin embargo, las redes neuronales recurrentes que veremos más adelante tienen unos requisitos que hacen que estas *units* sean mejor opción que las ReLUs (Goodfellow, I. *et al*., 2016).

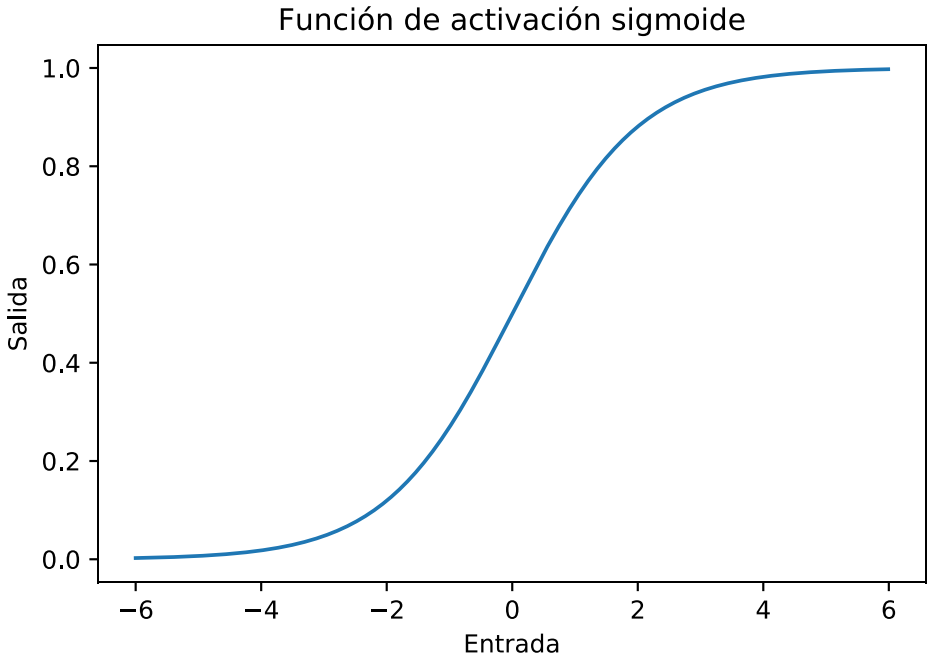


Figura 17. Función de activación sigmoide.

- **Función tangente hiperbólica** (*Hyperbolic Tangent*): La función de activación de estas *units* es:

$$g(z) = \tanh(z)$$

Está relacionada con la sigmoide ya que la sigmoide estándar es una función tangente hiperbólica escalada y con offset:

$$\tanh(z) = 2\sigma(2z) - 1$$

Normalmente en los casos en los que la función de activación sigmoide se tiene que utilizar, la función de activación tangente hiperbólica rinde mejor. Esto es así, porque se parece más a la función identidad, ya que tanh(0) = 0 mientras que $\sigma(0)=\frac{1}{2}$, lo que hace que el entrenamiento de una red neuronal profunda se parezca al entrenamiento de un modelo lineal haciéndolo más sencillo (Goodfellow, I. *et al.*, 2016).

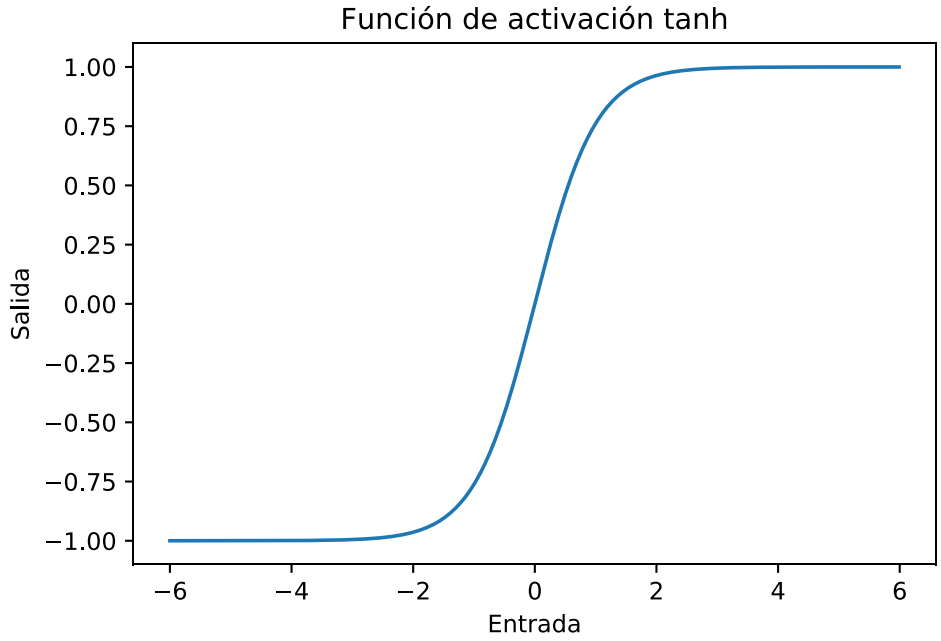
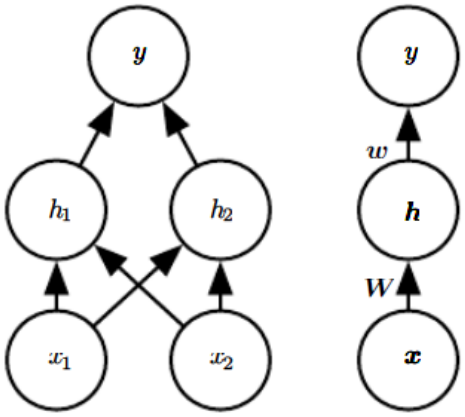


Figura 18. Función de activación de tangente hiperbólica.

Un ejemplo de red neuronal muy sencilla podría ser el que se muetras en la Figura 19 (Goodfellow, I. *et al.*, 2016), se trata de una red *Feedforward* con una *hidden layer* de dos *units* representada de dos formas, una en la que cada *unit* es un nodo (Figura 19.a) y otra en la que se representa un nodo vectorial, que contiene las dos *units* (Figura 19.b).



(a) (b)

Figura 19. Feedforward network modelo simple (a) y vectorial (b).

En las redes neuronales se realiza lo que se conoce como la ***back-propagation***, es decir, la propagación hacia atrás (hacia las capas anteriores) de los resultados y costes para que cada *unit* ajuste los parámetros en busca de un mejor resultado, es decir, aquel que minimice el error o coste. De esta forma, la red puede ir cambiando el valor de sus parámetros, hasta que consiga el resultado óptimo para esa red (Udacity, 2018).

Otro aspecto importante en las redes neuronales es la **regularización** (Udacity, 2018). La regularización es una ayuda para la optimización de la red. Cuantos más parámetros tenga una red neuronal mayor cantidad de datos son necesarios para mejorar su eficiencia, y normalmente, las redes neuronales trabajan con una gran cantidad de parámetros, pero, a través de la regularización, se puede reducir el número de parámetros libres, sin aumentar la complejidad de la red, ayudando a evitar en gran medida el *overfitting*. Hay varios tipos o técnicas de regularización, pero a continuación se comentan los que se han utilizado en este trabajo:

- ***Early stopping**:* Consiste en parar el entrenamiento cuando se detecta que el error en validación ha aumentado en varias iteraciones seguidas, quedándose con el modelo (pesos y sesgos de la red actualizados) que mejor precisión ha alcanzado hasta dicho momento.

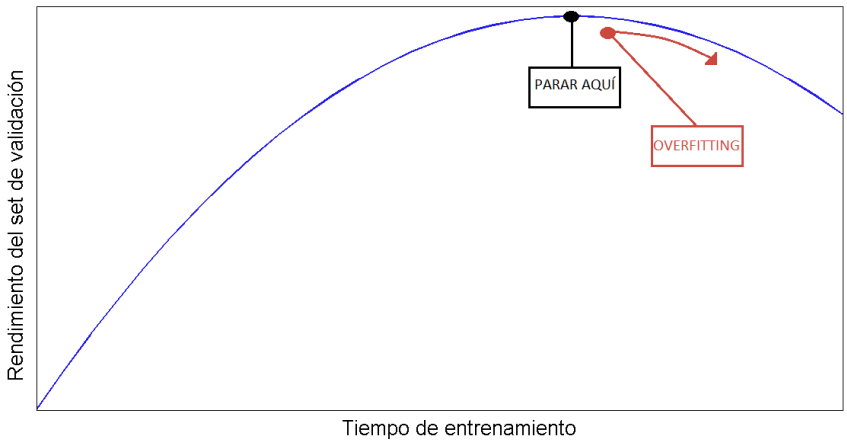


Figura 20. Técnica de Early stopping.

- **Regularización L2:** Consiste en añadir una penalización a la función de coste, obteniendo:

$$\varepsilon' = \varepsilon + \beta \frac{1}{2} \|w\|_2^2$$

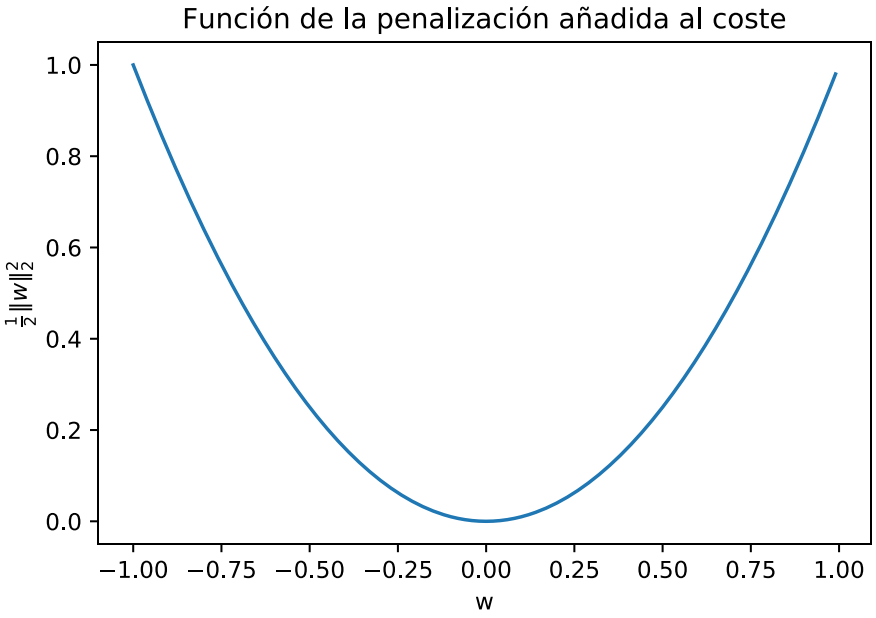


Figura 21. Función de penalización en la regularización L2.

Es una regularización simple, ya que solamente se añade la penalización a la función de coste, sin cambiar la estructura de la red.

- **Dropout**: Es una técnica más reciente que funciona realmente bien. Para cada uno de los ejemplos con los que se entrena la red, aleatoriamente, se ponen a cero (un porcentaje que se elija) las *activations* o valores de activación de cada capa. De esta forma, se realiza un aprendizaje redundante y robusto, siendo el resultado un consenso de varias redes (si imaginamos cada uno de los resultados como el resultado de una red independiente), obteniendo una mejora del rendimiento (Udacity, 2018).

Estás técnicas de regularización son controladas en el momento del entrenamiento, es decir, si se utiliza *Early Stopping*, quien realiza la red decide cuántas iteraciones debe estar aumentando el coste en validación para dar por finalizado el entrenamiento por considerar que empieza a haber *overfitting*, aunque lo habitual es poner como máximo 5 iteraciones. De igual manera, se puede elegir la penalización que se añade a la función de coste en la regularización L2 y el porcentaje de *activations* que se ponen a 0 en Dropout.

De cara a entrenar una red, es importante saber con qué datos se está trabajando, hay que saber la información relevante de esos datos para facilitar el aprendizaje de la red, mejorando su rendimiento. De esta forma, se utilizan distintos tipos de redes en función de lo que se quiera aprender, entre los que destacaremos los dos tipos de redes que utilizamos en este trabajo Fin de Grado: las redes neuronales convolucionales (utilizadas generalmente para trabajar con imágenes) y las redes neuronales recurrentes (utilizadas generalmente para trabajar con secuencias temporales).

Redes Convolucionales (CNN)

Una red convolucional es simplemente una red neuronal que utiliza la convolución en lugar de la habitual multiplicación de matrices en al menos una de sus capas (Goodfellow, I. *et al*., 2016).

En primer lugar, explicamos lo que es una convolución:

En su forma más general, una convolución es una operación que promedia basándose en unos pesos:

$$s(t) = \int x(a)w(t-a)da$$

$$s(t) = (x * w)(t) = x(t) * w(t)$$

En la terminología de las redes convolucionales, $x(t)$ es la entrada, $w(t)$ es el *kernel* (también conocido como *patch*) y $s(t)$ es la salida, que se refiere a un f*eature map* o mapa de características.

Como estamos trabajando con señales muestreadas en tiempo, la expresión de la convolución pasará a ser:

$$s(t) = (x * w)(t) = \sum_{a=-\infty}^{\infty} x(a)w(t-a)$$

En Machine Learning, las entradas a estas redes son a*rrays* multi-dimensionales, denominados **tensores**. Además, estas convolucionales no las vamos a realizar solamente en una dimensión, sino en dos, es decir, vamos a utilizar un *kernel* bidimensional. Tomando la entrada como I y el *kernel* como K, la expresión de la convolución bidimensional será:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(m,n)K(i-m, j-n)$$

Y gracias a la propiedad conmutativa de la convolución, se obtiene una expresión más utilizada en los algoritmos de Machine Learning al ser más directa de implementar:

$$S(i,j) = (K * I)(i,j) = \sum_m \sum_n I(i-m, j-n)K(i,j)$$

Aunque muchas bibliotecas de redes neuronales implementan lo que se conoce como la correlación cruzada, que es como una convolución, pero sin dar la vuelta al *kernel:*

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(i,j)$$

Este volteo del *kernel*, en ocasiones no es relevante en las redes neuronales. De esta forma, la convolución en una matriz se puede ver como:

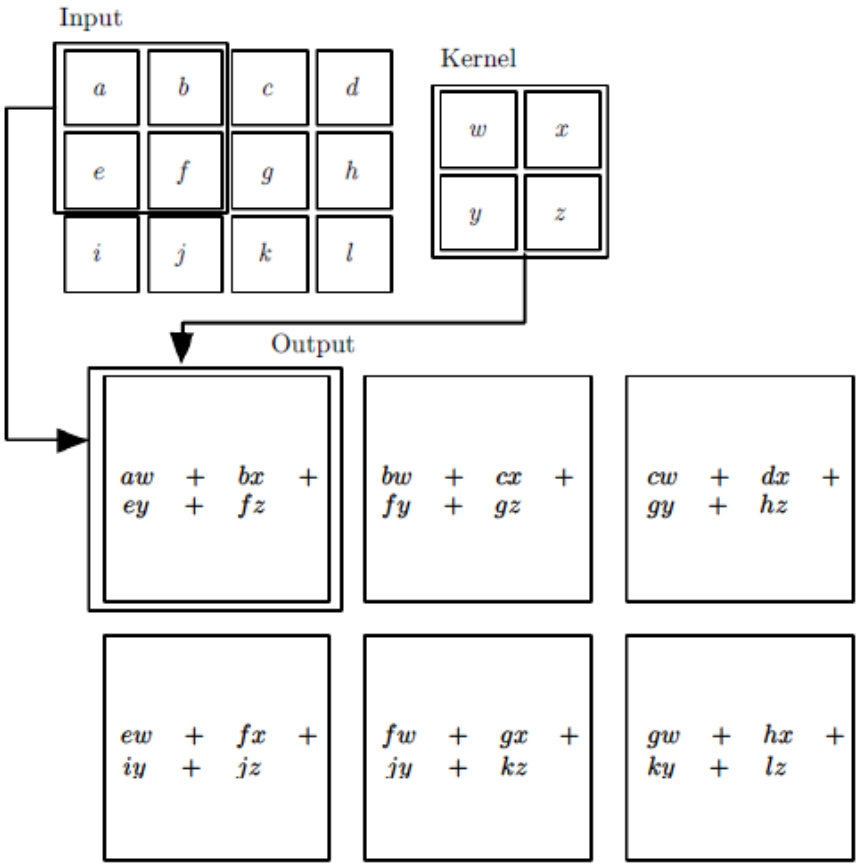


Figura 22. Operación que realiza una capa convolucional en una red.

En el caso de la Figura 22 (Goodfellow, I. *et al*., 2016) se tiene una convolución 2x2, es decir, el *kernel* es una matriz 2x2, y con un *stride* de 1. Este *stride* es el número de pasos que se mueve el *kernel* de una sola vez por la matriz de datos

Este tipo de redes neuronales, formadas, conceptualmente, por una serie de capas, utilizan lo que se conoce como "Weight sharing", es decir, se comparten los pesos entre las entradas que contengan la misma información y se entrenan de forma conjunta. Esto quiere decir que, por ejemplo, en imágenes, si en dos ejemplos que se introducen a la red aparece el mismo objeto, pero ubicado en una zona distinta en cada imagen, los pesos de esas dos entradas se compartirán y entrenarán de forma conjunta, ya que lo que se quiere es que la red aprenda lo que es el objeto, sin importar su ubicación en la imagen, es decir, se consigue una invarianza espacial.

En este caso se realiza un "Weight sharing" espacial, realizado mediante convoluciones.

El efecto que tiene una capa convolucional sobre el tamaño o la cantidad de datos, es el de añadir ***feature maps***, es decir, se utilizan distintos *kernels* (del mismo tamaño, pero con valores distintos), y por cada uno de ellos se genera un *feature map* distinto, lo que representa la profundidad de la capa.

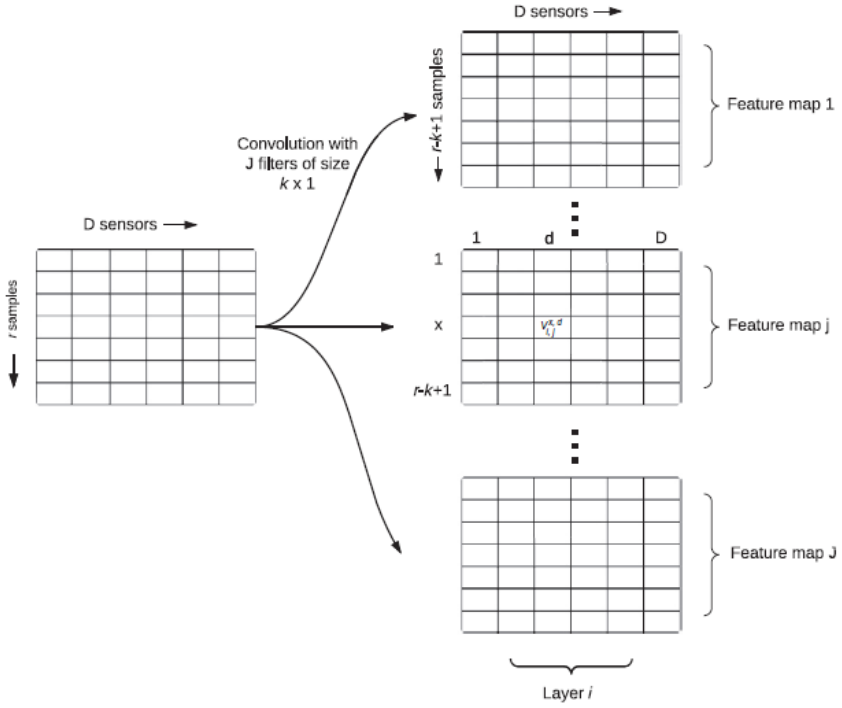


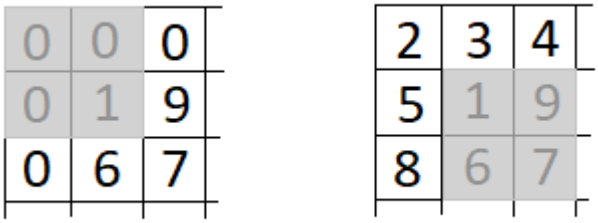
Figura 23. Salida de una capa convolucional.

En la Figura 23, de (San *et al.*, 2017, pp. 186-204), que representa solamente una capa convolucional, el tamaño del *kernel* sería kx1, y habría J *feature maps*.

Como se explicará más adelante, en este Trabajo Fin de Grado, al trabajar con convoluciones 2D, se necesita a la entrada de la red convolucional, un tensor 4D, en el que la 4ª dimensión es un 1. Esta dimensión irá cambiando en función del número de *feature maps* que se quieran en cada capa, con lo que el valor de esta 4ª dimensión a la entrada de la red se puede entender como que tenemos solamente 1 *feature map* de las características (*features*) con las que entrenamos.

Hay otro aspecto en las capas convolucionales a tener en cuenta, el ***padding*** o padeado (Udacity, 2018). Esta técnica permite ajustar el *kernel* al tamaño de los datos y hay dos tipos:

- ***Same***: En este caso, en los límites de la matriz de datos, se añaden una fila y columna a cada lado, llenado de ceros (Figura 24.a)
- ***Valid***: En este caso, en vez de padear con ceros como en el caso anterior, lo que se hace es que no se tiene en cuenta la última y fila y columna en cada caso (Figura 24.b).



(a) (b)

Figura 24. Padding same (a) y valid (b).

La forma en que vamos a ayudar al rendimiento de esta red es mediante lo que se conoce como:

- ***Pooling***: En vez de utilizar un *stride* grande para ir reduciendo el tamaño de la imagen, lo cual eliminará mucha información, utilizamos un *stride* de 1 y, tras la convolución, combinamos los resultados de cada convolución de alguna manera:
    - ***Max-pooling***: Es la forma más común (y la que se utiliza en este trabajo). Consiste en quedarse con el máximo de los datos de la ventana (*kernel*) que se elija (2x2, 3x3…). La estructura habitual de la red neuronal incluyendo este tipo de *pooling* es de incorporar esta capa después de una capa de convolución (Udacity, 2018).

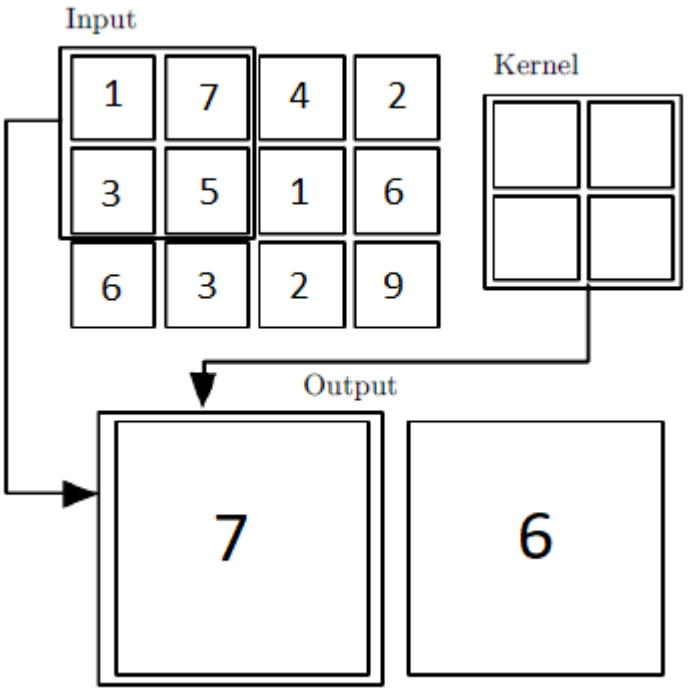


Figura 25. Operación de max-pooling.

