Universidad Internacional de Valencia

Modelo de IA Para La Clasificación de Señales De Radiofrecuencia Limpias o Alteradas

Titulación:
Máster U. en Inteligencia Artificial

Curso académico
2021-2022

Alumno/a: Cano Moya, Alejandro
D.N.I: 50998817-C

Director/a de TFM: José Gabriel García Pardo

Convocatoria:

Tercera

10 Mayo 2022

De:
Planeta Formación y Universidades

Universidad Internacional de Valencia

RESUMEN

Este Trabajo de Fin de Máster nace de la necesidad de una empresa de ser capaz de detectar cuándo una señal de radiofrecuencia está siendo interferida intencionadamente y de cómo mediante el uso de técnicas de Inteligencia Artificial y Deep Learning se va a tratar de solventar este problema de una manera distinta a la programación tradicional. Por tanto, este trabajo se va a tratar de un proyecto aplicado a un problema en la industria.

Las señales de radiofrecuencia son, en muchas ocasiones, blanco de actividades de interferencia intencionada (o *jamming*) producidas de manera deliberada por emisiones destinadas a hacer ininteligibles o falsear parcial o totalmente una señal objetivo, en especial en territorio militar o estratégico. Debido a esto, la exploración de una nueva herramienta capaz de detectar cuando una señal está siendo interferida de manera intencionada podría ser un avance en la seguridad de este campo ya que permitiría tomar una decisión en consecuencia al aviso que proporcionaría dicha herramienta.

Para abordar el problema, se va a utilizar un hardware capaz de generar y capturar señales de radiofrecuencia del tipo *Binary Phase-Shifting Key* (o BPSK) de manera que se podrá generar una base de datos controlada de señales tanto no interferidas, o limpias, como de señales interferidas. Para ello se utilizarán una serie de herramientas hardware para poder adquirir y procesar dichas señales y se crea una base de datos propia para guardar dichas señales y generar los *datasets* necesarios para el entrenamiento del modelo. Se generan un total de 80 señales (45 limpias y 35 alteradas) con 1000 registros de tiempo que se subdividirán en 800 señales de 100 registros de tiempo, pudiendo crear mapas de valores de señal de 100 registros por 121 valores de señal. Por otro lado, el software para poder procesar dichas bases de datos y generar una herramienta que permita clasificar dichas señales, serán un programa basado en el lenguaje de programación Python 3 y sus librerías y un modelo entrenado en base a dichos datos procesados.

Para ello, se va a desarrollar un modelo de Inteligencia Artificial, utilizando la librería de TensorFlow, Keras, para la clasificación de señales de radiofrecuencia limpias o alteradas analizando los patrones en los datos de las señales. Será un modelo convolucional que al ser entrenado con mapas de valores de señal, servirá como detector de interferencias intencionadas en las señales de comunicación en un ecosistema de dispositivos de la empresa en cuestión. Además, se utilizará la técnica de *fine tuning* con un modelo pre-entrenado para comprobar la eficacia del modelo creado en este trabajo.

Finalmente se verán los resultados en los que se aprecia cómo la metodología del proyecto permite que tanto el modelo creado como el pre-entrenado dan unos resultados de precisión muy altos con la diferencia de la optimización del modelo creado en este proyecto gracias a un menor número de parámetros, y por tanto, velocidad de procesado.

Por último, dado que este proyecto es de carácter privado, no se podrán mostrar datos sobre las señales generadas así como otras partes sensibles del trabajo. Además aclarar que se trata de una primera fase de lo que pretende ser un proyecto más ambicioso con una cantidad de datos mucho mayor, nuevas funcionabilidades, mayor precisión y escalabilidad para poder ser introducido en un ecosistema hardware y software específico de generación o recepción y procesado de señales de radiofrecuencia.

ÍNDICE

1. INTRODUCCIÓN

1.1 Motivación y descripción del problema

La motivación para realizar este Trabajo de Fin de Máster (TFM) es la de satisfacer la necesidad de un departamento de desarrollar una herramienta para detectar interferencias intencionadas, o *jamming*, en señales de comunicaciones.

Esta necesidad, junto con el deseo de generar una aplicación de Inteligencia Artificial aplicable al entorno laboral del autor del proyecto hace una simbiosis en la que se satisfice tanto la necesidad investigativa de la empresa como el aprendizaje en el mundo de la Inteligencia Artificial generando un enlace con el sector en el que trabaja el autor.

La radiofrecuencia [1] es la zona con menos energía del espectro electromagnético, que se encuentra entre los 3 hercios (Hz) y 300 gigahercios (GHz). Esta zona del espectro se utiliza, entre otros usos, para las radiocomunicaciones y radar, siendo por tanto una zona donde se realizan muchas aplicaciones de comunicación militares y civiles.