    - ***Average-pooling***: En este caso, en vez de quedarnos con el máximo de los datos de la ventana, se realiza un promedio de dichos datos (Udacity, 2018).

Por lo tanto, al diseñar una capa convolucional, hay que definir: el tamaño del *kernel*, el *stride*, el tipo de *padding* y el número de *feature maps* que se van a tener a la salida. En la capa en la que se realiza el *pooling* también hay que definir el tamaño del *kernel* y el *stride*.

Redes Recurrentes (RNN)

Al igual que las redes neuronales convolucionales, este tipo de redes utiliza el "Weight sharing", pero en este caso, en vez de espacial, se comparte de manera temporal.

Al trabajar con secuencias, la red estará compuesta conceptualmente por una serie de celdas que mantendrán un cierto estado correspondiente a cada unidad de tiempo considerada. Al tratarse de una secuencia, la *back-propagation* con la que actualizamos los pesos de la red, interesaría poder hacerla hasta el inicio de la secuencia para tener una idea de la variación de toda la secuencia a lo largo del tiempo, sin embargo, en la práctica, con secuencias largas, esto no va a ser posible, realizándose esta propagación hasta donde se pueda (Udacity, 2018).

Toda esta propagación actualiza los pesos de cada celda de la red, aplicándose, por ser "Weight sharing" sobre los mismos parámetros, con lo que tendremos muchas actualizaciones correladas, algo que nos puede llevar a los fenómenos conocidos como:

- ***Exploding gradient***: A medida que se realiza la propagación de las actualizaciones, el incremento de la actualización cada vez es mayor, tendiendo a infinito, llevando a la red a ser incapaz de aprender.
- ***Vanishing gradient***: Es el fenómeno opuesto, en el que llega un momento que la actualización tiende a 0, siendo incapaz de aprender más la red.

La forma de evitar estos dos problemas es mediante dos técnicas, una para cada uno de los problemas:

- ***Gradient clipping***: Se utiliza para evitar el *exploding gradient* sin un gran coste. Se define un valor de actualización máximo (Δmax) y no se deja superarlo.

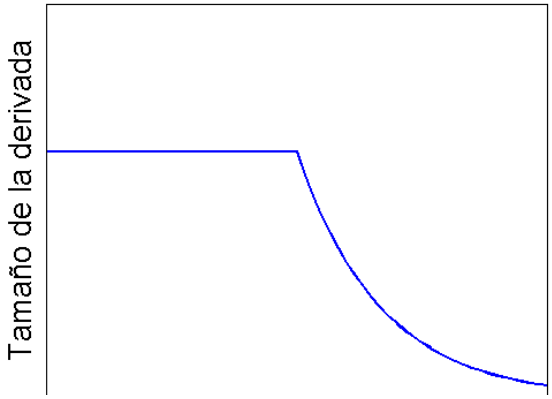


Figura 26. Ejemplo de gradient clipping aplicado a un gradiente que empieza a tender a infinito.

$$\Delta w \leftarrow \Delta w \frac{\Delta max}{\max(|\Delta w|, \Delta max)}$$

En la Figura 26 se ve que a medida que se propaga hacia atrás, el tamaño de la derivada va aumentando, lo que llevaría al *exploding gradient*, pero gracias a este sistema *gradient clipping*, este aumento se ve frenado en Δmax (Udacity, 2018).

- ***"Memory loss"***: Para evitar el *vanishing gradient*. En vez de intentar mantener el estado con respecto a toda la secuencia, cada celda de la red mantiene el estado con respecto a las celdas más cercanas a ella, olvidándose de la más alejadas. Sin embargo, este método es costoso (Udacity, 2018).

Es por esto último, que existen las celdas **LSTM** (*Long-Short-Term Memory*):

Como se ha indicado antes, las redes neuronales, están formadas conceptualmente, por una serie de celdas, todas iguales, que podrían considerarse, en el caso de las redes recurrentes normales:

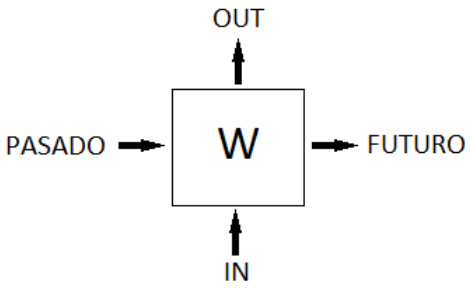
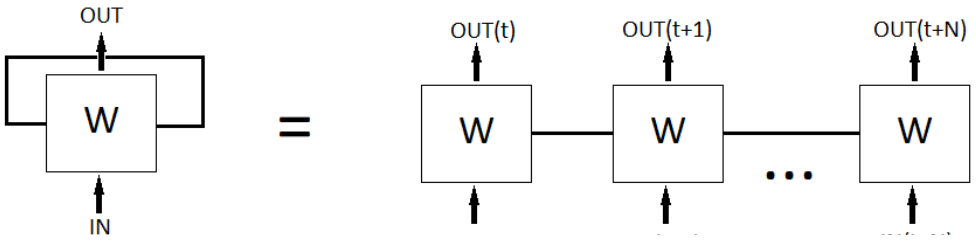


Figura 27. Ejemplo de una celda recurrente básica.

Esta celda, corresponde con una red neuronal en sí misma, y esto es algo que se va a cumplir con todos los tipos de celda de las redes recurrentes:



(a) (b)

Figura 28. Red neuronal recurrente en su versión "comprimida" (a) y unfolded ("desenrrollada") (b).

Sin embargo, en las redes neuronales recurrentes LSTM, la celda pasa a ser:

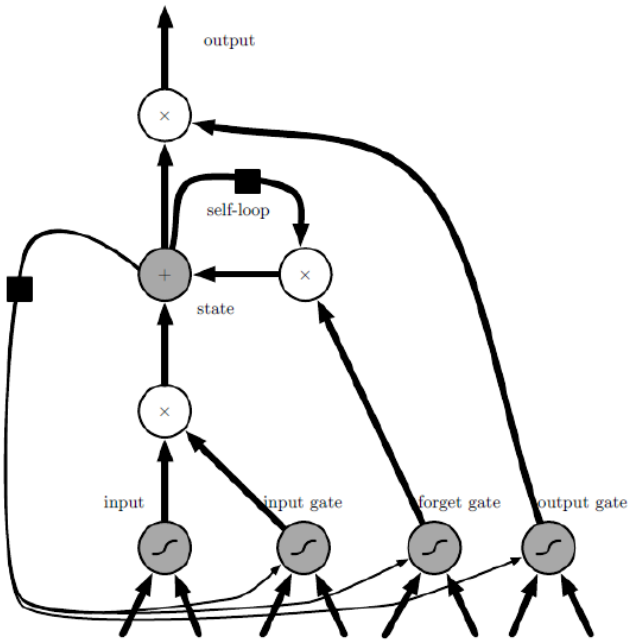


Figura 29. Celda LSTM.

Como se puede ver en la Figura 29 (Goodfellow, I. *et al*., 2016), esta celda trabaja como si fuera una pequeña red neuronal, permitiendo reducir el problema del *vanishing gradient*. Esta celda está constituida por una pequeña unidad de memoria que permite recordar el estado temporal de la red, además de unas unidades multiplicativas llamadas puertas para el control del flujo de la información. Cada celda contiene una "puerta de entrada" que controla el flujo de entrada de *activations* a la unidad de memoria, una "puerta de salida", que controla el flujo de salida de *activations* de la unidad de memoria al resto de la red y una "puerta de olvido", que escala el estado interno de la celda antes de añadirlo como una entrada a la celda a través de conexiones recurrentes propias de la celda, permitiendo resetear de forma adaptativa la memoria (el estado) de la celda.(Sak, H. *et al.*, 2014), lo que hace que evite el problema del *vanishing gradient*. Además, contiene lo que es conocido como "conexiones mirilla" (*peephole connections*) entre la unidad de memoria y las puertas, lo que permite realizar todas las acciones indicadas con una cierta probabilidad dependiendo del contexto de la red y de las entradas y salidas anteriores, no como una decisión binaria. Además de todo esto, permite la aplicación de regularizaciones como L2 (en todas las conexiones) y Dropout (solamente en las conexiones no recurrentes de la celda, es decir, en la entrada y salida, ya que, si se aplica a las conexiones recurrentes, se dificultaría a la celda el mantenimiento del estado) (Zaremba, W. *et al*., 2015).

Como comentario adicional en esta sección, cabe destacar otro tipo de *units*, con las que se han hecho alguna prueba. Estas *untis* son las **GRU** (*Gated Recurrent Unit*), similares a las LSTM, pero más sencillas, al no tener "puerta de salida" y tener menos parámetros, lo que las hace una mejor opción para cuando se tienen menos datos (Cho, K. *et al*., 2014).

En definitiva, las redes neuronales recurrentes son un modelo que permite mapear secuencias de longitud variable a vectores de longitud fija, o bien secuencias de longitud fija a vectores de longitud variable, así como secuencias de longitud variable a vectores de longitud variable. Es por ello por lo que se utilizan en gran medida para *Machine Translation* o *Speech Recognition*, pero en este caso, las utilizaremos para trabajar con la secuencia temporal de los datos provenientes de los sensores (Udacity, 2018).

Con este tipo de redes se trabajará de tal forma que se le introducirá una secuencia de vectores a la entrada, y a la salida se tendrá un vector que indicará la etiqueta de actividad.

3.4 Estado del Arte

En esta sección, se habla de una serie de publicaciones que ralizan el reconocimiento de actividades físicas o movimientos a través del Deep Learning y de los que se han sacado ideas de cara al diseño posterior de las redes a estudiar.

El Reconocimiento de la Actividad Humana (HAR), está siendo ampliamente abordado en los últimos años mediante algoritmos y técnicas tanto de Machine Learning como de Deep Learning más en concreto.

Tras una revisión bibliográfica, se ha podido comprobar que, hasta hace poco, seguía interesando resolver el problema HAR capturando los datos mediante otros sistemas que no eran los sensores inerciales, como son las cámaras *Kinect*, por ello, se han encontrado trabajos en los que se proponían sistemas que pudieran transformar un sistema completo de reconocimiento de actividades de la modalidad en la que se recogían los datos mediante las cámaras *Kinect* a la modalidad en la que se los datos eran recogidos mediante sensores inerciales, y viceversa (Baños, O. *et al*., 2012b).

Sin embargo, ya de hace pocos años hasta aquí, los esfuerzos ya se han centrado en resolver el problema HAR con sensores inerciales y técnicas de Deep Learning. En este sentido, se han podido encontrar todo tipo de arquitecturas de red para distintos dataset: se han podido ver redes que combinaban capas convolucionales con capas recurrentes con *units* de tipo LSTM (Ordóñez, F. J. y Roggen, G., 2016), la utilización de redes completamente convolucionales, pero en la que los datos se disponen en forma de "bandas", representando la secuencia del movimiento (Zhang, R. y Li, C., 2015, pp. 13-23), otras redes en las que se hacen uso de espectrogramas para sacar características significativas de los datos procedentes de los sensores de un teléfono móvil (Ravì, D. *et* al., 2017, pp.56-64) o sistemas en los que se utilizan otros algoritmos de Machine Learning para hacer un preprocesado más exhaustivo de los datos y poder sacar características más significativas, además de aplicar la técnica de la ventana, para, posteriormente, introducirlas en una red neuronal (Hassan, M. M. *et al*., 2018).

Pero no hay que olvidar que los algoritmos en Machine Learning, y, por lo tanto, las redes neuronales en Deep Learning son *ad-hoc*, se diseñan específicamente para unos datos concretos. En ese sentido, lo interesante para este Trabajo Fin de Grado era encontrar trabajos que utilizaran algoritmos de Machine Learning o bien diseñaran redes neuronales, específicamente para los datos del dataset a utilizar, REALDISP.

De estas características se quieren destacar cuatro trabajos:

El primero de ellos es uno de los propios diseñadores del dataset REALDISP, en el que utilizan algoritmos de Machine Learning para estudiar el efecto que tiene el desplazamiento de los sensores en la toma de medidas de cara al reconocimiento de los movimientos posterior (Baños, O. *et al*., 2012a). Aunque en este caso indican que este no era el objetivo principal del trabajo, lo añaden como complemento al objetivo primario, que es el de la realización del dataset. De esta forma, utilizando los algoritmos de *Nearest Class Center* (NCC), *K-nearest neighbors* (KNN) y *Decision Trees* (DT) realizan una clasificación del set de datos ideales, datos *self* y datos *mutual* en el caso de 7 sensores desplazados. En la Figura 30 se muestran los resultados que obtienen para cada algoritmo y tipo de set de datos.

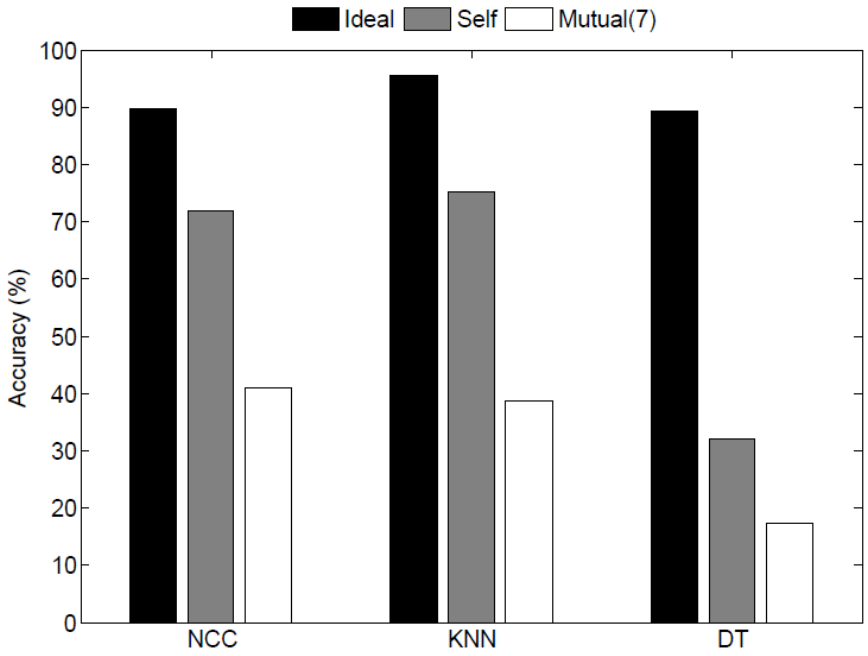
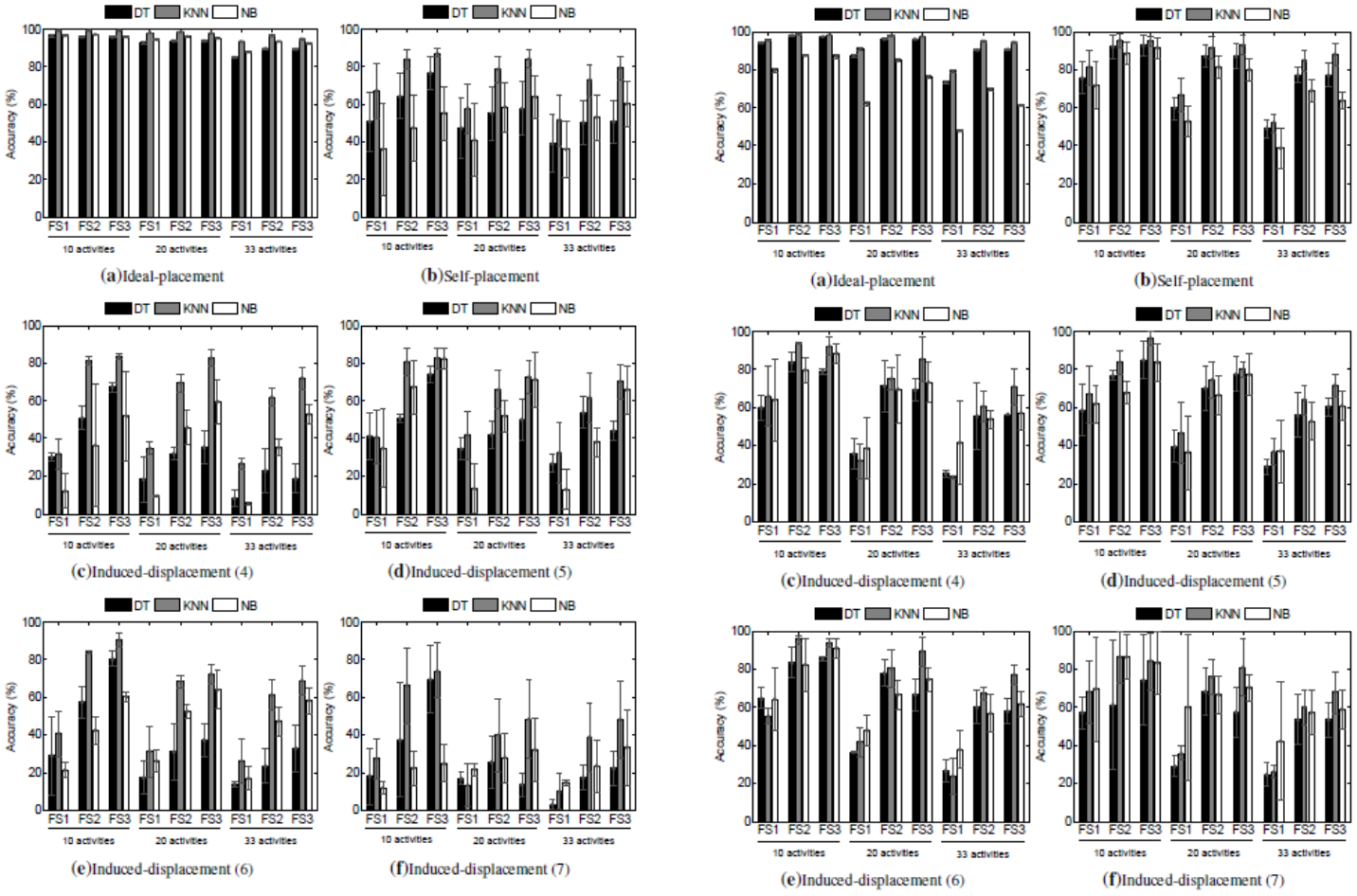


Figura 30. Resultados del reconocimiento de movimientos en (Baños, O. et al., 2012a).

Posteriormente, los mismos diseñadores del dataset REALDISP, realizaron un trabajo en el que el objetivo principal sí era el estudio de los efectos de la captura de datos con sensores desplazados (mal colocados) en el reconocimiento de movimientos (Baños, O. *et al*. 2014b, pp. 9995-10023). En esta ocasión, también utilizan algoritmos de Machine Learning para hacer la clasificación, pero esta vez son *K-nearest neighbors* (KNN), *Decision Trees* (DT) y *Naive-Bayes* (NB). El preprocesamiento de datos que realizan es el uso de una *sliding window* sin *overlap* y diferencian tres *feautre maps* con los que van a hacer pruebas: FS1 = "media", FS2 = "media y desviación estándar y FS3 = "media, desviación estándar y *crossing rate* máxima, mínima y media" (la *crossing rate* es la tasa a la que cambian los símbolos de una señal) y consideran dos tipos de tratamiento previos

a la clasificación, que son *Feature-Fusion Multi-sensor Activity Recognition Chain* (FFMARC) y *Hierarchical Weighted Classifier* (HWC). Además, diferencian tres escenarios, uno en el que solamente trabajan con 10 actividades, otro en el que solamente trabajan con 20 actividades y otro en el que trabajan con 33 actividades (no consideran la clase "No actividad"), además, consideran, al igual que en el anterior trabajo, la clasificación para los tres tipos de set de datos. En la Figura 31 se presentan algunos de los resultados obtenidos en este trabajo para el caso en el que consideran todos los sensores.



(a) (b)

Figura 31. Resultados del trabajo (Baños, O. et al. *2014b, pp. 9995-10023) para FFMARC (a) y HWC (b).*

Como se puede observar en la Figura 31, para el caso FFMARC, el algoritmo KNN es el más fiable, alcanzando un 95% de precisión para el caso en el que los sensores se colocan de forma ideal en el caso más complejo, es decir, FS3 y 33 actividades. También se puede observar que es el algoritmo más fiable, ya que es el que sufre una menor bajada en el porcentaje al utilizar los sensores mal colocados.

En el caso de HWC, las conclusiones son similares. El algoritmo que presenta más fiabilidad y robustez es el KNN. En el caso de colocación ideal de los sensores, los tres algoritmos proporcionan resultados similares, ligeramente inferiores al modelo FFMARC, sin embargo, al igual que antes, el KNN alcanza también un 95% de precisión para el caso más complejo. En lo que destaca el modelo HWC sobre el FFMARC, es que en los casos en los que el posicionamiento no es ideal, se consiguen resultados mucho mejores, especialmente con el algoritmo KNN.

También cabe destacar un trabajo (Zhu, J. *et al*., 2017), que se centra en la extracción de las características y compara sus resultados con los del trabajo de (Baños, O. *et al*. 2014b, pp. 9995-10023). En este trabajo se utiliza otro algormito de Machine Learning, el *Random Forest*, y supera al trabajo de (Baños, O. *et al*. 2014b, pp. 9995-10023), pero se centra en la importancia de la extracción de las características. En este sentido utiliza tanto *features* en el dominio del tiempo (extraídas de las señales provenientes de los sensores y algunas señales procesadas, como pueden ser el cuadrado de esas señales) como en el dominio de la frecuencia (extraídas de las FFTs (*Fast Fourier Transforms*) de las señales en el dominio del tiempo).

De esta forma, para el domino del tiempo utiliza *features* como la media, la desviación típica o la energía de la señal y para el dominio de la frecuencia utiliza otras como el índice de la componente en frecuencia con la magnitud mayor, una media en frecuencia o la energía de algunas bandas de la FFT.

Como comentario final de este trabajo, las precisiones máximas que consiguen son un 99,4% de precisión para los datos de colocación ideal sin la actividad "No actividad" y separando los sets de entrenamiento, validación y test por sujetos, y un 99,1% de precisión para el mismo escenario, pero incluyendo la actividad "No actividad".

Por último, se quiere destacar un trabajo que sí utiliza técnicas de Deep Learning (San *et al.*, 2017, pp. 186-204), en concreto, una red convolucional que se puede ver en la Figura 32, con la que consiguen (sin incluir la "No actividad", lo que reduce drásticamente la cantidad de datos) un 90,1% de precisión para el conjunto de datos en los que los usuarios se colocan los sensores (*self*). Además, utilizan una *sliding window* sin *overlap* de 1 s de duración para aumentar la cantidad de datos, es decir, muy pequeña, lo que lleva a que algunas actividades no se capturen completas, y con ello, dificulte el entrenamiento. Pero este hecho se suple con la semejanza entre algunas actividades, llegando a como ya se ha dicho, un 90,1% de precisión.

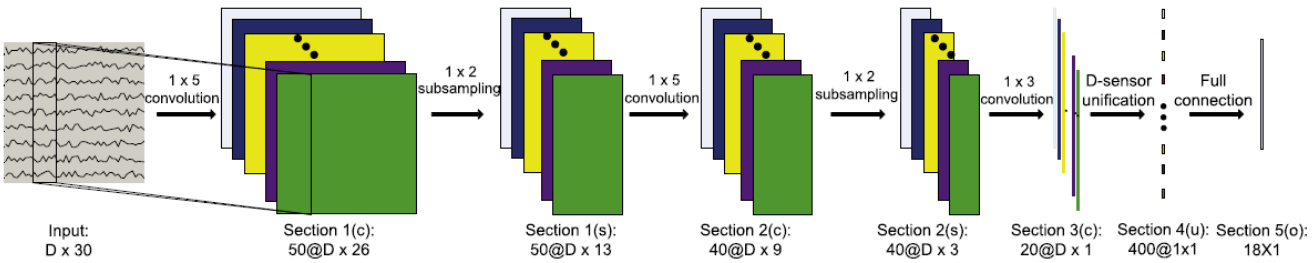


Figura 32. Red convolucional de (San et al.*, 2017, pp. 186-204).*

Esta mejora con respecto a los otros trabajos con los que compara Baños, O. *et al*. 2014b, pp. 9995-10023) en los que se realiza una extracción de características “a mano” como ya se ha visto, los lleva afirmar que “la red convolucional es capaz de descubrir representaciones de características mejores que las realizadas a mano”.

Además, ya para terminar, indican el potencial de estas redes y de las recurrentes (en concreto las que contienen *units* LSTM) para el reconocimiento de actividades.

Tras esta revisión bibliográfica, se llegó a la conclusión de la necesidad de centrarse en el diseño de la red neuronal, y no tanto en la extracción de las características (*features*) para el entrenamiento, por la capacidad de la propia red neuronal de sacar características significativas de los datos que se le proporcionan.

Capítulo 4 Desarrollo de redes neuronales

En este capítulo se describen las distintas **arquitecturas de red neuronal** desarrolladas durante este Trabajo Fin de Grado, además de todo el proceso que se lleva a cabo desde que se obtienen los datos hasta que la red es entrenada y lista para su evaluación.

4.1 Pruebas previas con redes

Como introducción al Reconocimiento de Actividades Humanas, se realizaron una serie de pruebas previas al objetivo último de este Trabajo Fin de Grado y en este apartado se muestran algunas de ellas:

HAR-CNN+LSTM: Esta prueba supuso una primera toma de contacto con el problema HAR y en ella se realizó una red compuesta tanto por capas convolucionales como por capas recurrentes LSTM para el reconocimiento del dataset anteriormente explicado "Dataset para Reconocimiento de Movimientos utilizando Smartphones". Esta red se basa en la de https://github.com/healthDataScience/deep-learning-HAR ,de la que se han hecho ligeras modificaciones y con la que se ha conseguido una precisión del 89,08% en test.

Ya con el dataset REALDISP, es decir, con los datos que en realidad vamos a utilizar para el desarrollo de las redes explicadas en el siguiente apartado ***4.2 Diseño y estudio de redes neuronales,*** se hicieron una serie de pruebas en las que se probaron redes para reconocer solamente 4 de las 34 actividades incluidas en este dataset. No se muestran todas las pruebas, solamente las que mejor precisión dieron:

- **Prueba 1**: Red que consta de 4 capas convolucionales y 2 recurrentes con *units* LSTM. En este caso después de cada capa convolucional no se hacía el *max-pooling*. La precisión de esta red en evaluación para 4 actividades del dataset REALDISP es de un **88,15%**.

- **Prueba 2**: Esta red consta solamente de una capa convolucional, seguida de una de *max-pooling* y 2 capas recurrentes con *units* LSTM. La precisión de esta red en evaluación para 4 actividades del dataset REALDISP es de un **88,46%**.

- **Prueba 3**: En este caso hay 4 capas convolucionales, seguidas de una de ellas por una capa de *max-pooling* y después, se termina con 2 capas recurrentes con *units* LSTM. La precisión de esta red en evaluación para 4 actividades del dataset REALDISP es de un **93,65%**.

- **Prueba 4**: Intentando mejorar la anterior, se añadieron, al principio de la red, 2 capas convolucionales, con sus respectivas capas de *max-pooling*, quedando 6 convolucionales y 2 recurrentes con *untis* LSTM. En este caso, para las mismas 4 actividades que en el caso anterior, se obtuvo un **96,84%** de precisión en la evaluación de la red.

- **Prueba 5**: Finalmente, con la que mejor resultado se obtuvo fue con una red de 6 capas convolucionales, seguidas cada una de ellas por sus respectivas capas de *max-pooling*. La precisión obtenida para las 4 actividades fue de un **98,13%**.

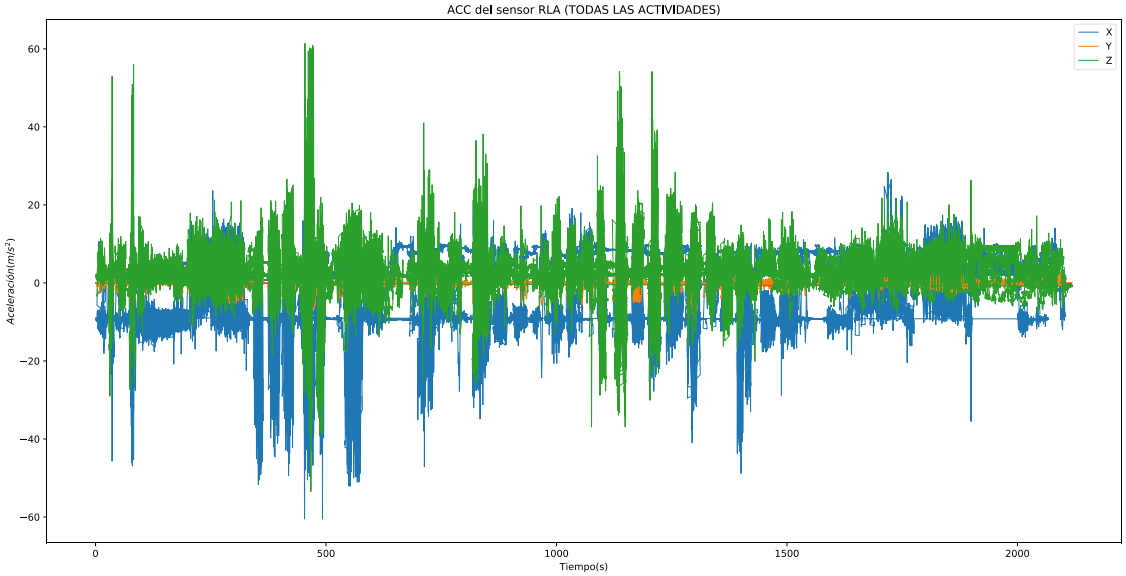
4.2 Diseño y estudio de redes neuronales

4.2.1 Preprocesamiento de los datos del dataset

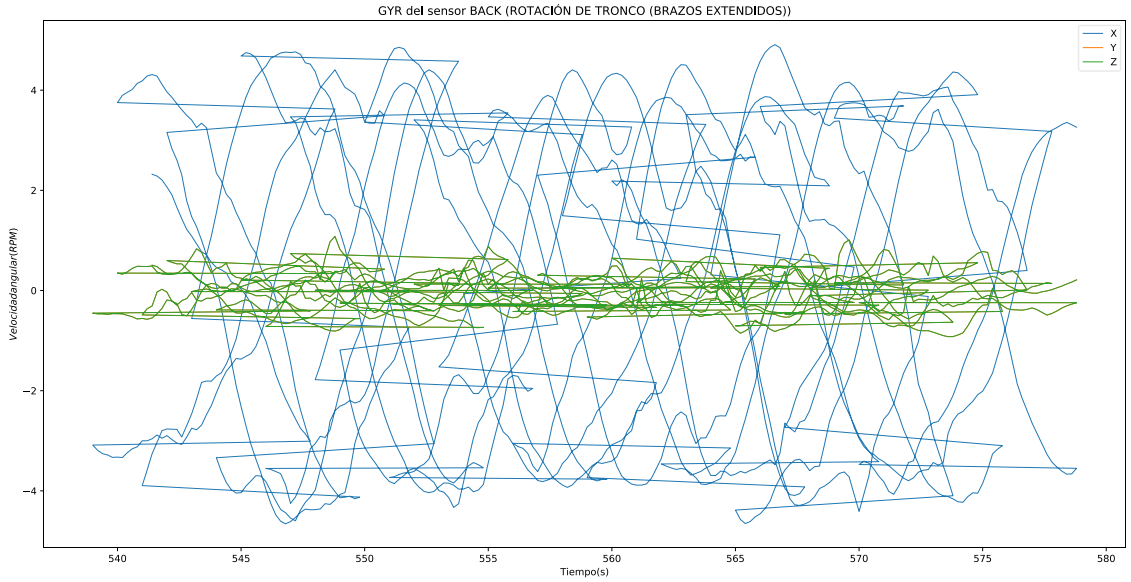
Antes de comenzar a entrenar la red, se hizo un estudio de los datos, y se preprocesaron para para darles un formato adecuado para la red.

Como se ha indicado en la sección 2.3.2, los datos del dataset REALDISP, vienen repartidos en 43 ficheros de datos en forma de matriz, en la que la 1ª y 2ª columna determinan los tiempos de captura del sensor, de la 3ª a la 119ª columna son las medidas del acelerómetro, giroscopio, campo magnético y cuaterniones y la 120ª y última columna son las etiquetas de actividad.

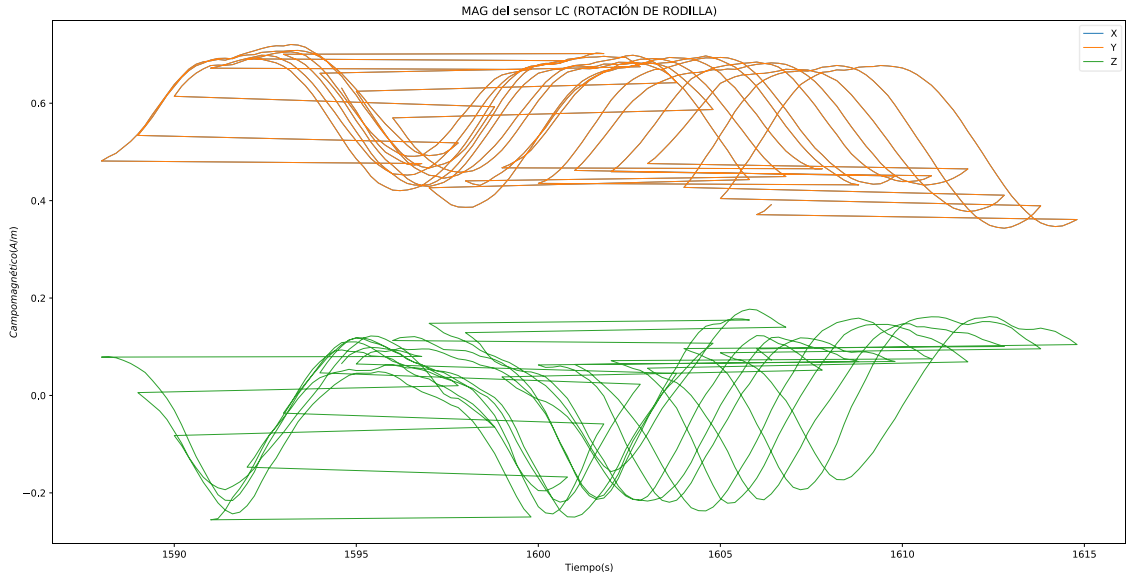
En la Figura 33 se representan datos del acelerómetro, giroscopio, magnetómetro y cuaterniones, así como un ejemplo de las etiquetas de actividad de uno de los ficheros, en concreto del perteneciente al sujeto 10, en el caso de colocación ideal de los sensores.



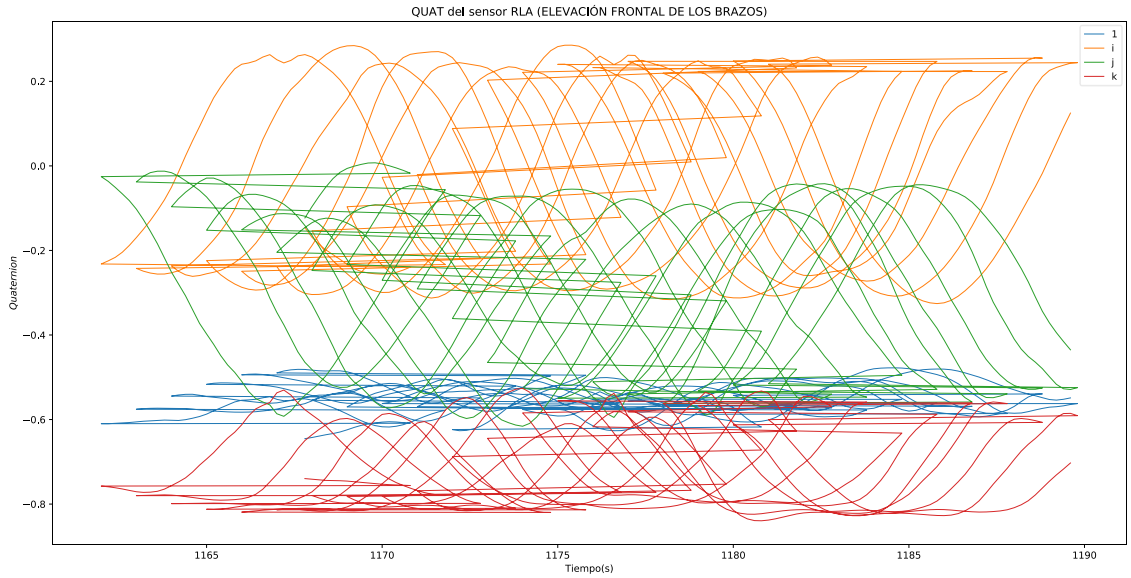
(a)


GYR del sensor BACK (ROTACIÓN DE TRONCO (BRAZOS EXTENDIDOS))
X
Y
Z
Velocidadangular(RPM)
Tiempo(s)

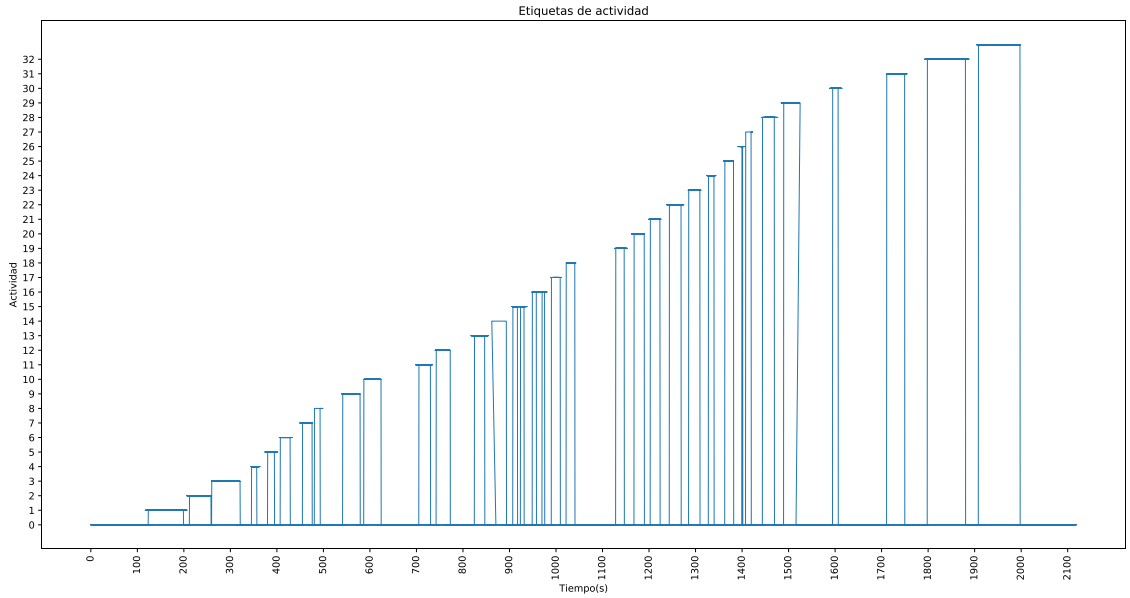
(b)


MAG del sensor LC (ROTACIÓN DE RODILLA)
X
Y
Z
Camponmagnético(A/m)
Tiempo(s)

(c)


QUAT del sensor RLA (ELEVACIÓN FRONTAL DE LOS BRAZOS)
1
i
j
k
Quaternion
Tiempo(s)

(d)



(e)

Figura 33. Datos del dataset REALDISP.

En la Figura 33.a, se representan todos los datos del acelerómetro recogidos por el sensor RLA en el sujeto 10, por lo tanto, en esa gráfica se encuentran todas las actividades mostradas en la Figura 33.e, y si las superpusiéramos, podríamos ver qué fracción de la señal del acelerómetro del sensor RLA corresponde a cada actividad. Mientras, en las Figuras 33.b-d, se representan los datos del giroscopio, magnetómetro y cuaterniones, respectivamente, para distintas actividades y distintos sensores.

Como comentario adicional a la gráfica de las etiquetas de actividad, Figura 33.e, cabe destacar que, en todos los ficheros, las actividades están separadas por un periodo de inactividad, como se muestra en dicha gráfica.

Una vez vistos los datos, se separaron en función del tipo de sensor que los recogía, es decir, se separaron los datos del acelerómetro, los del giroscopio, los del magnetómetro y los cuaterniones. De esta forma, se pueden utilizar solamente los cuaterniones, ya que los sensores de los que se disponen trabajan principalmente con ese tipo de datos, y, además, el resto de los tipos de datos se tienen para futuros usos (mayor entrenamiento, complemento de los cuaterniones…).

Tras la separación de los datos se tenían 43 matrices (una por cada fichero), compuesta por las mismas filas que antes, pero con solamente 36 columnas (parte real, y las 3 partes imaginarias de cada uno de los cuaterniones de los 9 sensores).

Como en estos 43 ficheros tenemos los datos ideales, los de *self* y los de *mutual*, también se separaron de esta manera, para poder trabajar individualmente con cada uno de los grupos y en caso de necesitar usar dos grupos o los tres, también poder hacerlo. De esta forma, se tienen 17 ficheros de datos ideales (uno por cada sujeto), 15 ficheros de datos *self* (tendría que ser uno por cada sujeto, pero como se ha indicado en la sección 2.3.2 se

tuvieron que eliminar algunos ficheros por estar corruptos) y 11 ficheros de datos "mutual" (tanto del caso de 4 sensores desplazados, como de 5, 6 y 7).

Esta separación tan exhaustiva de los datos permitiría después una lectura más fácil y rápida, además de un uso más cómodo.

Otro preprocesamiento común a todas las pruebas realizadas ha sido el de la estandarización de los datos. En todos los casos, antes de entrenar la red y de incluso colocar los datos de una forma adecuada para la red, se estandarizan para que sigan una distribución de media 0 y desviación típica 1. Para ello, en primer lugar, se concatenan todos los datos de cuaterniones de los ficheros que se van a utilizar en la prueba, obteniendo una matriz (matriz X) de 36 columnas y un gran número de filas, que depende del tipo de datos que utilicemos (ideales, *self*, *mutual*, ideales y *self* …). Posteriormente se hallan la media y desviación típica de cada una de las columnas de esa matriz, es decir, se hallan la media y desviación típica, por separado, de los datos que corresponden a la parte real, imaginaria en i, imaginaria en j e imaginaria en k, de cada uno de los 9 sensores, obteniendo dos vectores de 36 elementos, uno de ellos conteniendo las medias de cada columna y el otro las desviaciones típicas. Para finalizar, se genera una matriz (matriz Y) de las mismas dimensiones que la matriz de la que se han obtenido las medias y desviaciones típicas, y cada columna de esta nueva matriz va a ser:

$$Y_j = \frac{X_j - E\{X_j\}}{\sqrt{E\{(X_j - E\{X_j\})^2\}}}$$

Siendo Y_j la columna j de la matriz Y, X_j la columna j de la matriz X, $E\{X_j\}$ la media de la columna X_j y $\sqrt{E\{(X_j - E\{X_j\})^2\}}$ la desviación típica de la columna X_j (raíz cuadrada de la varianza).

Este proceso se realiza motivado por la corriente muy extendida en Machine Learning de utilizar datos que sigan una distribución con media 0 y desviación típica habitualmente 1, con el mismo fin que se indicaba en la sección 3.2.1, para facilitar la minimización del error.

Durante este proceso de estandarización, en el momento en el que se concatenan los datos de los cuaterniones de cada fichero para obtener una matriz X, en algunas pruebas se ha realizado un *max-pooling* 2x2, con el objetivo de quedarnos con las dos dimensiones (**real**, **i**, **j** o **k**) más significativas de cada cuaternión en dos instantes de tiempo consecutivos, separados por 0.02 s (recordemos que los datos del dataset REALDISP están tomados a 50 Hz). Este *max-pooling* previo también afecta a las etiquetas de actividad, ya que vamos a tener la mitad de filas en la matriz que si no se realizara, por lo que se elige como nueva etiqueta de actividad la etiqueta del último instante de tiempo tomado en la matriz de *pooling* 2x2, como se muestra en la Figura 34. Esta elección está llevando consigo un error, ya que, habrá momentos, en los límites de las actividades, en los que se esté mezclando una muestra de una actividad con una muestra del perido de inactividad, pero consideramos este error despreciable, al darse solamente, como mucho, en 33 ocasiones (el momento en el que hay una muestra de actividad, y la siguiente es de inactividad, y se elige como etiqueta la de la inactividad) de las 2580576 muestras que tiene, por ejemplo, el set de datos ideales.

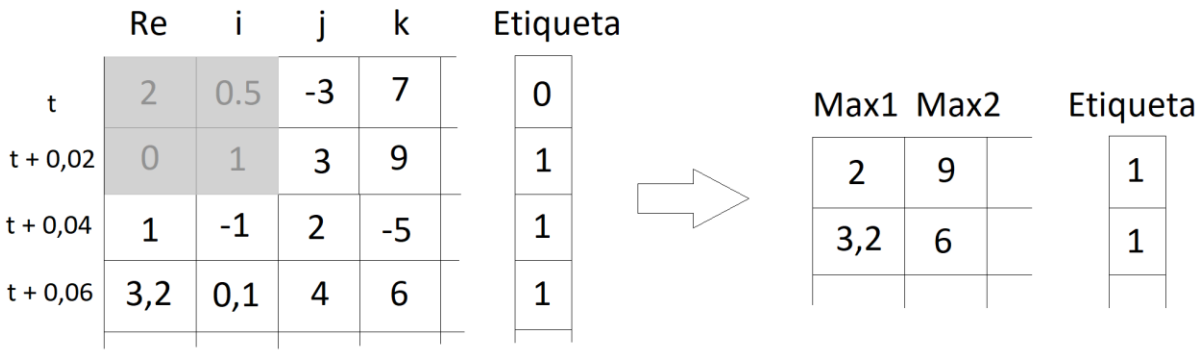


Figura 34. Proceso de max-pooling previo.

Posteriormente, tras la estandarización y previo al entrenamiento, basándonos en los trabajos de la sección 3.4, en todas las pruebas se ha utilizado la **técnica de la ventana deslizante**, aunque en unos casos se ha utilizado con *overlap* y en otros sin *overlap*:

- Ventana deslizante (***sliding window***) **con *overlap***: Esta técnica consiste en utilizar una ventana para producir "segmentos" de los datos con los que se "alimentará" la entrada de la red. En el caso concreto de este Trabajo Fin de Grado, la ventana deslizante es de longitud ***n_time_steps***, que cambiará dependiendo de la prueba que se realice y que se desplaza ***step*** muestras para ir recorriendo toda la secuencia. El valor de *step* debe ser más pequeño que el de *n_time_steps*.

  Los "segmentos" generados se almacenan en un tensor de dimensiones **[nº de filas, nº de *features*, *n_time_steps*]**, siendo el nº de *features* las columnas de la matriz estandarizada, es decir, 36 en el caso en el que no se realiza el *max-pooling* previo y 18 en el que sí se realiza. Una vez se ha obtenido el tensor, se redimensiona para una mejor comprensión: se coloca de la forma **[nº de filas, n_time_steps, nº de *features*]**, pudiéndose ver como una serie de matrices (tantas como indique la dimensión "nº de filas", ya que depende de la cantidad de datos que se tengan) de tamaño **[*n_time_steps*, nº de *features*]** (tamaño de la ventana).
- Ventana deslizante (***sliding window***) **sin *overlap***: En este caso, se utiliza un procesamiento igual al caso en el que sí hay *overlap*, con la diferencia de que, ahora, el parámetro ***step*** es igual al parámetro ***n_time_steps***, por lo tanto, los "segmentos" generados no contienen información que ya haya en otros "segmentos", son disjuntos (se elimina el *overlap*).

Generalmente, utilizar la ventana deslizante con *overlap* lleva a un mejor resultado en la evaluación, ya que se generan más ejemplos de entrenamiento.

Esta técnica también afecta al vector de etiquetas, generando uno nuevo, en el que se va a tener una etiqueta por cada una de las matrices de tamaño **[*n_time_steps*, nº de *features*]**, que se obtiene coniderando como etiqueta del "segmento" a la etiqueta que más se repite durante el "segmento".

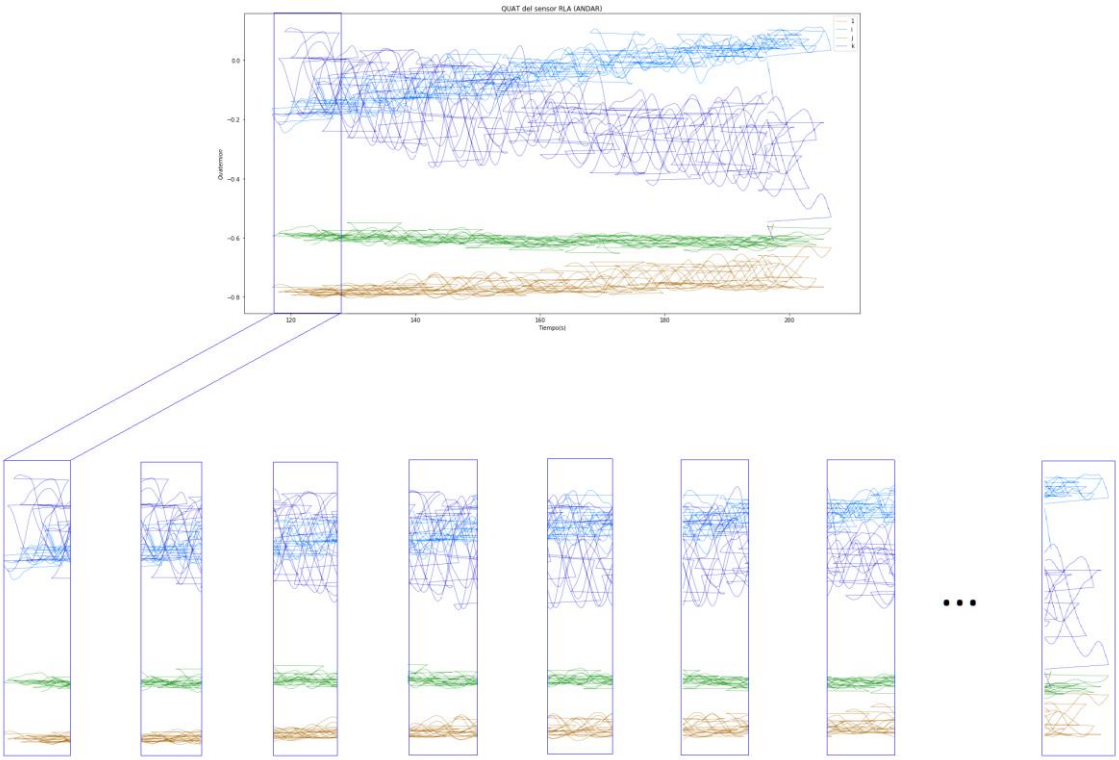


Figura 35. Técnica de ventana deslizante.

En la Figura 35 se muestra la idea conceptual de la técnica de *sliding window* para el caso de la evolución temporal del cuaternión del sensor RLA, en la actividad de ANDAR. En la gráfica se muestra la evolución de la parte real y las tres partes imaginarias del cuaternión del sensor RLA, y se ve cómo la ventana recorre la gráfica extrayendo los segmentos de dimensiones fijas y que, en el caso representado se tiene un ligero *overlap* entre ellos.

Tras este proceso, se deben adecuar las dimensiones del tensor a la entrada de la red, y este adecuamiento va a ser distinto en función del tipo de entrada que se tenga. En este Trabajo Fin de Grado se han trabajado solamente con dos tipos de entradas distintas:

- Entrada convolucional: Para este tipo de entrada, es necesario añadir una cuarta dimensión, que representa la profundidad (correspondiente a los *feature maps*) y que va a ser de tamaño 1 en este caso.
  Por lo tanto, para una entrada convolucional se tendrá un tensor de dimensiones: **[nº de filas, *n_time_steps*, nº de *features*, 1]**.

- Entrada LSTM: En este caso, no es necesario realizar ningún cambio sobre las dimensiones del tensor de "segmentos".
  Así que, las dimensiones del tensor para la red cuando la capa de entrada es una capa de *units* LSTM es: **[nº de filas, *n_time_steps*, nº de *features*]**.

En ambos casos, el vector de etiquetas de actividad se pasa al formato *One-hot encoding*, generándose una matriz de tantas filas como elementos tuviera el vector y con tantas columnas como clases (en este caso, actividades) se quieran aprender.

Finalmente, como último paso previo al entrenamiento, el tensor que contiene los datos de la ventana deslizante para cada instante de tiempo se "barajea" y se separa en los sets de entrenamiento, validación y test, obteniendo un set de entrenamiento que supone un 60% de los datos totales, un set de validación que supone un 15% de los datos y totales y un set de test que supone un 25% de los datos totales (estos porcentajes son aproximados, en ocasiones se han utilizado unos porcentajes ligeramente distintos), siendo, como se indicó anteriormente, el set de test totalmente disjunto a los de entrenamiento y validación. Esta técnica se denomina *random-partitioning* ya que no se distingue entre sujetos para separar los datos, es decir, puede haber distintos datos de un mismo sujeto en los tres sets.

4.2.2 Entrenamiento de la red

La forma de entrenar la red se ha realizado con una serie de hiperparámetros, que se han ido ajustando hasta conseguir el mejor resultado para cada arquitectura de la red. Los hiperparámetros sobre los que se ha actuado en las distintas redes para la mejora del resultado han sido:

- ***n_time_steps***: Como se ha indicado en el subapartado anterior (4.3.1), este parámetro indica la longitud de la ventana para generar los "segmentos" que van a "alimentar" la red.
- ***n_classes***: Indica el número de clases a aprender por la red.
- ***n_channels***: Es el parámetro "nº de features" del subapartado anterior. Indica las características de las que la red aprenderá.

Estos tres hiperparámetros están ajustados a los datos y dimensiones de los que se dispongan, y afectan a la estructura de la red neuronal. Sin embargo, hay otra serie de hiperparámetros que no están ajustados a la estructura de la red y que son los que habitualmente se modificarán en busca de un mejor resultado:

- ***epochs***: Indica el número de épocas que va a tener el entrenamiento, es decir, el número de veces que se pasa todo el set de entrenamiento a la red neuronal para que aprenda de él.
- ***batch_size***: Este parámetro indica con cuántos "segmentos" a la vez se alimenta la red, lo que determinará el número de iteraciones que se realiza en cada época de entrenamiento (*epochs*). Por ejemplo, para un caso en el que tuviéramos en el set de entrenamiento 10 ejemplos de "segmentos" de tamaño [5,4], si este parámetro es *batch_size* = 2, una época de entrenamiento constaría de 5 iteraciones, en la que cada una se le pasan a la red 2 ejemplos de "segmentos" de tamaño [5,4] (En el caso de entrada convolucional se añadiría otra dimensión a los segmentos).
- ***learning_rate***: Como se ha indicado en la sección 3.3, este parámetro es la tasa de entrenamiento.

En el caso de trabajar con alguna capa con *units* LSTM, hay que utilizar otros dos hiperparámetros además de los anteriores:

- ***lstm_layers***: Como su nombre indica, son el número de capas con *units* LSTM con las que se va a trabajar.
- ***num_units***: Este parámetro indica el número de *units* LSTM que tendrá cada una de las capas indicadas en *lstm_layers*. Puede ser un número escalar, con lo que todas las capas las mismas *units* o un vector, en el que se indique el número de *units* que tendrá cada capa.

Finalmente, para todas las redes se van a tener 2 entradas:

- La entrada de los datos.
- La entrada de las etiquetas.

4.2.3 Evaluación de la red

La forma de evaluar la red es recuperando el modelo (red) entrenado y alimentándolo, en este caso, con el set de test de igual manera (en cuanto a dimensiones) que como se ha hecho con el set de entrenamiento y el de validación durante el entrenamiento.

Esta evaluación arrojará un porcentaje, indicando la precisión de la red, y será en base a esta precisión en la evaluación por lo que cambiaremos los valores de algunos de los hiperparámetros para un reentrenamiento de la red.

4.2.4 Arquitecturas de las redes y resultados

Para este Trabajo de Fin de Grado se han diseñado, programado y utilizado distintas arquitecturas paras las redes. Sin embargo, vamos a englobar estos distintos tipos de redes en 3 grupos principalmente:

- **CNN (*Convolutional Neural Networks*)**: Se han realizado pruebas con redes 100% convolucionales, es decir, solamente se utilizan capas convolucionales en la red. Entre las distintas pruebas se han cambiado las formas (2x2, 4x4…) de la convolución, el número de etapas… pero siendo siempre completamente convolucionales.
- **RNN (*Recurrent Neural Networks*)**: Se han hecho unas pocas pruebas con redes completamente recurrentes, en las que solamente se han utilizado celdas o *units* de tipo LSTM (en la gran mayoría) y tipo GRU (por ver su efecto).
- **CNN+LSTM**: Este tipo de redes está basado en los trabajos encontrados acerca de reconocimiento de actividades con técnicas de Deep Learning, como (Ordóñez, F. J. y Roggen, D., 2016). En este tipo de redes habitualmente, las primeras capas son convolucionales y las últimas son recurrentes con celdas de tipo LSTM, de esta forma, las capas convolucionales sirven de preprocesado de los datos para la entrada a las capas recurrentes, con la intención de una mejora en el rendimiento y resultado.

En esta sección se presentan las distintas redes utilizadas, únicamente en su mejor versión, es decir, para un mismo número y tipo de capas en una red solamente se explican en detalle los parámetros y estructura que permiten un mayor porcentaje de acierto, aunque sí se presentarán los porcentajes de las redes de ese mismo tipo, pero con menores porcentajes de acierto.

La notación que se va a seguir para las distintas estructuras es la utilizada en (Ordóñez, F. J. y Roggen, D., 2016), y consiste en:

- $C(F)$: Se refiere a una capa convolucional con *F feature maps*.
- $R(n)$: Se refiere a una capa recurrente con *n units* de tipo LSTM.
- $RG(n)$: Se refiere a una capa recurrente con *n units* de tipo GRU.
- S_m: Se refiere a la última capa, que constituye una capa *fully connected*, es decir, en la que se fusionan todos los resultados de la capa previa, junto con la aplicación de la fusión *softmax*.

Antes de comenzar a ver las estructuras de las redes y sus detalles, se quieren hacer dos comentarios que valen para los 3 tipos de redes que se tratan:

- En primer lugar, se quiere destacar que siempre se han realizado las pruebas con un número de *units*, es decir, *num_units*, múltiplo de 32, ya que esto mejora la velocidad del entrenamiento, debido a que en CUDA la unidad ejecutable más pequeña de paralelismo es de 32 hilos (lo que se conoce como un *warp* de hilos, implementado en hardware), con lo que el *warpsize*, es 32. (NVIDIA Corporation, 2018b)
- Otro comentario que cabe destacar en esta parte es en relación a la longitud de la ventana. En (Baños, O. *et al*. 2014b, pp. 9995-10023) utilizan una ventana de 6 s sin *overlap*, pero esto nos generaría muy pocos segmentos, y por ello, datos, con respecto a lo que se requiere para el entrenamiento de una red neuronal. Es por ello por lo que se ha decidido reducir el tamaño de la ventana (las ventanas utilizadas, generalmente están alrededor de los 2-3 segundos, salvo alguna excepción), pero siempre teniendo en cuenta que éste debe ser suficientemente grande como para recoger, en un único segmento, un ciclo significativo de la actividad. Además, este tamaño de ventana ha sido elegido acorde a a las dimensiones de la red y el número de *features*.

Los resultados obtenidos se van a mostrar a partir de dos tablas y una matriz para cada uno de los tres tipos de redes que se han tratado, así:

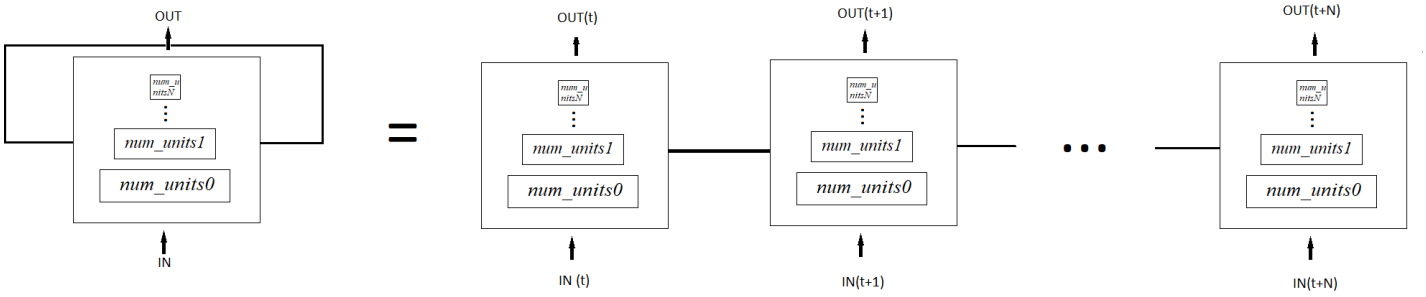
- En la primera tabla se mostrarán algunos hiperparámetros, como: el *overlap* de la *sliding window*, el número de *features* con el que se entrena la red, controlado por *n_channels*, el tamaño de *batch* con el que se "alimenta" a la red, controlado por el *batch_size* y la tasa de aprendizaje o *learning_rate*. Finalmente se mostrará la precisión obtenida en la evaluación de la red.
- En la segunda tabla se mostrará el rendimiento de la red con otros tipos de datos. Esto está motivado principalmente por los trabajos de (Baños, O. *et al*. 2014b, pp. 9995-10023) y (Zhu, J. *et al*., 2017), en el que usan esta forma para mostrar cómo reaccionan las distintas redes a los sets con los que no se entrenan, es decir, al set

de datos *self* y *mutual*, lo que llevaría a ver cómo afecta el mal colocamiento de los sensores en la predicción de la red.
De esta forma se obtiene la tabla, en la que se evalúan las redes con las que se han obtenido mejor resultado, con todos los datos que se disponen del tipo *self*, con solamente 1 sujeto del tipo *self*, con todos los datos disponibles del tipo *mutual* y ocn solamente 1 sujeto del tipo *mutual*, en concreto, uno en el que se tienen 7 sensores descolocados (el peor de los casos).

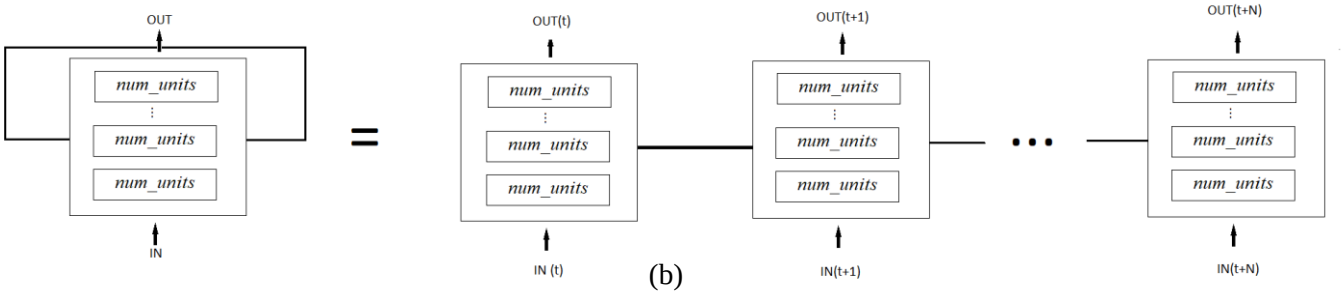
- Finalmente, se han obtenido las matrices de confusión para todos los casos. La matriz de confusión permite ver el rendimiento de una red (o de un algoritmo de Machine Learning), mostrando en cada fila el número de ejemplos o instancias de la clase verdadera y en las columnas el número de veces que la red o algoritmo ha predicho una clase. Cada elemento de la matriz se puede leer como el número de veces que la red predice una actividad *X*, siendo verdadera la actividad *Y*. Por lo tanto, este tipo de matrices interesan que sean diagonales, lo que significaría que la red siempre predice correctamente la actividad.

RNN

En el caso de las redes neuronales completamente recurrentes, como se ha indicado, se han utilizado *units* LSTM y GRU, pero, estas redes las podemos ver, de forma general, como en la Figura 36: varias capas, cada una con un número de *units* en ellas distinto (generalmente, se van reduciendo el número de *units* a medida que añadimos otra capa más al modelo) como en el caso de la Figura 36.a, o todas las capas con el mismo número de *units*, como se representa en la Figura 36.b.



(a)



(b)

Figura 36. Estrucutras de las redes recurrentes utilizadas.

En cuanto a los resultados, expuestos en la Tabla 1, se puede observar que *n*_channels es siempre 18, es decir, en todos los casos se hizo un *max-pooling* previo, como se indicó en la sección 4.2.1. Además. se quiso comprobar en este tipo de redes la importancia del *overlap* al aplicar la ventana deslizante. En este sentido, en el caso en el que no se tiene *overlap*, son necesarias al menos 4 capas con *units* LSTM para conseguir una precisión de un 86,33%, pudiendo alcanzar un 96% con solamente una capa con 128 *units* LSTM si tenemos *overlap*. Por ejemplo, una situación concreta y clara, es en la que se tienen 3 capas recurrentes con 128, 64 y 32 *units* respectivamente y se probó a entrenar la red con los mismos parámetros, pero probando a aplicar la ventana deslizante con *overlap* y sin él, obteniendo un 21% más de precisión cuando se tiene *overlap* que cuando no, dándose esta mejoría tan notoria en todos los casos. Con esta serie de pruebas se comprueba que el uso de la técnica de *sliding window* con *overlap* mejora considerablemente los resultados, siendo este hecho el esperado, al disponer de más datos con los que entrenar la red.

Por otro lado, también se buscó comprobar en este tipo de redes qué forma, de las expuestas en la Figura 36, es decir, en la que el número de *units* se va reduciendo a medida que añadimos otra capa y en la que el número de *units* es el mismo en todas las capas, daba mejores resultados.

En este sentido, se puede ver un ejemplo concreto, en el que se comprobó, para un modelo de dos capas, con los mismos parámetros, un entrenamiento con 128 *units* en las dos capas y otro entrenamiento con 192 *units* en la primera capa y 64 en la segunda. En los dos casos, el número de *units* total es el mismo, 256, pero la distribución es diferente, obteniendo mejor resultado (97,9 %) cuando se utilizan el mismo número de *units* en todas las capas frente al modelo en el que se ut8ilizan más *units* en la primera capa que en la segunda (94,9219 %).

Además, se ha querido ver el efecto de sustituir las *units* de tipo LSTM por una de tipo GRU. Esto lleva al resultado, para una sola capa recurrente de 128 *units*, de un 96,08 % para las de tipo LSTM y un 95,9 % para las de tipo GRU. Como se ve, el resultado no es muy distinto entre ambas, sin embargo, el tiempo de entrenamiento sí difiere, ya que para las de tipo LSTM es de 1 hora, 58 minutos y 34,6 segundos, mientras que para la de tipo GRU, es de 6 horas, 58 minutos, 37 segundos, llevándonos a preferir una de tipo LSTM.

Por último, cabe destacar el efecto del tamaño de la ventana con un pequeño ejemplo, en el que se puede ver que para una red de una capa hay que tener cuidado al determinar dicho tamaño, porque recordemos que a cada segmento se le asigna como etiqueta aquella actividad que más tiempo esté presente en el segmento, y si el tamaño de la ventana es demasiado grande, puede llevar a que ciertas actividades de menor duración estén en segmentos con otras actividades de mayor duración.

Arquitectura	*sliding window*	*n_time_steps*	*n_channels*	*batch_size*	*learning_rate*	Precisión (%)
R(128)-R(64)-R(32)- S_m	Sí	144 (2,88 s)	18	256	10^{-4}	96,7198
R(128)-R(64)-R(32)- S_m	No	144 (2,88 s)	18	256	10^{-4}	75,0391
R(256)-R(128)-R(64)-R(32)- S_m	No	144 (2,88 s)	18	256	10^{-4}	77,4219
R(512)-R(256)-R(128)-R(64)- S_m	No	144 (2,88 s)	18	256	10^{-4}	85,1172
R(512)-R(256)-R(256)-R(128)- S_m	No	144 (2,88 s)	18	256	10^{-4}	86,3281
R(512)-R(256)-R(256)-R(256)-R(128)-R(128)- S_m	No	144 (2,88 s)	18	256	10^{-4}	85
R(128) - S_m	No	144 (2,88 s)	18	256	10^{-4}	72,6172
R(128) - S_m	Sí	144 (2,88 s)	18	256	10^{-4}	96,08
RG(128)- S_m	Sí	144 (2,88 s)	18	256	10^{-4}	**95,9**
R(128) - S_m	No	256 (5,12 s)	18	144	10^{-4}	72,4306
R(128) - S_m	Sí	256 (5,12 s)	18	144	10^{-4}	94,5
R(128)-R(128)- S_m	Sí	144 (2,88 s)	18	256	10^{-4}	**97,9**
RG(128)-RG(128)- S_m	Sí	144 (2,88 s)	18	256	10^{-4}	**95,9815**
R(128)-R(64) - S_m	Sí	144 (2,88 s)	18	256	10^{-4}	97,1492
R(192)-R(64) - S_m	Sí	144 (2,88 s)	18	256	10^{-4}	94,9219

Tabla 1. Resultados de las redes recurrentes diseñadas.

Respecto al rendimiento de las redes con otros tipos de datos (Tabla 2), se obtienen resultados, lógicamente, inferiores a los de la evaluación de la red con los datos del mismo tipo que los datos con los que se entrena la red. Sin embargo, se puede ver cómo las redes con *units* de tipo GRU, a pesar de conseguir resultados peores con respecto a las redes con *units* tipo LSTM, muestran una ligera mejoría cuando se evalúan con los otros tipos de datos, dejando ver que son más robustas frente al desplazamiento de los sensores.

Este hecho lleva a que, a la hora de utilizar la red neuronal en una aplicación real, como puede ser en un juego serio con sensores utilizado para ayudar a realizar ejercicios de rehabilitación, haya que decidir entre un tipo de *unit* u otro. En este sentido, dependería de si los ejercicios se van a realizar bajo la supervisión de un especialista, es decir, si se van a realizar siempre en una consulta en la que los sensores son puestos por el especialista, de igual manera y en la misma posición que cuando se entrenó la red, atendiendo a los resultados obtenidos, parece que podría interesar más utilizar una red con *units* LSTM, ya que responde mejor en esas situaciones. Sin embargo, si los ejercicios los va a realizar el paciente en su domicilio y él mismo se va a poner los sensores para realizarlo, interesaría la opción de una red con *units* GRU.

Arquitectura	Entrenamiento	Precisión (%)				
		Ideal	*Self*		*Mutual*	
			Todos los sujetos	*1 sujeto*	*Todos los sujetos*	*1 sujeto (mutual 7)*
R(128) - S_m	Ideal	94,5	74,0929	77,8586	70,0260	45,0758
RG(128)- S_m	Ideal	**95,9**	79,5209	77,6724	70,0675	56,6623
R(128)-R(128) - S_m	Ideal	**97,9**	74,8204	74,9333	70,0675	50,6510
RG(128)-RG(128) - S_m	Ideal	**95,9815**	79,0599	77,8821	70,0675	51,3021

Tabla 2. Resultados de las redes recurrentes diseñadas con datos de distinto tipo a los de entrenamiento.

Además, se quiere mostrar el proceso de entrenamiento y la matriz de confusión de la red *R(128)-R(128) - S_m* con 97,9 % de precisión, que es la mejor red recurrente conseguida.

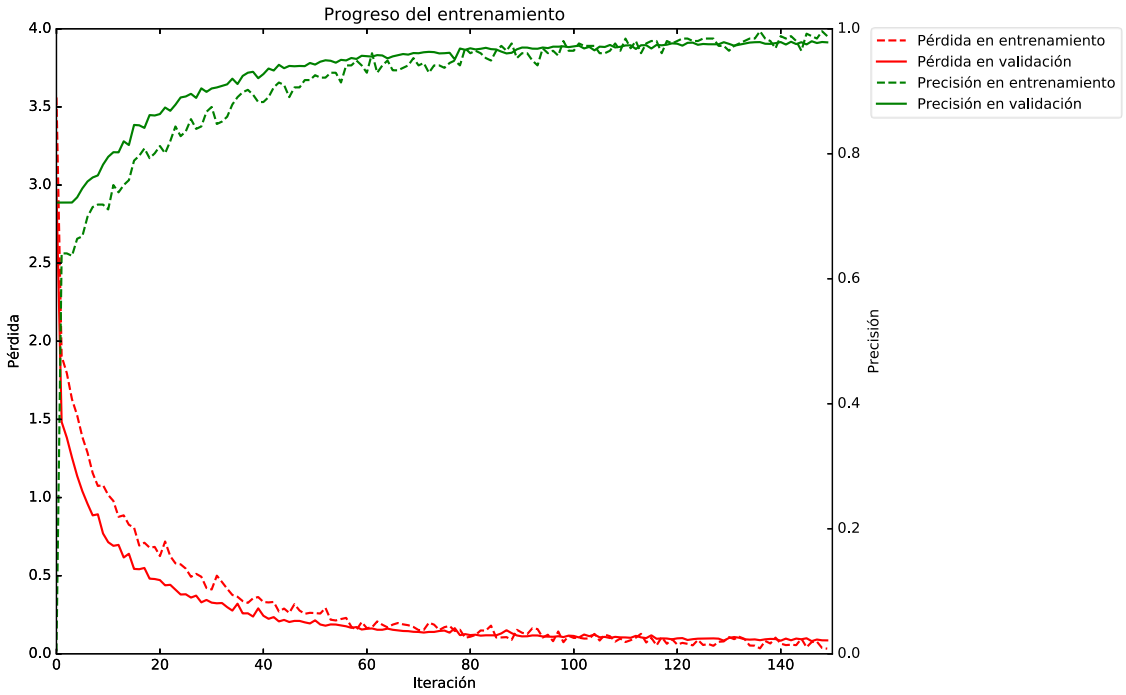


Figura 37. Progreso de entrenamiento de la red R(128)-R(128) - Sm .

Así, en la Figura 37 se puede ver cómo a medida que iba avanzando el entrenamiento, la pérdida (el coste o error) de entrenamiento (y de validación) van disminuyendo, mientras que la precisión va aumentando. En el eje de abscisas se está representando la iteración en la que se encuentra el entrenamiento, mientras que tenmos dos ejes de ordenadas, uno reprsentando la pérdida, y el otro representando la precisión.

Mientras, en la Figura 38, se representa la matriz de confusión de esta red cuando se evalúa con los datos ideales. Lo que la matriz de confusión muestra es que la red acierta prácticamente la totalidad de las actividades, teniendo solamente mayores problemas con la "No actividad", como muestran la primera fila y columna. También se puede observar que en alguna ocasión confunde actividades entre sí, como la "Cintura doblada hacia delante", que la confunde con otras 5 actividades: "Inclinarse tocando el pie con la mano opuesta", "Inclinación lateral", "Inclinación lateral con brazo en alto", "Estiramiento hacia delante repetitivo" y "Elevación frontal de los brazos", habiendo, en la mayoría de ellas, movimiento de la cintura, que sumado a que esta confusión solamente se da una vez con cada actividad frente a las 3800 veces que acierta la actividad, se puede considerar un muy buen rendimiento.

También se muestra figura de la estructura de la red *R(128)-R(128) - S_m* (Figura 39), siguiendo otra vez la notación de (Ordóñez, F. J. y Roggen, D., 2016), es decir, $a_{t,i}^{l}$, indica la activación de la *unit i* en el instante de tiempo *t* en la capa *l*.

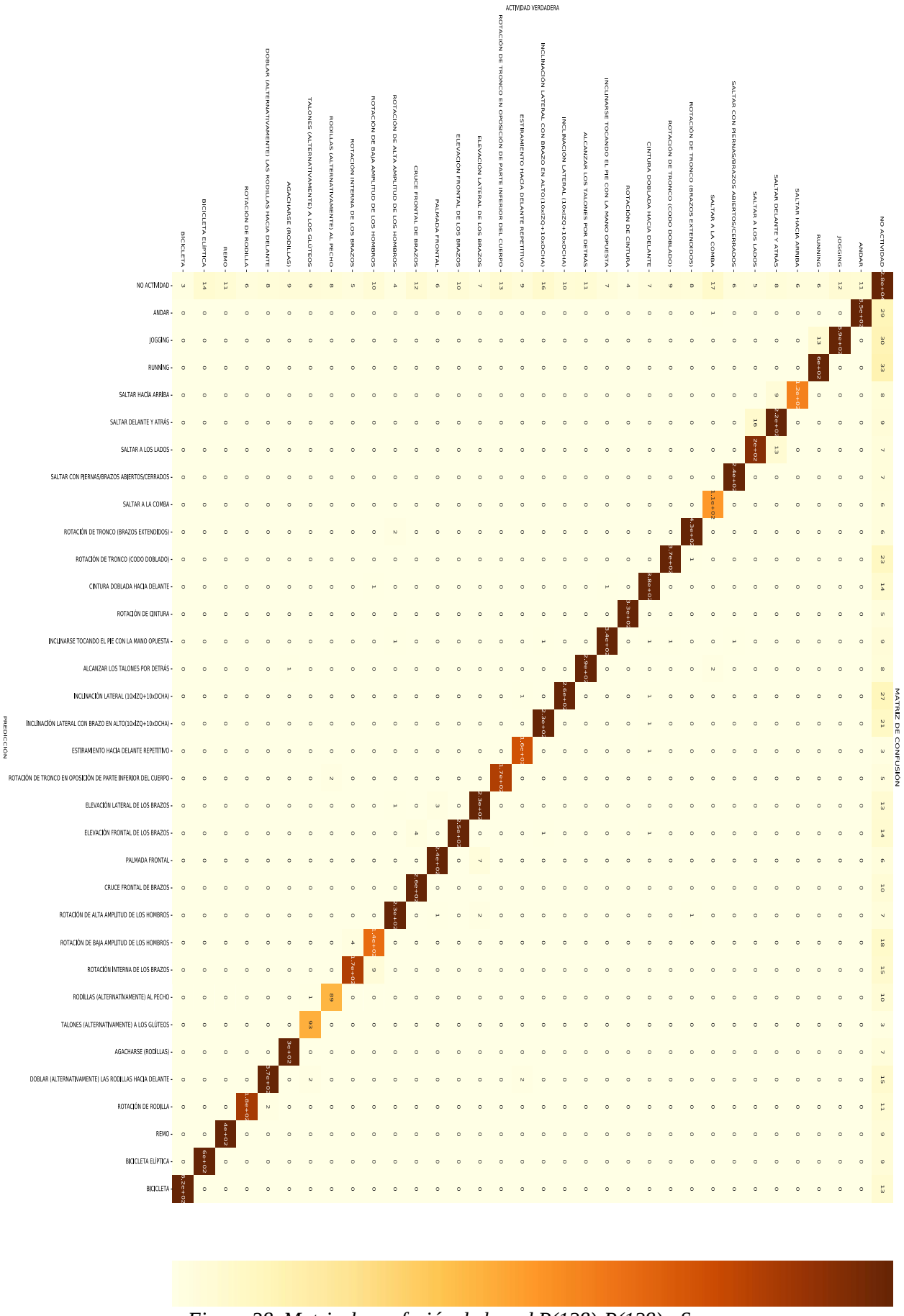


Figura 38. Matriz de confusión de la red R(128)-R(128) - Sm .

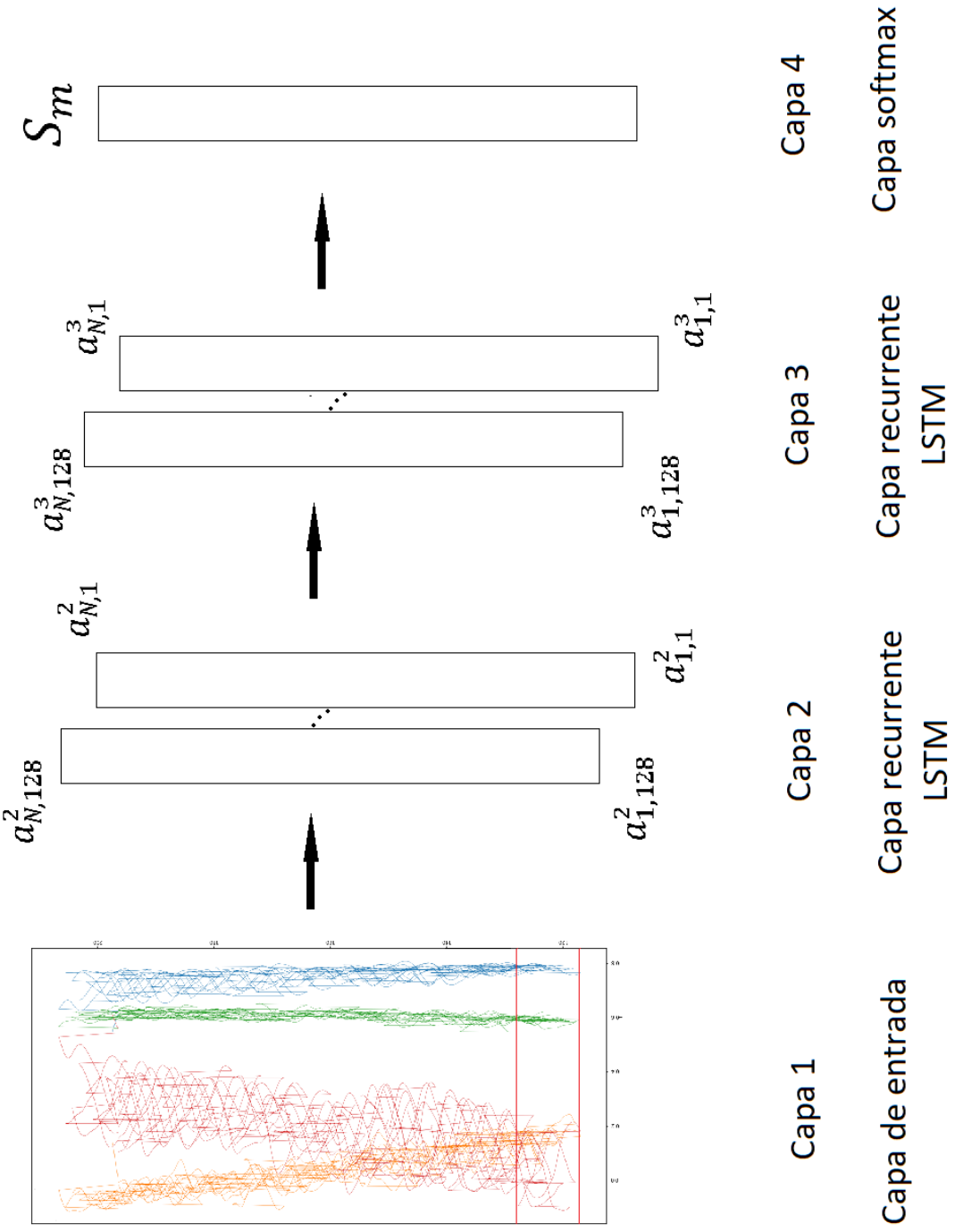


Figura 39. Estructura completa de la red R(128)-R(128) - Sm.

CNN+LSTM

Las redes que combinan capas convolucionales con capas recurrentes son un tipo de redes que se utilizan para el problema HAR, con el objetivo de que las capas convolucionales, previas a las recurrentes, extraigan las *features* de los datos, ya que será capaz de hacer una mejor extracción que las que se pueda realizar "a mano", como se indicó en la sección 3.3 a raíz del trabajo (San *et al.*, 2017, pp. 186-204). Así, la parte del reconocimiento se realiza en las capas recurrentes, pero con las *features* extraídas en las capas convolucionales previas.

En cuanto a los resultados, expuestos en la Tabla 3, se puede ver que, generalmente, son mejores que para las redes completamente recurrentes de la Tabla 1, lo que hace presente el efecto de las capas convolucionales previas antes comentado.

Ya una vez visto el efecto del *overlap* en el rendimiento de las redes en la sección anterior, se decidió aplicar la técnica de la ventana deslizante siempre con *overlap* debido a su notable mejoría. Asimismo, también se tenía una idea del tamaño de segmento que mejor funcionaba, es por ello por lo que se hicieron pocos cambios en este sentido. En cuanto al tamaño de las convoluciones, siempre se ha optado por una convolución 2D de tamaño 4x4 con un stride de 1 para, en los casos en los que tenemos 36 *features* abarcar todas las dimensiones de un cuaternión de una vez, y en el caso de tener 18 *features*, aprovechamos para abarcar las dos dimensiones más significativas de dos cuaterniones. Además, de esta manera se tienen en cuenta las relaciones temporales de cada cuaternión consigo mismo y con los cuaterniones adyacentes.

Con respecto al número de *features* (*n_channels*), se probó a utilizar 36 y posteriormente 18 observando que no había gran diferencia en cuanto a la precisión, pero sí en cuanto a tiempo de entrenamiento, siendo considerablemente menor en el caso de tener 18 *featutres*.

Además, se estudió la infuencia del tamaño del *batch*, determinando como mejor opción un tamaño de *batch* de 256 (como se puede observar en la tabla, el hecho de "alimentar" a la red con demasiados segmentos cada vez puede ir en nuestra contra, teniendo que encontrar el tamaño con el que mejor trabaje).

También, como interés, se probó a entrenar alguna red con datos ideales mezclados ocn datos de tipo *self* obteniendo muy buenos resultados.

Finalmente, se puede ver un gran resultado con una red de 3 capas convolucionales y una recurrente, de un 98,3832 % de precisión, siendo esta misma red también la mejor opción para un entrenamiento conjunto de datos del tipo ideal y del tipo *self*, alcanzando una precisión en test de 98,4760 %.

Arquitectura	*sliding window*	*n_time_steps*	*n_channels*	*batch_size*	*learning_rate*	Precisión (%)
C(16)-C(32)-R(128)-R(64)- S_m	Sí	128 (2,56 s)	36	1024	10^{-4}	95,3841
C(16)-C(32)-R(108)-R(108)- S_m	Sí	128 (2,56 s)	36	1024	10^{-4}	95,9440
C(16)-C(32)-R(108)- S_m	Sí	128 (2,56 s)	36	1024	10^{-5}	79,4922
C(64)-C(128)-R(128)-R(128)- S_m	Sí	128 (2,56 s)	36	512	10^{-4}	85,1172
C(64)-C(128)-R(128)-R(128)- S_m	Sí	128 (2,56 s)	36	256	10^{-4}	97,5391
C(128)-C(256)-R(128)-R(128)- S_m	Sí	128 (2,56 s)	36	256	10^{-4}	97,7539
C(16)-C(64)-C(128)-R(128)-R(128)- S_m *(ideal+self)*	Sí	144 (2,88 s)	18	256	10^{-4}	97,9632
C(16)-C(64)-C(128)-R(256)-R(64)- S_m *(ideal+self)*	Sí	144 (2,88 s)	18	256	10^{-4}	**98,4760**
C(16)-C(64)-C(128)-R(256)-R(64)- S_m	Sí	144 (2,88 s)	18	256	10^{-4}	**98,3832**
C(16)-C(64)-R(128) - S_m	Sí	144 (2,88 s)	18	256	10^{-4}	98,2
C(16)-C(64)-R(128)-R(128)	Sí	144 (2,88 s)	18	256	10^{-4}	98,2357
C(16)-C(64)-R(256)-R(64) - S_m	Sí	144 (2,88 s)	18	256	10^{-4}	98,0520
C(16)-C(64)-C(128)-RG(128)-RG(128)- S_m	Sí	144 (2,88 s)	18	256	10^{-4}	95,98

Tabla 3. Resultados de las redes convolucionales-LSTM diseñadas.

Por otro lado, revisando la Tabla 4, se ve un rendimiento mucho mayor frente a datos de otros tipos que en las redes recurrentes, especialmente notable cuando se evalúa solamente un sujeto de tipo *mutual 7*, en el que casi aumenta un 20%.

En esta ocasión se quiso comprobar de nuevo la robustez de las *units* de tipo GRU, motivado por lo visto en las redes recurrentes. Por ello, se entrenó una red exactamente igual a la red con la que se consiguió un mejor resultado, pero cambiando las *units* GRU por *units* LSTM. Con esta red se obtuvo un 95,98 % de precisión con los datos ideales, sin embargo, no se obtuvieron mejores resultados que para la red con *units* LSTM en el caso de evaluar con los otros tipos de datos. De hecho, se obtienen resultados similares, a excepción del caso de un sujeto *mutual 7*, lo que nos lleva a pensar que, si se consiguiera un 98 % en datos ideales, sí se podría superar el rendimiento para los otros tipos de datos.

Por ello se probó…

Arquitectura	Entrenamiento	Precisión (%)				
		Ideal	*Self*		*Mutual*	
			Todos los sujetos	*1 sujeto*	*Todos los sujetos*	*1 sujeto (mutual 7)*
C(16)-C(64)-C(128)-R(256)-R(64)- S_m (ideal+self)	Ideal+self	**98,4760**			70,8171	69,4010
C(16)-C(64)-C(128)-R(256)-R(64)- S_m	Ideal	**98,3832**	81,4656	77,0103	70,0833	68,1207
C(16)-C(64)-C(128)-RG(128)-RG(128)- S_m	Ideal	95,98	79,0599	77,8821	70,0675	51,3021

Tabla 4. Resultados de las redes convolucionales-recurrentes diseñadas con otros datos distintos de los de entrenamiento.

En este caso, también se muestra el proceso de entrenamiento (Figura 40) y la matriz de confusión (Figura 41) para los datos ideales de la red que mejor precisión ha dado con dichos datos., que ha sido la *C(16)-C(64)-C(128)-R(256)-R(64)-* S_m, con un 98,3832% de precisión.

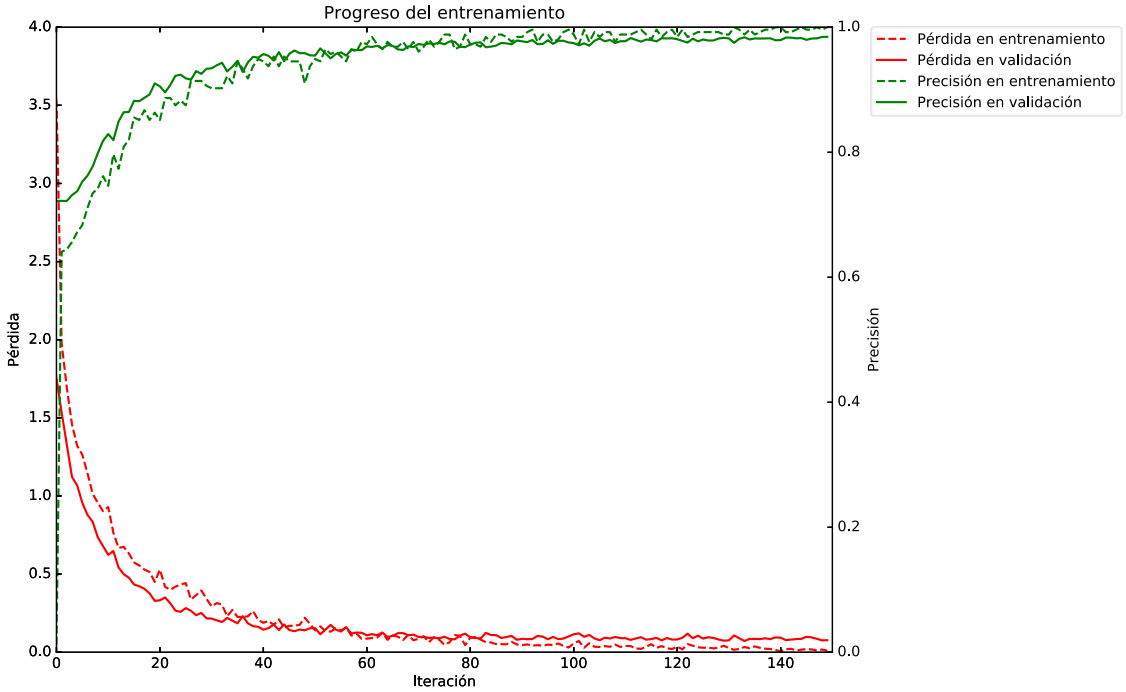


Figura 40. Progreso de entrenamiento de la red C(16)-C(64)-C(128)-R(256)-R(64)- Sm.

Como se puede observar, el proceso de entrenamiento es similar al de la red recurrente. Solamente cabe destacar que a partir de la iteración 80, las curvas de entrenamiento y validación se cruzan, indicando que podría haber un ligero *underfitting* (proceso contrario al *overfitting* antes explicado), algo que se podría haber arreglado aplicando el *early stopping*. Sin embargo, este efecto es mínimo, por lo que no se tiene en consideración.

En cuanto a la matriz de confusión, ocurre lo mismo que en las redes recurrentes, la "No actividad" genera problemas, pero comparado con la cantidad de datos evaluados, estos erorres sin pequeños, generando una precisión tan alta. De los errores no relacionados con la "No actividad", caben destacar las 11 ocasiones que confunde "Saltar hacia arriba" con "Saltar hacia deante y atrás", ambas actividades pueden considerarse semejantes y 11 ocasiones frente a las otras 2200 que no las confunde puede considerar un buen resultado.

Finalmente, se muestra una figura de la estructura de la red la *C(16)-C(64)-C(128)-R(256)-R(64)-* S_m (Figura 42). La notación para las capas recurrentes es como la indicada en el apartado anterior, mientras que para las capas convolucionales tenemos a_i^l, que indica la activación que define el *feature map i* en la capa *l*.

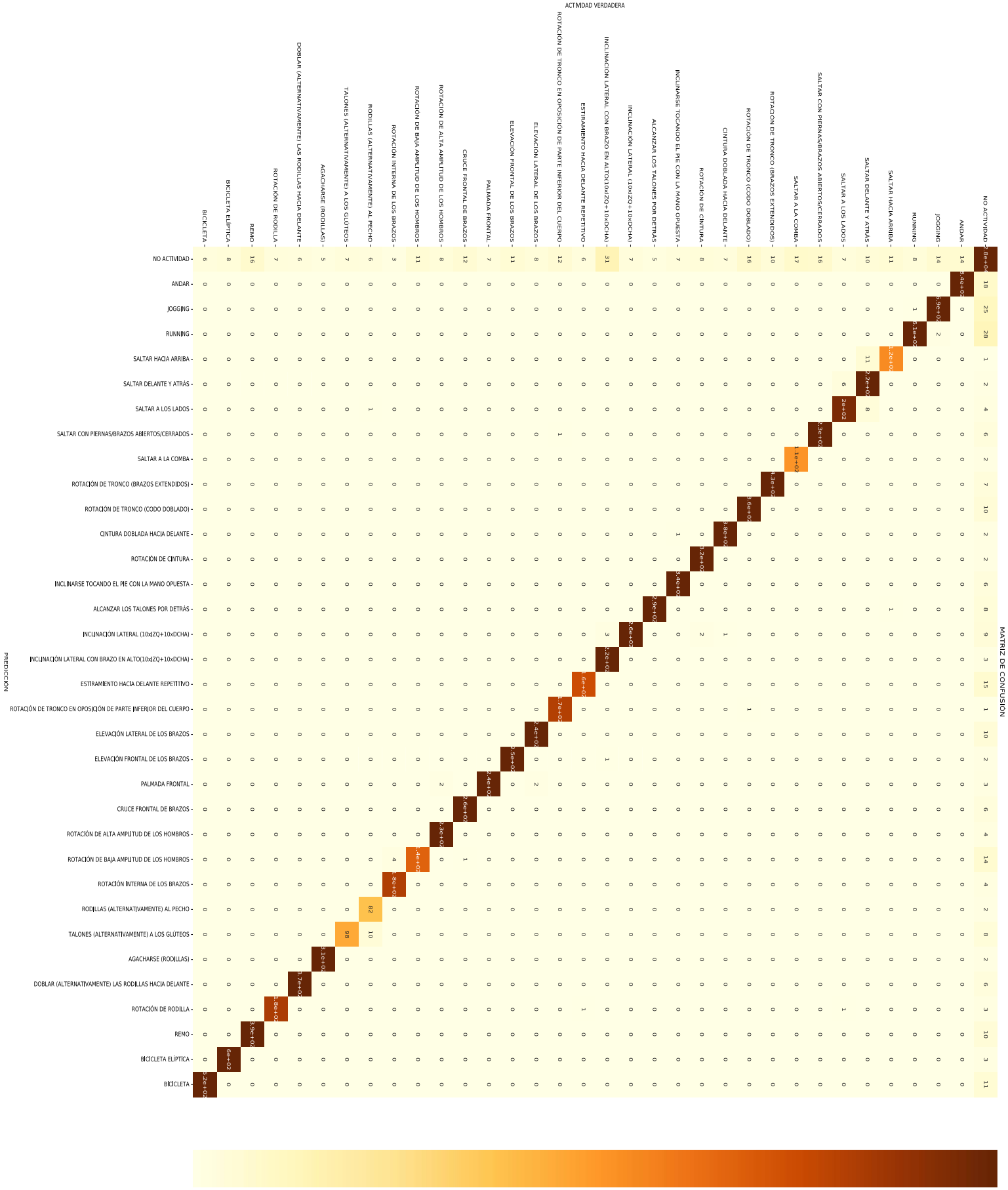


Figura 41. Matriz de confusión de la red C(16)-C(64)-C(128)-R(256)-R(64)- Sm.

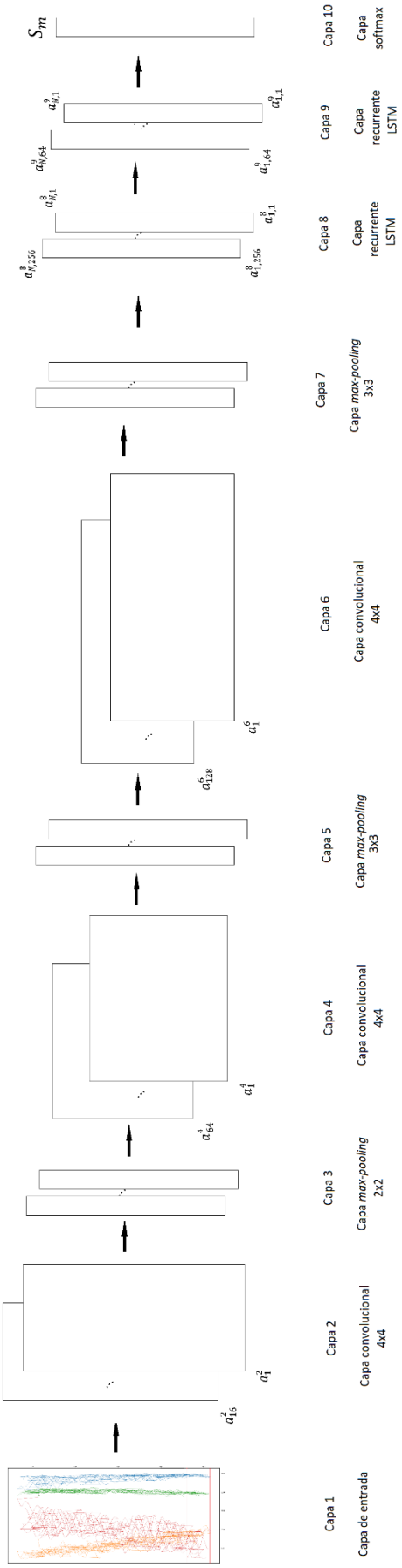


Figura 42. Estructura completa de la red C(16)-C(64)-C(128)-R(256)-R(64)- Sm.

CNN

Las redes convolucionales son ampliamente utilizadas en los campos de *Computer Vision* y *Speech Recognition*, pero también en la resolución del problema HAR.

Además, es el tipo de red que se utiliza en el trabajo de (San *et al.*, 2017, pp. 186-204), explicado antes. Es por ello por lo que se va a utilizar este trabajo para comparar las distintas redes convolucionales realizadas, centrándonos en que la red convolucional del trabajo utiliza 3 capas convolucionales (1x5, 1x5 y 1x3) separadas pr 2 capas de "submuestreo" 1x2, que corresponde con las capas de *max-pooling*, entrenando con los datos ideales y evaluando con los datos tipo *self*, de tal forma que obtiene un 90,1 % de precisión.

Por lo que el objetivo es alcanzar ese 90,1 % en datos *self*. Para ello, se realizaron una serie de pruebas, mostradas en la Tabla 5, con distintos tamaños de ventana (desde 1 segundo hasta 4 segundos de longitud), con distinta cantidad de *features* y distintos tamaños de *batch*. Además, se probó a normalizar con respecto al máximo y al mínimo los datos, obteniendo un resultado similar al caso en el que se estandarizan, pero con un tiempo de entrenamiento del doble, lo que refuerza la elección de la estandarización de los datos.

Motivado por el trabajo de (Zhu, J. *et al*., 2017), se realizó una prueba en la que los datos de los sujetos no estaban mezclados entre los sets de entrenamiento, validación y test como en todo el resto de las pruebas (*random-partitioning*), sino que todos los datos de un sujeto estaban únicamente en un set, por lo tanto, se utilizaron 8 sujetos en el set de entrenamiento, otros 4 sujetos diferentes en el de validación y otros 5 para el set de test. De esta forma nos aseguramos de que la red no está aprendiendo información relacionada con el usuario, es decir, que, a la hora de hacer el test, no va a poder reconocer que una actividad la está realizando un sujeto que ya conoce ni va a poder ayudarse de ese hecho para realizar la predicción de la actividad, porque se le estará realizando el test con un sujeto que no conoce. Sin embargo, esta red tuvo un rendimiento de un 80,2370 % de precisión, lejos del más del 95 % de precisión alcanzado por la mayoría del resto de redes convolucionales entrenadas.

Finalmente, cabe destacar el 98,7221 % de precisión alcanzado por una red de solamente dos capas convolucionales con sus respectivas capas de *max-pooling*.

Arquitectura	*sliding window*	*n_time_steps*	*n_channels*	*batch_size*	*learning_rate*	Precisión (%)
C(16)-C(64)- S_m	Sí	200 (4 s)	36	512	10^{-4}	98,1510
C(16)-C(64)- S_m (norm, tarda el doble en entrenar)	Sí	200 (4 s)	36	512	10^{-4}	98,4798
C(2)-C(16)-C(64)- S_m (por sujetos)	Sí	144 (2,88 s)	18	256	10^{-4}	**80,2370**
C(16)-C(64)- S_m	Sí	144 (2,88 s)	18	512	10^{-4}	96,6927
C(16)-C(64)-C(128)- S_m	Sí	144 (2,88 s)	18	256	10^{-4}	98,3728
C(16)-C(64)-C(128)- S_m (ideal+self)	Sí	54 (1,08 s)	18	75	10^{-4}	98,6402
C(16)-C(64)-C(128)- S_m (ideal+self)	Sí	144 (2,88 s)	18	256	10^{-4}	**98,7976**
C(16)-C(64)- S_m	Sí	200 (4 s)	36	512	10^{-4}	98,6328
C(16)-C(32)- S_m	Sí	128 (2,56 s)	36	1024	10^{-4}	97,9753
C(16)-C(32)- S_m (B)	Sí	128 (2,56 s)	36	1024	10^{-4}	**97,9363**
C(16)-C(32)- S_m (B)	Sí	128 (2,56 s)	36	600	10^{-4}	82,6707
C(16)-C(64)- S_m	Sí	144 (2,88 s)	18	256	10^{-4}	**98,7221**
C(16)-C(64)-C(128)- S_m	Sí	144 (2,88 s)	18	128	10^{-4}	98,6781

Tabla 5. Resultados de las redes convolucionales diseñadas.

Y para poder comparar las redes convolucionales realizadas con la del trabajo de (Zhu, J. *et al*., 2017), se evaluaron las redes más interesantes con los datos tipo *self* y *mutual*, obteniendo los datos mostrados en la Tabla 6.

Como se puede observar, al igual que en el caso de datos ideales, en cuanto a precisión, el resultado de la red entrenada con los datos estandarizados frente a la que tiene los datos normalizados no difiere demasiado, pudiéndolas considerar semejantes salvo por el hecho del tiempo de entrenamiento comentado anteriormente.

Un detalle que cabe destacar es la buena realización de la red entrenada por sujetos en los datos *self* y *mutual*, alcanzando prácticamente la misma precisión para los datos *self* que para los datos ideales. Aunque en este resultado están influyendo positivamente los datos de los 8 sujetos con los que se entrenó a la red, se puede considerar un buen resultado.

Finalmente, con la red con la que se obtenía un 98,7221 % de precisión, siendo la más elevada para datos ideales, se obtiene un 82,4891 % de precisión para todo el set de datos *self*, siendo otra vez la red con más precisión en estas condiciones, quedándose a un 7,5% de alcanzar la máxima precisión alcanzada con una red convolucional con los datos del dataset REALDISP encontrada durante la revisión bibliográfica del Estado del Arte.

Arquitectura	Entrenamiento	Precisión (%)				
		Ideal	*Self*		*Mutual*	
			Todos los sujetos	*1 sujeto*	*Todos los sujetos*	*1 sujeto (mutual 7)*
C(16)-C(64)- S_m	Ideal	98,1510	80,4458	86,1328	69,9983	58,7891
C(16)-C(64)- S_m (norm, tarda el doble en entrenar)	Ideal	98,4798	81,1322	81,8115	70,1115	50,7812
C(2)-C(16)-C(64)- S_m (por sujetos)	Ideal	**80,2370**	76,4738	78,025	70,0675	51,02
C(16)-C(64)-C(128)- S_m (ideal+self)	Ideal+self	**98,7976**			70,0675	56,4019
C(16)-C(64)- S_m	Ideal	**98,7221**	82,4891	77,7391	70,3317	57,2917
C(16)-C(64)-C(128)-S_m	Ideal	98,6781	82,1964	78,0066	70,0653	55,2576

Tabla 6. Resultados de las redes convolucionales diseñadas con datos de distinto tipo de los de entrenamiento.

Al igual que en los dos tipos de red anteriores, se quiere mostrar, para la red que mayor precisión ha dado, la *C(16)-C(64)- S_m*, con un 98,7221%, el proceso de entrenamiento, mostrado en la Figura 43.

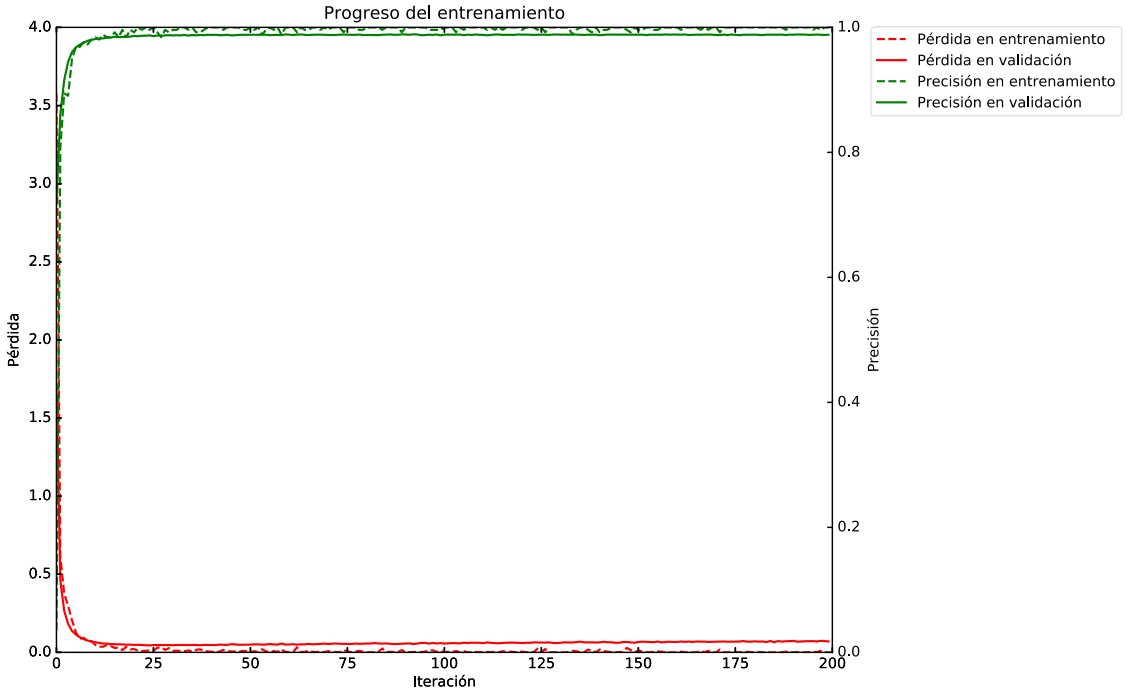


Figura 43. Progreso de entrenamiento de la red C(16)-C(64)- Sm.

Al igual que en el caso de la CNN+LSTM, parece que se produce un ligero *underfitting* antes de la iteración 25. Por la misma razón que antes, no se tiene en consdieración.

Además, se incluye también la matriz de confusión del test con los datos ideales (Figura 44). En la matriz se puede ver que otra vez la "No actividad" es la que más problemas da en la predicción de todas las actividades (primera fila y primera columna), sin embargo, en el resto de la matriz no hay ningún fallo en la predicción, solamente en 5 ocasiones, que confunde las actividades "Saltar hacia delante y atrás" y "Saltar a los lados" entre sí. Este error se puede despreciar teniendo en cuenta la semejanza de las actividades y que en otras 4300 ocasiones la red determina bien ambas actividades.

Y para terminar, se muestra una figura de la estructura de la red la *C(16)-C(64)- S_m* (Figura 45), con la notación ya comentada para las capas convolucionales.

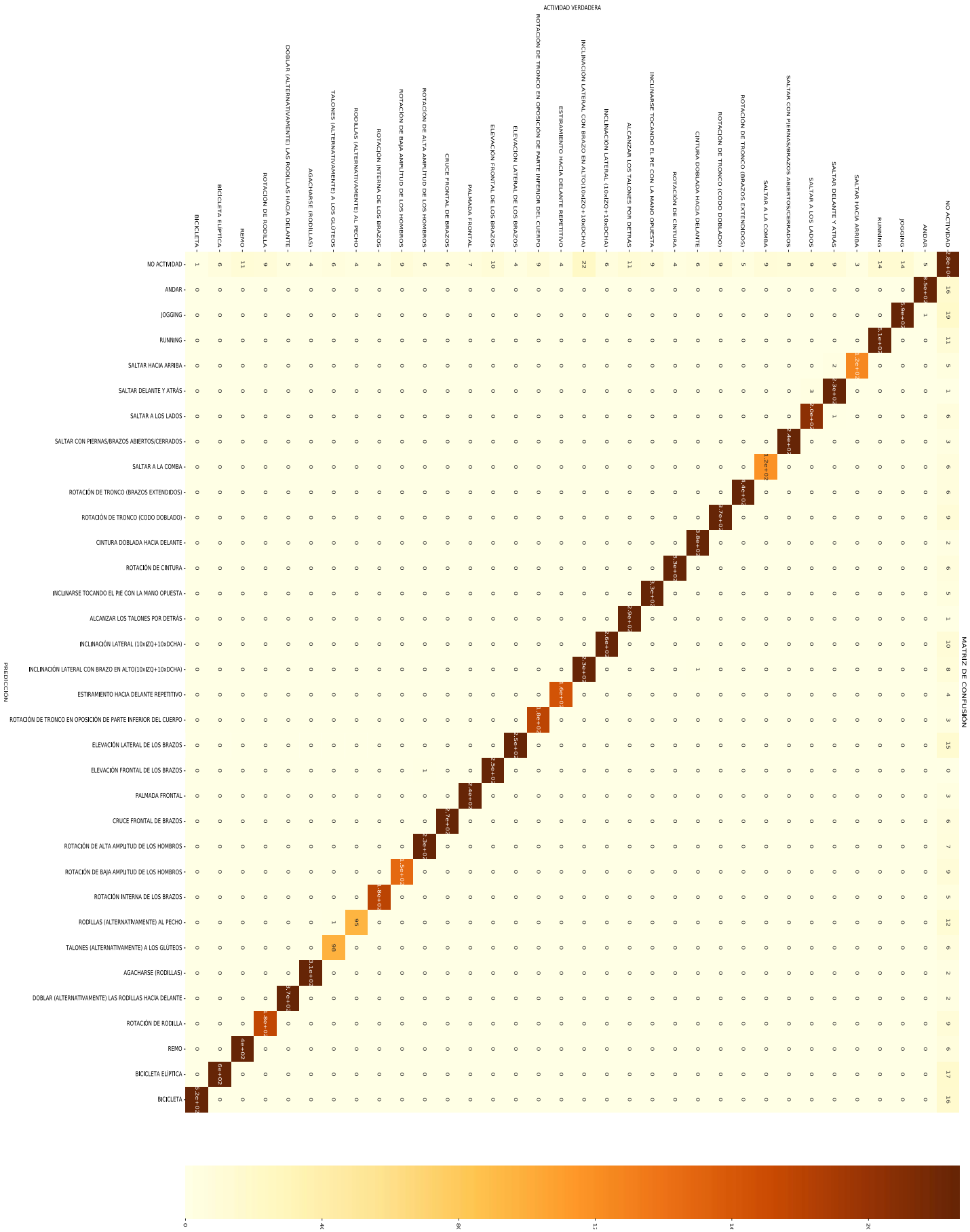


Figura 44. Matriz de confusión de la red C(16)-C(64)- Sm.

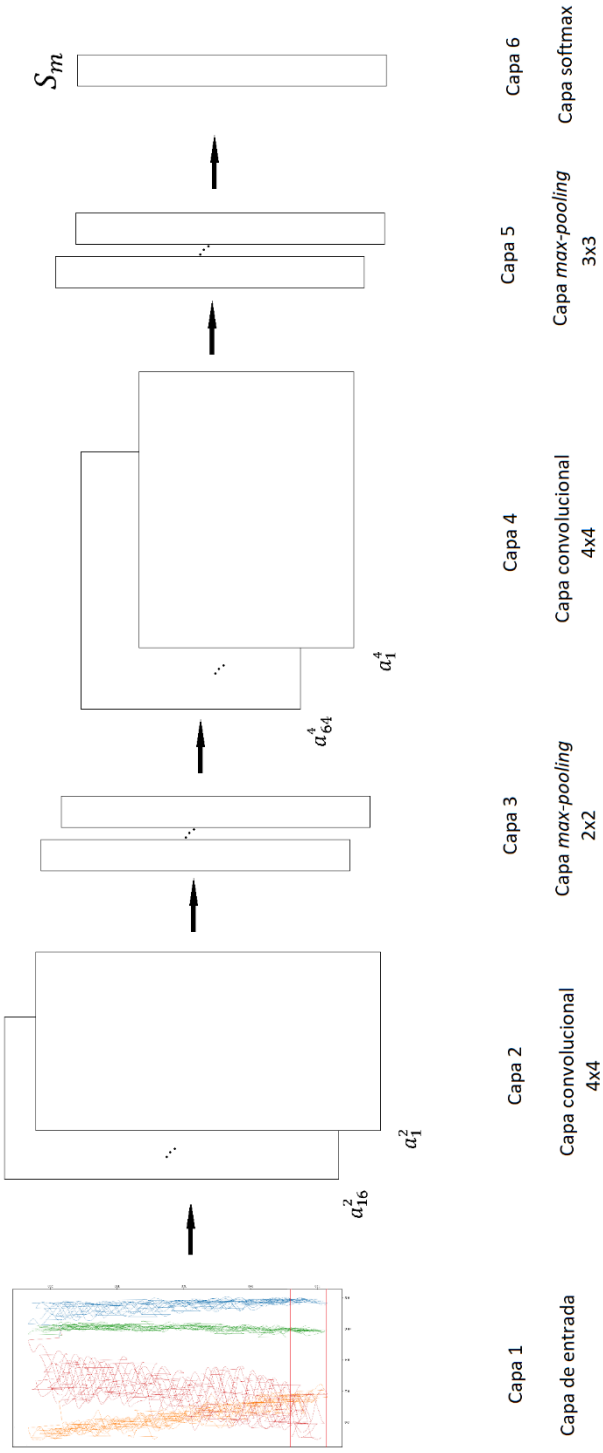


Figura 45. Estructura completa de la red C(16)-C(64)- Sm.

4.3 Presupuesto

Para terminar este capítulo, se quiere mostrar la inversión necesaria para la realización de un estudio de estas características. Esta inversión se puede dividir en gastos de equipamiento y gastos en personal.

En las siguientes tablas se desglosa y detalla la inversión necesaria. El gasto en personal (Tabla 7) se calcula a partir del tiempo empleado en el desarrollo del estudio, sin incluir la etapa de aprendizaje de la teoría y del software de desarrollo.

Horas	Precio por hora (€)	Total (€)
390	15	5850

Tabla 7. Presupuesto de ingeniero.

Equipamiento	€/Unidad	Nº de unidades	Subtotal (€)
Portátil amortizado	780	1	**780**
GASTOS PERSONAL			**5850**
GASTOS SERVIDOR (amortizado)			**7225,16**
Procesador Intel® Xeon® X5650	854,12	2	1708,24
Tarjeta gráfica GeForce GTX 1080 Ti	977,92	1	977,92
Tarjeta gráfica GeForce GTX 970	339	1	339
Placa base TYAN S 7025	700	1	700
Mantenimiento, reparaciones y otros gastos	-	-	3500
IMPORTE TOTAL			**13855,16**

Tabla 8. Presupuesto total.

En la Tabla 8, se han utilizado precios de venta actuales, sin embargo, no todos los componentes han sido adquiridos recientemente, por lo que el precio real podría ser ligeramente mayor. Además, se ha añadido un sobrecoste de mantenimiento, reparaciones y otros gastos que contiene la estimación de los costes de los dispositivos del servidor de los que no se ha podido obtener el precio junto con los costes que supone su mantenimiento y el reemplazo de algunos componentes a lo largo de los años. De esta forma, el gasto en el servidor se estima alrededor de unos 7500€.

Capítulo 5 Conclusiones y líneas futuras

Poder desarrollar distintas redes neuronales de aprendizaje profundo para el Reconocimiento de la Actividad Humana supone un trabajo de gran interés, puesto que tanto la Inteligencia Artificial como el Deep Learning son los campos que se están desarrollando con más notoriedad en los últimos años dentro de la Ingeniería, proporcionando una serie de grandes avances que no hacen más que fomentar aún más la investigación de estos campos.

5.1 Conclusiones

El principal objetivo de este Trabajo Fin de Grado era explorar mediante redes de aprendizaje profundo el reconocimiento de actividades físicas de un dataset de movimientos capturado con sensores inerciales.

Este objetivo se ha cumplido a lo largo de las fases marcadas al inicio de este trabajo, ya que se ha conseguido un resultado cercano al resultado más elevado encontrado para el dataset elegido.

El estudio desarrollado ha demostrado el potencial del Deep Learning para la resolución del problema HAR y su posibilidad de aplicación como complemento del desarrollo de un juego serio para la rehabilitación de pacientes, ayudando a determinar si los ejercicios marcados para la recuperación física están siendo correctamente realizados por el paciente o no.

La **primera conclusión** a la que se ha llegado, tras realizar un estudio del problema del reconocimiento de actividades físicas, es que este problema está ya suficientemente estudiado mediante otras técnicas y algoritmos de Machine Learning, con tasas realmente elevadas, pero con un preprocesado de los datos muy profundo y una programación subyacente muy tediosa. El Deep Learning ha irrumpido con fuerza en este problema y son cada vez más las redes neuronales diseñadas para su resolución, permitiendo obtener buenos resultados de manera más sencilla que con otros algoritmos, sin necesidad de un preprocesado de los datos tan exhaustivo, debido al propio concepto de la red neuronal, aunque también hay que recordar que se deben ajustar los parámetros de la red para un rendimiento mejor.

La **segunda conclusión** es que no solamente hay un tipo de red neuronal posible o adecuada para la resolución del problema. En este trabajo se han visto 3 tipos con distintas configuraciones, viendo que, en ocasiones, dos redes pueden ser semejantes en el caso de los datos ideales, pero diferir en el caso de los datos *self*, hecho que puede interesar si el

paciente realiza siempre los ejercicios en su domicilio y se pone él mismo los sensores, pudiendo colocarlos mal en alguna ocasión.

La **conclusión final** a la que se ha llegado es que se ha realizado un amplio estudio de cómo resolver el problema HAR, que ha permitido ver un alto potencial en las redes convolucionales, tanto por su rapidez a la hora de entrenar en comparación con los otros dos tipos, como por los resultados que ha dado, siendo con este tipo de redes con las que se han obtenido precisiones más altas para todos los tipos de datos, **cercanas** a las obtenidas por **otros trabajos** sobre este mismo dataset aplicadno técnicas de **Deep Learning**.

5.2 Líneas futuras

Tras el cumplimiento del objetivo principal de este Trabajo Fin de Grado, queda la posibilidad de afrontar varias líneas de futuro, de cara a mejorar los resultados obtenidos

Una de las **posibilidades** sería probar a utilizar todos los datos disponibles en el dataset, es decir, utilizar en el entrenamiento los datos procedentes del acelerómetro, girsocopio y del magnetómetro junto con los cuaterniones ya utilizados. De esta forma, se reforzaría el entrenamiento, principalmente, por el aumento de la cantidad de datos ofreciendo información sobre las actividades a la red.

Otra posible línea futura seria continuar trabajando con las redes recurrentes y las redes que combinan capas convolucionales y capas recurrentes. A pesar de que en este estudio se ha visto que las convolucionales resultaban ser la mejor opción, estos dos tipos de redes también han dado buenos resultados, y podría resultar interesante continuar por esta línea.

Finalmente, quedaría abierta la línea de, a partir de este estudio, plantearse la realización de un **dataset propio**, con las actividades específicas que se deseen, para después desarrollar una red que rinda bien con esos datos y, por último, implementar esa red de reconocimiento de actividades físicas en un **juego serio** diseñado para la rehabilitación.

REFERENCIAS

Anguita, D., Ghio, A., Onteo, L., Parra, X. y Reyes-Ortiz, L. (2013) "A Public Domain Dataset for Human Activity Recognition Using Smartphones", *European Symposium on Artifical Neural Networks, Computational Intelligenes and Machine Learning*. Brujas (Bélgica), 24-26 abril 2013.

Baños, O., Damas, M., Pomares, H., Rojas, I., Tóth, M. A. y Amft, O. (2012a) *A benchmark dataset to evaluate sensor displacement in activity recognition*.

Baños, O., Calatroni, A, Damas, M., Pomares, H., Rojas, I., Sagha, H., del R. Millán, J., Tröster, G., Chavarriaga, R. y Roggen, D. (2012b) "Kinect=IMU? Learning MIMO Signal Mappings to Automatically Translate Activity Recognition Systems Across Sensor Modalities", *16th International Symposium on Wearable Computers*. Newcastle, 18-22 junio 2012. Disponible en: https://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6242775 [Consultado 7-6-2018]

Baños, O. y Tóth, M. A. (2014a) *Realistic sensor displacement benchmark dataset*. Manual del dataset.

Baños, O., Toth, M. A., Damas, M.m Pomares, H. y Rojas, I. (2014b) "Dealing with the Effects of Sensor Displacement in Wearable Activity Recognition", *Sensors*, 14, pp. 9995-10023.

Bengio, Y. (2009) *Learning Deep architectures for AI*. Found Trends Mach Learn

Buck, I. (2010) "The Evolution of GPUs for General Purpose Computing", *GPU Technology Conference*. San Jose, CA, 20-23 septiembre 2010.

Bulling, A., Blanke, U., Schiele, B. (2014) *A tutorial on human activity recognition using body-worn inertial sensors*.ACM Comput Surv

Center for Machine Learning and intelligent Systems (n.d.) *UCI Machine Learning Repository*. Disponible en: https://archive.ics.uci.edu/ml/datasets/REALDISP+Activity+Recognition+Dataset [Consultado 8-3-2018]

Chauvin, Y. y Rumelhart, D. E. (1995) *Backpropagation: Theory, architectures, and applications*. New Jersey: Lawrence Erlbaum Associates, Publishers.

Cho, K., Gulcehre, B. C., Bahdanu, D., Bougares, F., Schwenk, H. y Bengio, Y. (2014) *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*.

Cortes, C. y Vapnik, V. (1995) *Support-vector networks. Machine Learning*.

Docker (2018) *What is Docker?*. Disponible en: https://www.docker.com/what-docker [Consultado 18-4-2018]

Duchi, J., Hazan, E. y Singer, Y. (2011) "Adaptive subgradient for online learning and stochastic optimization", *The Journal of Machine Learning Research*, 12, pp. 2121-2159.

Fukushima, K. y Miyake, S. (1982) *Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Visual Pattern Recognition*. Competition and Cooperation in neural nets: Springer.

González Alonso, J. (2017) *Periféricos basados en Arduino para interacción con sistemas médicos de simulación y rehabilitación*. Trabajo Fin de Grado. Universidad de Valladolid.

Goodfellow, I., Bengio, Y. y Courville A. (2016) *Deep Learning*. Cambridge, MA (EEUU):MIT Press.

Hassan, M. M., Huda, S., Uddin, M., Almogren, A., Alrubaian, M. (2018) "Human Activity Recognition from Body Sensor Data using Deep Lerning", *Journal of Medical Systems*.

Hochreiter, S. y Schmidhuber, J. (1997) "Long short-term memory". En: *Neural Computation*. MIT Press.

Intel Corporation (n.d.) Intel. Disponible en: https://ark.intel.com/es-es/products/47922/Intel-Xeon-Processor-X5650-12M-Cache-2_66-GHz-6_40-GTs-Intel-QPI [Consultado 10-6-2018]

Ivakhnenko, A. G. y Ivakhnenko, G. A. (1995) "The Review of Problems Solvable by Algorithms of the Group Method of Data Handling (GMDH)". En: *Patern Recognition and Image Analysis*.

Jupyter (2018) Jupyter. Diponible en: http://jupyter.org/ [Consultado 18-4-2018].

Kingma D. P. y Lei Ba, M. (2015) "Adam: A method for stochastic optimization". ICLR. San Diego, 7-9 mayo 2015.

Krizhevsky, I., Sutskever, I. y Hinton. G. E. (2012) *ImageNet Classification with Deep Convolutional Neural Networks*

LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W. y Jackerl, L. D. (1989) "Backpropagation Applied to Handwritten Zip Code Recognition". En: *Neural Computation*, pp. 541-551.

LeCun, Y., Bengio, Y. y Hinton, G. (2015) *Deep Learning*. Nature Publishing Group.

Matich, D. J. (2001) *Redes Neuronales. Conceptos básicos y aplicaciones*. Cátedra. Universidad Tecnológica Nacional – Facultad Regional Rosario.

McCulloch, W., S. y Pitts, W. (1943). "A logical calculus of the ideas immanent in nervous activity". *The bulletin of mathematical biopshisycs*.

NVIDIA Corporation (2014) Nvidia GeForce GTX 970 *datasheet*.

NVIDIA Corporation (2017) Nvidia GeForce GTX 1080 Ti *datasheet*.

NVIDIA Corporation (2018a) *Digits*. Disponible en: https://developer.nvidia.com/digits [Consultado 18-4-2018]

NVIDIA Corporation (2018b) *CUDA Toolkit Documentation*. Disponible en: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html#memory-optimizations [Consultado 22-6-2018]

Ordóñez, F. J. y Roggen, D. (2016) *Deep Convolutional ans LSTM Recurrent Neural Networks for Multimodal Wearable Activity Recognition*.

Python (2018) Python. Disponible en: https://www.python.org/ [Consultado 18-4-2018].

Raví, D., Wong, C., Lo, B. y Yang, G. (2017) "A Deep Learning Approach to on-Node Sensor Data Analytics for Mobile or Wearable Devices", *IEEE Journal of biomedical and health informatics*, 21 (1), pp. 56-64.

Rosenblatt, F. (1958), *The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain*. Cornell Aeronautical Laboratory, Psychological Review, v65, No.

Sak, H., Senior, A. y Beaufays, F. (2014) *Long short-term memory based Recurrent Neural Network Architectures for large vocabulary speech recognition*.

6, pp. 386-408

San, P. P., Kakar, P., Li, X., Krishnaswamy, S., Yang, J., Nguyen, M. N. (2017) "Deep Learning for Human Activity Recognition". En: Hsu, Hui-Huang *et al.* Eds. *Big Data Analytics for Sensor-Network Collected Intelligence*. Singapur: *Elsevier Inc.*

Tensorflow (2018) Tensorflow. Disponible en: https://www.tensorflow.org/ [Consultado 18-4-2018].

Udacity (2018) *Courses*. Disponible en: https://classroom.udacity.com [Consultado 12-3-2018]

Valenti, R. G., Dryanowski, I. y Xiao, J. (2015) "Keeping a Good Attitude: A Quaternions-Based Orientation Filter for IMUs and MARGs", *Sensors*, 15, pp. 19308-19314.

Zaremba, W., Sutskever, I. y Vinyals, O. (2015) *Recurrent Neural Network Regularization*.

Zhang, R y Li, C. (2015) "Motion Sequence Recognition with Multi-sensors Using Deep Convolutional Neural Network", Second Euro-China Conference, ECC. Ostrava, República Checa, , 29 junio- 1 julio 2015. Suiza: Springer International Publishing Switzerland, pp. 13-23.

Zhu, J, San-Segundo, D. y Pardo, J. M. (2017) “Feature extraction for robust physical activity recognition”, *Human-centric Computing and Information Sciences*, 7 (16). Disponible en: https://link.springer.com/content/pdf/10.1186%2Fs13673-017-0097-2.pdf [Consultado 20-6-2018]

ANEXOS

ANEXO I

Preparación del espacio de trabajo

Para realizar este Trabajo de Fin de Grado se tuvo que preparar un espacio de trabajo concreto. Este espacio de trabajo consiste en un servidor con las herramientas necesarias para este Trabajo de Fin de Grado, es decir, herramientas necesarias para el desarrollo de trabajos de Deep Learning. A continuación, se especifica qué fue necesario y cómo se realizó dicha preparación para obtener un servidor de Deep Learning:

1. Instalación del Sistema Operativo

Seguimos los pasos de instalación de Ubuntu 16.04.3 LTS.

2. Timeshift

En primer lugar, descargamos e instalamos TimeShift para hacer snaps (puntos de restauración):

```
apt-add-repository -y ppa:teejee2008/ppa
apt-get update
apt-get install timeshift
```

Para restaurar:

```
timeshift --restore
```

Se pondrá en modo interactivo, donde se elegirá el snap que queremos cargar y dónde, además de otras opciones.

Con este programa se pretende tener una copia de seguridad de los ficheros del sistema en caso de algún fallo de este.

3. NAS

El lugar donde guardaremos las copias de seguridad será en una carpeta compartida por NFS en nas.gti.tel.uva.es:

Primero, instalamos portmap y nfs-common:

```
apt-get install portmap nfs-common
```

Reiniciamos el servicio portmap:

```
systemctl restart portmap
```

Ahora ya estaremos en condiciones de montar la carpeta compartida en nuestro sistema de archivos. De esta manera, el acceso a la carpeta compartida es exactamente igual que el acceso a cualquier otra carpeta de nuestro disco duro.

Montamos la carpeta compartida por NFS:

```
mount -t nfs ip-del-servidor:/carpeta-en-servidor /carpeta-en-local
```

Así se guarda en /carpeta-en-local lo que haya en carpeta-en-sevidor.

4. Configuración de la red

El fichero /etc/network/interfaces debe quedar de la siguiente manera:

```
# interfaces(5) file used by ifup(8) and ifdown(8)
auto lo
iface lo inet loopback

auto enp2s0
iface enp2s0 inet dhcp

auto enp3s0
iface enp3s0 inet static
address 157.88.130.167
```

```
netmask 255.255.255.0
broadcast 157.88.130.255
network 157.88.130.0
gateway 157.88.130.250
dns-nameservers 157.88.130.247 157.88.130.90
```

5. SSH

Para poder acceder por SSH deberemos instalar el cliente SSH:

```
apt install ssh
```

6. Instalación de los drivers

Estos han sido los pasos que se han seguido para la instalación de drivers por cambios de tarjeta gráfica:

Descargamos el driver de la página de nvidia y damos permisos de ejecución al archivo:

```
chmod 777 -R NVIDIA-Linux-x86_64-390.25.run
```

Eliminamos todos los drivers anteriores de nvidia que haya:

```
apt-get remove nvidia* && sudo apt autoremove
```

Instalamos algunos paquetes para construir el kernel:

```
apt-get install build-essential linux-headers-$(uname -r)
apt-get install dkms build-essential linux-headers-generic
```

Bloqueamos y deshabilitamos nouveau:

```
apt-get remove --purge xserver-xorg-video-nouveau
nano /etc/modprobe.d/blacklist.conf
```

Añadimos las siguientes líneas al final del fichero /etc/modprobe.d/blacklist.conf:

```
blacklist nouveau
blacklist lbm-nouveau
options nouveau modeset=0
```

```
alias nouveau off
alias lbm-nouveau off
```

Ahora escribimos el siguiente comando:

```
echo options nouveau modeset=0 | sudo tee -a /etc/modprobe.d/nouveau-kms.conf
```

Si nos indica que el fichero nouveau-kms.conf no existe no nos preocupamos de ello, ya que no tiene porqué existir.

Construimos el nuevo kernel:

```
update-initramfs -u
```

Y apagamos:

```
shutdown 0
```

Ahora añadimos la tarjeta dedicada a la placa base, encendemos y continuamos con la instalación. Salimos del entorno gráfico:

```
service lightdm stop
```

Se nos quedará la pantalla en negro y accedemos a la terminal pulsando la siguiente combinación de teclas: CTRL+ALT+F1.

Ahora ya podemos instalar el driver. Hacemos log in y ya podemos seguir. Instalamos el driver con:

```
sh NVIDIA-Linux-x86_64-390.25.run
```

Nota: Ante el error al instalar drivers: "an nvidia kernel module 'nvidia-uvm' appears to already be loaded in your kernel"
Para saber qué está usando nvidia:
lsmod | grep -i nvidia
Suele dar problemas nvidia.uvm, para ver qué lo usa y poder pararlo:
lsof | grep nvidia.uvm

Reinicia el entorno gráfico con:

```
service lightdm start
```

Y hacemos reboot y entramos en la BIOS para cambiar a la tarjeta dedicada.

Nota: Al instalar el driver nos preguntó si queríamos crear el archivo de configuración "/etc/X11/xorg.conf" y le dijimos que sí.

7. Instalación de Docker CE

Podemos instalar Docker CE de diferentes maneras:

- Montando los repositorios de Docker CE e instalarlo desde ahí. Es lo recomendable.
- Descargando el paquete DEB y hacer una instalación manual.
- Usando los scripts convenientes en los que se automatiza la instalación de Docker.

7.1 Instalación usando el repositorio

Primero debemos montar el repositorio de Docker y posteriormente instalar y actualizar Docker desde el repositorio.

REPOSITORIO

Actualizamos el index del paquete apt:

```
apt-get update
```

Instalamos los paquetes para permitir a apt usar el repositorio sobre HTTPS:

```
 apt-get install \
 apt-transport-https \
 ca-certificates \
curl \
software-properties-common
```

Añadimos la clave GPG oficial de Docker:

```
curl  -fsSL  https://download.docker.com/linux/ubuntu/gpg  |
sudo apt-key add -
```

Verificamos que tenemos la clave con la huella 9DC8 5822 9FC7 DD38 854A E2D8 8D81 803C 0EBF CD88, buscando los últimos 8 caracteres de la huella:

```
apt-key fingerprint 0EBFCD88
```

Nos tiene que mostrar:

```
pub   4096R/0EBFCD88 2017-02-22
Key fingerprint = 9DC8 5822 9FC7 DD38 854A  E2D8 8D81 803C 0EBF CD88
uid                  Docker Release (CE deb) <docker@docker.com>
sub   4096R/F273FCD8 2017-02-22
```

Utilizamos el siguiente comando para montar el repositorio stable, que es el que vamos a utilizar:

```
add-apt-repository \
"deb [arch=amd64] https://download.docker.com/linux/ubuntu \
$(lsb_release -cs) \
stable"
```

Actualizamos el index del paquete apt:

```
apt-get update
```

Instalamos la última versión de Docker CE:

```
apt-get install docker-ce
```

Finalmente, verificamos que la instalación de Docker CE se ha hecho correctamente corriendo la imagen de hello-world:

```
docker run hello-world
```

Este comando descarga una imagen de test y la ejecuta en un contenedor, el cual imprime un mensaje informativo y sale.

7.2 Comandos de Docker

En este apartado se muestra una lista con varios de los comandos de Docker y algunos comentarios adicionales:

docker attach *Unirse a un contenedor que está corriendo.*

docker build *Construir una imagen de un Dockerfile.*

docker commit *Crear una imagen desde un contenedor con cambios.*

docker cp *Copia los ficheros/carpetas entre un contenedor y sistema de ficheros local.*

docker create *Crea un nuevo contenedor.*

docker diff *Inspecciona los cambios en el sistema de ficheros de un contenedor.*

docker events *Obtiene, en tiempo real, los eventos del servidor.*

docker exec *Ejecuta un comando en un contenedor activo.*

docker export *Exporta el sistema de ficheros de un contenedor como un archivo tar.*

docker history *Muestra el historial de una imagen.*

docker images *Lista las imágenes.*

docker import *Importa los contenidos de un tar para crear una imagen.*

docker info *Muestra la información del sistema.*

docker inspect *Devuelve información de bajo nivel en un contenedor o imagen.*

docker kill *Mata un contenedor activo.*

docker load *Carga una imagen de un archivo tar o STDIN.*

docker login *Hacer log in en un registro Docker.*

docker logout *Hacer log out de un registro Docker.*

docker logs *Muestra los logs de un contenedor.*

docker network *Administra las redes Docker.*

docker pause *Para todos los procesos en un contenedor.*

docker port *Lista los puertos de un contenedor.*

docker ps *Lista los contenedores activos.*

docker pull *Descarga una imagen o un repositorio de un registro.*

docker push *Sube una imagen o un repositorio a un registro.*

docker rename *Renombra un contenedor.*

docker restart *Reinicia un contenedor.*

docker rm *Elimina uno o más contenedores.*

docker rmi *Elimina una o más imágenes.*

docker run *Ejecuta un comando en un nuevo contenedor.*

docker save *Guarda una o más imágenes a un archivo tar.*

docker search *Busca en el Docker Hub imágenes.*

docker start *Inicia uno o más contenedores parados.*

docker stats *Muestra las estadísticas del uso de recursos de un contenedor.*

docker stop *Para un contenedor activo.*

docker tag *Etiqueta una imagen en un repositorio.*

docker top *Muestra los procesos activos en un contenedor.*

docker unpause *Reactiva todos los procesos (antes parados) de un contenedor.*

docker update *Actualiza la configuración de uno o más contenedores.*

docker version *Muestra la información de la versión de Docker.*

docker volume *Administra los volúmenes Docker.*

docker wait *Bloquea hasta que un contenedor se para, entonces imprime su código de salida.*

docker kill $(docker ps -q) *Para todos los contenedores a la vez.*

docker rm $(docker ps -a -q) *Elimina todos los contenedores a la vez.*

docker rmi $(docker images -q) *Elimina todas las imágenes a la vez.*

docker volume ls -qf dangling=true | xargs -r docker volume rm *Elimina todos los volúmenes.*

Los comandos más utilizados y su forma de uso para la gestión de contenedores en este servidor han sido:

docker pull NOMBRE_IMAGEN *Lo indicado anteriormente.*

dcoker run NOMBRE_IMAGEN *Crea un contenedor de una imagen y lo inicia directamente.*

docker start ID_CONTENEDOR *Imprescindible tener iniciado/activo un contenedor para poder entrar en él después.*

docker exec -it -u 0 ID_CONTENEDOR bash *Para entrar a la VM de un contenedor (también se puede hacer con docker attach, pero en ese caso, los contenedores que tengan como función ejecutar un comando en concreto, como puede ser abrir un Jupyter Notebook no permitirá entrar de esa forma, por eso se utiliza esta, ya que lo permite en todos los casos. Además, la opción -u 0 permite entrar como root en el contenedor).*

docker commit -m "Cambios hechos en la imagen" -a "Nombre autor/autores" \ ID_CONTENEDOR REPOSITORIO/NOMBRE_NUEVA_IMAGEN *(las opciones -m y -a se pueden omitir, crea una imagen de un contenedor).*

docker ps *Lo indicado anteriormente.*

doker ps -a *Lista todos los contenedores que haya creados en el sistema, tanto los activos como los no activos en ese momento.*

docker system df -v *Muestra el nº de imágenes, contenedores y volúmenes, así como su tamaño.*

docker ps -s *Muestra solamente el tamaño de los contenedores.*

docker volume ls *Muestra una lista con los volúmenes creados.*

Se puede salir de un contenedor escribiendo exit, lo cual, para por completo el contenedor, o con CTRL-p o CTRL-q, que te saca del contenedor, pero lo deja corriendo.

8. Jupyterhub

8.1 Instalación de Jupyterhub

En este servidor se creará un servicio basado en Jupyterhub, con el objetivo de poder realizar notebooks basados principalmente en Python y Tensorflow a través de él. Para ello se han seguido los siguientes pasos:

Creamos un directorio: /etc/jupyterhub

Hay que instalar antes de nada pip:

```
apt-get install python-pip
apt-get install python3-pip
```

Y npm:

```
apt-get install build-essential
apt-get install nodejs
```

Finalmente:

```
apt-get install npm (Seguramente se haya instalado al instalar nodejs)
```

Hacer un enlace:

```
ln -s /usr/bin/nodejs /usr/bin/node
```

Instalamos jupyterhub:

```
python3 -m pip install jupyterhub
npm install -g configurable-http-proxy
```

Comprobamos la instalación desplegando la ayuda de jupyterhub y configurable-http-proxy (-h).

Instalamos dockerspawner:

```
pip3 install dockerspawner -> con pip3 para evitar problemas
```

Para que se comunique por el Docker:

```
python3 -m pip install dockerspawner netifaces
```

Generamos un fichero de configuración por defecto de jupyterhub, en el que haremos cambios:

```
jupyterhub --generate-config
```

Instalamos nvidia-docker:

```
curl -s -L https://nvidia.github.io/nvidia-docker/gpgkey | apt-key add -
curl -s -L https://nvidia.github.io/nvidia-docker/ubuntu16.04/amd64/nvidia-docker.list | tee\ /etc/apt/sources.list.d/nvidia-docker.list
apt-get update
apt-get install nvidia-docker
```

Ahora preparamos la imagen Docker a nuestro gusto. Descargamos una imagen con **Tensorflow 1.3** y soporte para **GPU (CUDA 8.0, cuDNN 6.0)**, **PyTorch** y **Jupyter**:

```
git clone https://github.com/aburnap/docker-stacks
```

Vamos al directorio base-notebook de lo que nos hemos descargado y construimos la imagen base de la que obtendremos la que queremos:

```
cd docker-stacks/base-notebook
```

```
docker build -t base-notebook-gpu .--no-cache
```

Este base-notebook hace varias cosas: Primero crea un usuario no-root llamado "UROP_student", también descarga cuDNN 6.0 y CUDA 8.0.

Ahora nos movemos al directorio tensorflow-notebook-gpu y construimos a imagen con tensorflow que parte de la imagen base notebook antes construida:

```
docker build -t tensorflow-notebook-gpu . –no-cache
```

Entramos a un contenedor de esta imagen ya creado y lo modificamos con lo que queremos utilizar para después hacer una imagen de ello:

```
nvidia-docker start ID_CONTENEDOR  (nvidia-docker para que lo inicie con GPU)

docker exec –it –u 0 ID_CONTENEDOR bash  (para entrar al contenedor que tiene por defecto iniciar el notebook directamente, entramos como root)
```

Una vez creado el contenedor de la imagen anteriormente bajada, y ya dentro de él, realizamos los siguientes cambios:

- Actualizamos a Tensorflow (GPU) 1.4.0
- Creamos un environment py27 con Python 2.7 con Tensorflow (GPU) 1.4.0.
- Creamos un environment digits con Digits instalado (para futuros usos).
- Damos permisos al usuario del contenedor.

Hacemos la imagen del contenedor para que lo use el dockerspawner con jupyterhub:

```
docker commit -m "Cambios hechos en la imagen" -a "Nombre autor/autores" ID_CONTENEDOR REPOSITORIO/NOMBRE_NUEVA_IMAGEN  (las opciones -m y -a se pueden omitir)
```

Accedemos a un contenedor de la imagen recién creada y actualizamos todo lo que tengamos que actualizar (esta imagen va con conda):

```
docker exec –u 0 –it

conda update conda
```

8.2 Creación del servicio Jupyterhub

Hacemos el servicio jupyterhub:

```
nano /lib/systemd/system/jupyterhub.service
```

El fichero sería:

```
[Unit]
Description=Jupyterhub Server
After=network.target network-online.target docker.service

[Service]
ExecStart=/usr/local/bin/jupyterhub
WorkingDirectory=/etc/jupyterhub



[Install]
WantedBy=multi-user.target
```

Para activarlo:

```
systemctl daemon-reload
systemctl enable jupyterhub.service (para que se ponga en el boot)
systemctl status jupyterhub.service
```

Hay que tener cuidado al cambiar la configuración del servicio, ya que, si no lo paramos y cambiamos la configuración, fallará y no se podrá iniciar ya hasta que se haga reboot de la máquina.

8.3 Cambiar de entorno en el contenedor desde Jupyter

Esto lo queremos para poder tener dos versiones distintas de Python dentro de un mismo contenedor.

Primero, comprobamos la versión de conda, la cual debe ser al menos una 4.1.0 para poder hacer esto. Si es menor, hacemos conda update conda:

```
conda --version
```

Ahora comprobamos si la librería nb_conda_kernels está instalada con:

```
conda list
```

Si no está instalada, lo instalamos en el entorno root (que es el que tenemos por defecto en el contenedor):

```
conda install nb_conda_kernels
```

Ahora creamos los entornos:

```
conda create -n root Python=3.6 ipykernel
conda create -n py27 Python=2.7 ipykernel
```

Ahora al iniciar jupyter tendremos las siguientes opciones al crear un nuevo notebook:

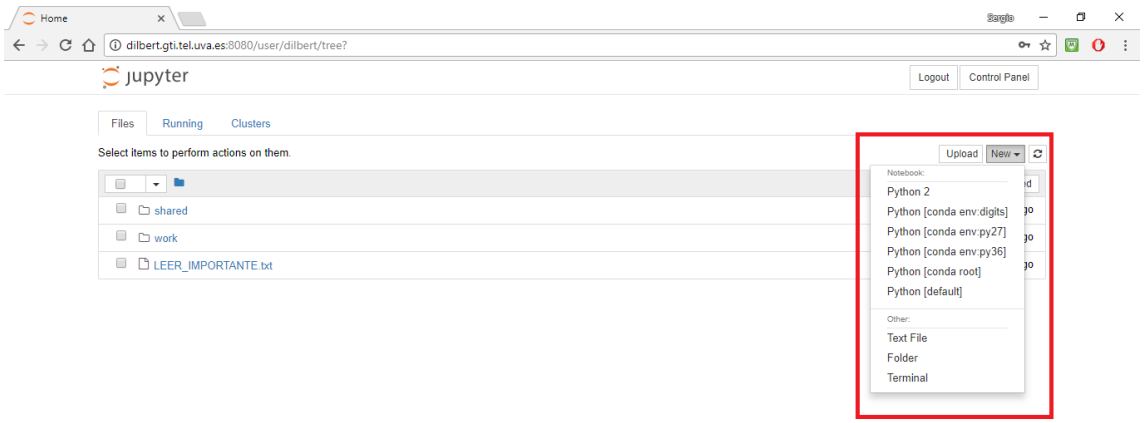


Figura. Ubicación y forma de las opciones de Jupyter.

8.4 Dar permisos a usuario del contenedor para usar conda

El usuario creado por defecto dentro del contenedor no tiene permisos para utilizar conda, por ello se los damos de la siguiente manera, hacemos que a conda le sirva con los permisos del usuario, no de root únicamente:

```
chown -R UROP_student:users /opt/conda
```

8.5. Utilización de volúmenes Docker con Jupyterhub

Realizamos una serie de modificaciones en el fichero de configuración del Jupyterhub para que se creen volúmenes independientes para cada usuario del servicio y uno común a todos ellos, de esta forma, y montando el nas en el directorio en el que se montan los volúmenes Docker, podemos salvar todos los contenedores de cada usuario fuera del servidor, además de mantener los ficheros en caso de que se elimine el contenedor del usuario.

Primero montamos el nas en el directorio /mnt/nas y copiamos en él el fichero metadata.db del directorio /var/lib/docker/volumes para que al montarlo en dicho directorio no dé errores.

Además, modificamos el fichero /etc/fstab para añadir una última línea en la que hacemos que el nas se monte en el boot del servidor en el directorio donde se montan los volúmenes Docker, lo cual nos interesa para que, en caso de reinicio del servidor los volúmenes se guarden en el nas de igual manera. El fichero queda de la siguiente manera:

```
# /etc/fstab: static file system information.
#
# Use 'blkid' to print the universally unique identifier for a
# device; this may be used with UUID= as a more robust way to name devices
# that works even if disks are added and removed. See fstab(5).
#
# <file system> <mount point>   <type>  <options>       <dump> <pass>
/dev/mapper/ubuntu--vg-root  /                       ext4 errors=remount-ro 0       1
# /boot was on /dev/sda1 during installation
```

```
UUID=efe584b8-0ba5-4858-b3c5-394bbb489bc9 /boot           ext2
defaults        0       2

/dev/mapper/ubuntu--vg-swap_1 none               swap     sw
0       0

10.0.101.207:/BACKUP_CLUSTER   /var/lib/docker/volumes   nfs
auto  0  0
```

Añadimos un volumen común a todos los usuarios, a modo de carpeta compartida.

Nota: Los volúmenes que se crean se define en el fichero jupyterhub_config.py, mostrado al final de este apartado 8.

8.5. Fichero de configuración de Jupyterhub

Por último, se muestra cómo queda finalmente el fichero de configuración de Jupyterhub (/etc/jupyterhub/jupyterhub_config):

```
##################################################################

##      DockerSpawner

##################################################################

from dockerspawner import DockerSpawner

c.JupyterHub.spawner_class = DockerSpawner

c.DockerSpawner.volumes     =     {'jupyterhub-{username}':
'/home/UROP_student/work',             'jupyterhub-shared':
'/home/UROP_student/shared'}


# Image

c.DockerSpawner.image = 'jupyterhub/dl6'


# GPU

c.DockerSpawner.read_only_volumes                              =
{"nvidia_driver_390.25": "/usr/local/nvidia"}

c.DockerSpawner.extra_create_kwargs.update                     =
{"volume_driver": "nvidia-docker"}

c.DockerSpawner.extra_host_config = {

    "devices":     ["/dev/nvidiactl",     "/dev/nvidia-uvm",
"/dev/nvidia-uvm-tools", "/dev/nvidia0", "/dev/nvidia1"]}
```

```
#########################################################################
##################
##      Network Configuration
#########################################################################
##################

## The port and IP address of the proxy server
c.JupyterHub.port = 8080
c.JupyterHub.ip = '157.88.130.167'

# Docker
import netifaces
docker0 = netifaces.ifaddresses('docker0')
docker0_ipv4 = docker0[netifaces.AF_INET][0]
c.JupyterHub.hub_ip = docker0_ipv4['addr']

## The port for the Hub process
c.JupyterHub.hub_port = 8081

#########################################################################
#################
## Authentication

c.Authenticator.delete_invalid_users = True
c.Authenticator.whitelist = {'dilbert', 'jose', 'sergio',
'pacper', 'marmar', 'roberto'}
c.Authenticator.admin_users = {'dilbert'}
c.JupyterHub.authenticator_class                                =
'jupyterhub.auth.PAMAuthenticator'

#########################################################################
################
## Log
c.JupyterHub.extra_log_file = '/var/log/jupyterhub.log'
```

Ahora se explica cada apartado de este fichero de configuración:

- DockerSpawner: Es la clase con la que se crearán los contenedores, por ello se define como clase de Jupterhub.
    - Volúmenes: Se definen el nombre del volumen y la carpeta que se guardará en el volumen. En este caso definimos uno privado para cada usuario con su carpeta work, y otro común a todos, en la carpeta shared.
    - Imagen: Se define la imagen a partir de la cual DockerSpawner creará los contenedores. En este caso: jupyterhub/dl6.
    - GPU: En este apartado se indican los volúmenes de solo lectura, en este caso, el de nvidia, para que DockerSpawner cree los contenedores con el comando nvidia-docker, en vez de con docker, para que así tengan la funcionalidad de utilizar las GPUs del servidor.
- Network Configuration: Aquí se definen todos los aspectos de la red del Jupyterhub:
    - Puerto e IP del proceso: 8080 y la IP pública del servidor.
    - Docker: En este apartado se hace que el hub escuche en la IP de Docker, para que se pueda tener comunicación con los contenedores.
    - Puerto del hub: Se define el 8081.
- Authentication: Se define la forma en que se autentifican los usuarios para poder hacer uso del servicio.
    - Se indica que utilice la autentificación por defecto (PAM).
    - Se indica una lista blanca, es decir, los usuarios que pueden hacer uso del servicio.
    - Se indica una lista de administradores del servicio.
    - Se indica que, en caso de borrar algún usuario, lo borre de las cookies que guarda (delete_invalid_users).

9. CUDA

Descargamos la versión de CUDA de la página de nvidia, paramos el servidor X y seguimos las ayudas del prompt y realizamos lo siguiente:

Añadimos en el root /etc/bash.bashrc (para que afecte a todos los usuarios):

```
export PATH=/usr/local/cuda-9.1/bin${PATH:+:${PATH}}
export                      LD_LIBRARY_PATH=/usr/local/cuda-9.1/lib64${LD_LIBRARY_PATH:+:${LD_LIBRARY_PATH}}
export CUDA_HOME=/usr/local/cuda
```

Para actualizarlo escribimos:

```
source /etc/bash.bashrc
```

De esta forma se nos ha cambiado el PATH en todos los usuarios.

Instalación de CUDNN:

Descargamos la versión de cudnn (Runtime library, Developer Library y Samples Code for Ubuntu 16.04(Deb)) y lo movemos a /usr/local/cuda-9.1 y vamos a ese directorio, en el que ponemos:

```
dpkg -i <nombre_del_fichero_descargado>
```

Verificamos la instalación:

```
cp -r /usr/src/cudnn_samples_v7/  $HOME
cd $HOME/cudnn_samples_v7/mnistCUDNN
Make clean && make
./mnistCUDNN
```

Si hemos instalado bien nos debe salir:

```
Test passed!
```

ANEXO II

Preparación previa

En este anexo se incluyen los cursos que se han realizado de cara a la realización de este Trabajo Fin de Grado y que resultan de gran interés como introducción al mundo de la Inteligencia Artificial y del Deep Learning.

1. Cursos

1.1 Curso de Introducción al Machine Learning

Este curso se realizó como introductorio al ámbito de la Inteligencia Artificial y del Machine Learning en la plataforma de cursos Udacity. Este curso constaba de 17 capítulos en los que se pueden ver aspectos como:

- Una introducción a la Inteligencia Artificial y al Machine Learning.
- Aprendizaje sobre algunos de los algoritmos de Aprendizaje Supervisado (*Supervised Learning)* del Machine Learning más utilizados, como, por ejemplo: *Naive Bayes, Support Vector Machine (SVM), Decision Trees (DT).*
- Aprendizaje sobre algunos de los algoritmos de Aprendizaje No Supervisado (*Unsupervised Learning)* del Machine Learning más utilizados, como, por ejemplo: *k-means, Clustering…*
- Visión sobre la regresión lineal, los *outliers*, cómo elegir un algoritmo antes que otro, sistemas de reducir la dimensionalidad de un problema de este tipo como *PCA (Principal Component Analysis) …*

- Una serie de actividades y ejercicios para profundizar en los conceptos teóricos vistos durante el curso, así como pequeños proyectos y ejercicios en los que se ha tenido que programar para la utilización de lo que se ha ido viendo durante el curso.

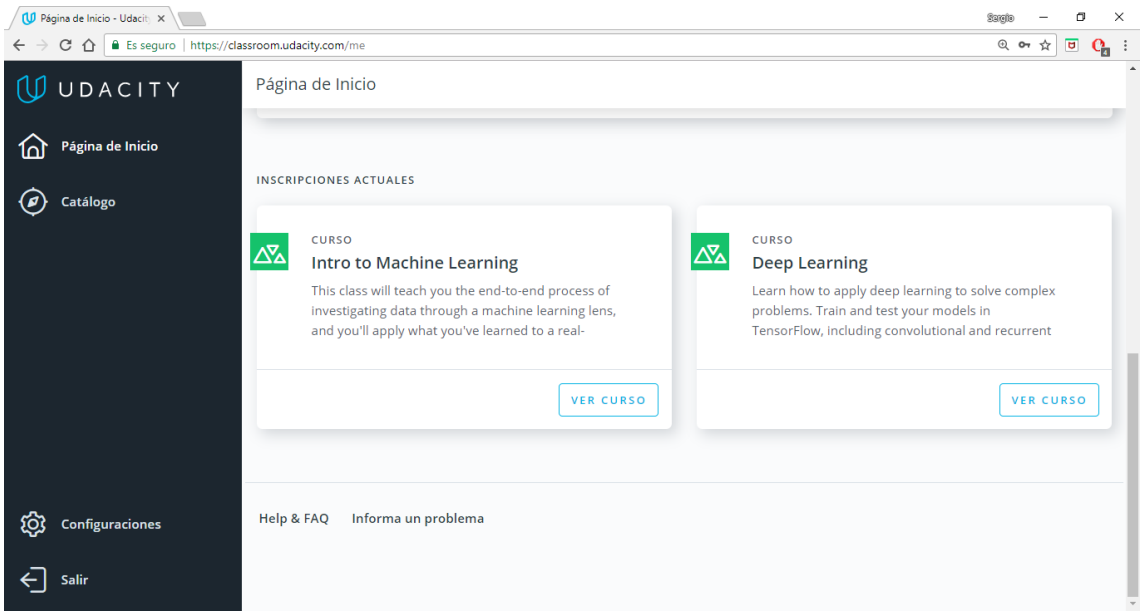


Figura. Página de inicio del perfil de Udacity.

1.2 Curso de Deep Learning

Curso de Deep Learning: Este curso se realizó para profundizar algo más sobre el Deep Learning, también realizado en la plataforma Udacity. El curso constaba de 7 capítulos en los que se pueden ver aspectos como:

- Un paso del Machine Learning al Deep Learning, en el que se dan los detalles que definen la filosofía del Deep Learning.
- Redes Neuronales Profundas.
- Redes Neuronales Convolucionales (*Convolutional Neural Networks, CNN).*
- Modelos para texto y secuencias, es decir, una introducción a las Redes neuronales recurrentes (*Recurrent Neural Networks, RNN),* más en concreto a las formadas por celdas LSTM (*Long Short-Term Memory).*

- Una serie de actividades y ejercicios para profundizar en los conceptos teóricos vistos durante el curso, así como un pequeño proyecto en el que se incide en la programación y manejo de este tipo de redes.

Figura. Logotipo de Udacity.

1.3 Aprendizaje de Python y Tensorflow

También se tuvo que aprender el Lenguaje de Programación Python y de la utilización de la biblioteca de funciones Tensorflow, perteneciente a Google. Este aprendizaje se ha podido realizar gracias a las actividades y proyectos realizados en los dos cursos anteriormente mencionados.

1.4 Curso con DIGITS

También se realizó un curso de clasificación de imágenes con DIGITS (*Image Classification with DIGITS*) de la plataforma Qwiklabs. En este curso, de corta duración, se han podido utilizar herramientas para la clasificación de imágenes y el uso de redes neuronales para tal fin mediante el uso de Jupyter y DIGITS.

Figura. Logotipo de Qwicklabs.