El problema que se trata en este trabajo es el de la detección de interferencias intencionadas de señales de radiofrecuencia. Los ataques de *jamming* [2] pueden dañar significativamente la calidad y rendimiento de las señales de comunicaciones a las que atacan. Un ejemplo de jamming sería el que vemos en la *Figura 1*, en la que se puede ver a la izquierda la señal limpia, sin alterar, y a la derecha la misma señal con una señal de *jamming* interfiriéndola:

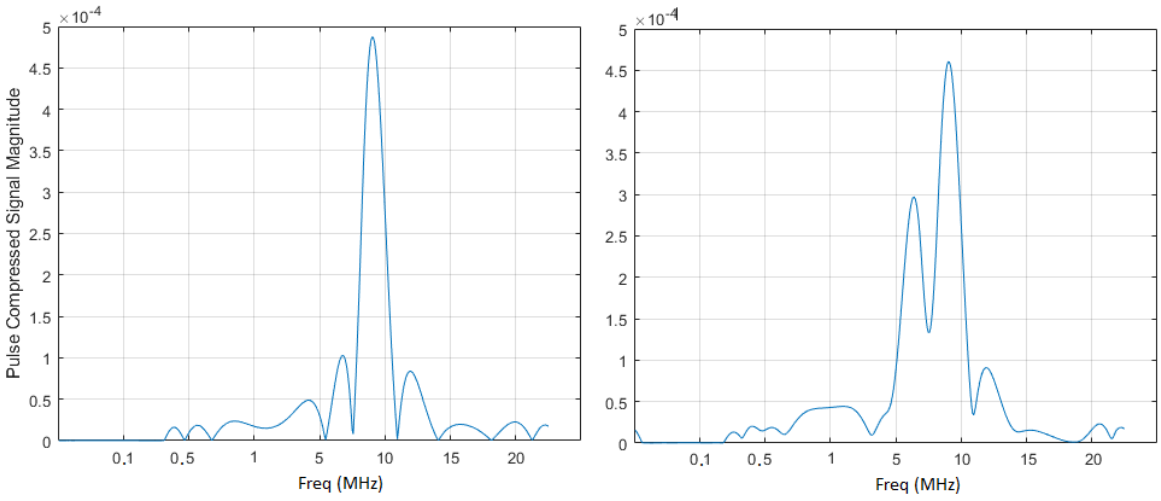


Figura 1: Señal limpia y señal con jamming

En este caso, se puede ver con claridad la diferencia entre ambas señales, ya que son pulsos simples que se pueden diferenciar en la representación gráfica que se puede ver en la anterior figura. Pero cuando las señales son más complejas es más difícil visualizar cuándo hay *jamming* y cuando son otros parámetros de señal, ruido de fondo, alteraciones naturales de la señal o cualquier otra variable que no se corresponde con una alteración intencionada.

En este trabajo se va a tratar un tipo de señal llamada PSK [3], del inglés, *Phase-Shift Keying*¸ que es un tipo de señal modulada por desplazamiento en fase. Este tipo de modulación digital es un proceso que transmite datos cambiando o modulando la fase de una señal de referencia con una frecuencia constante, es decir, la señal portadora. Para este proyecto se van a ver uno de los tipos de señales PSK, en concreto, las señales BPSK. Estas son un tipo de señales PSK pero controladas por un bit de control digital que es el encargado del cambio de fase de señal, como se puede ver en la *Figura 2*.

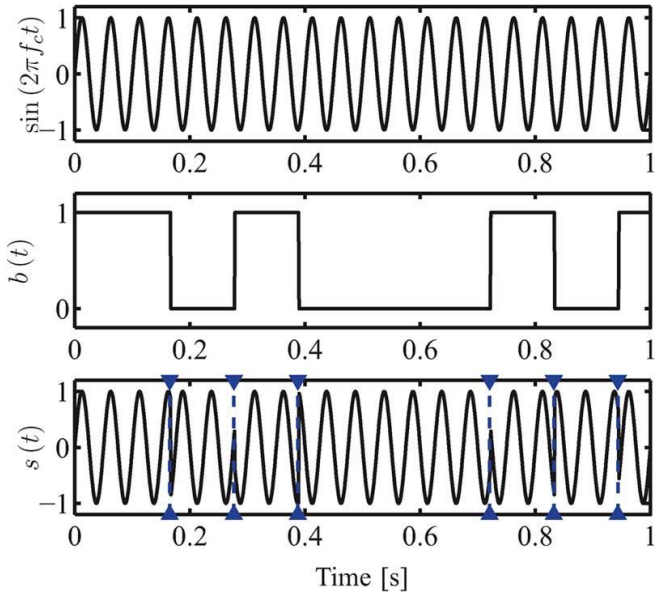


Figura 2: Ejemplo de modulación BPSK

Este tipo de señales, como otras de la misma familia de las PSK, tienen unas características asociadas a valores de señal llamados IQ que serán tratados más adelante en este mismo apartado. Estos valores permiten generar unas constelaciones de puntos que dan información sobre la señal de la manera que se ve en la *Figura 3*.

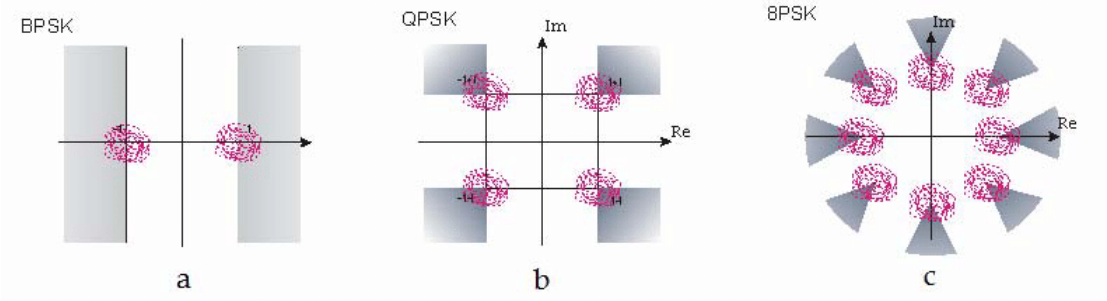


Figura 3: Diagramas de constelación BPSK, QPSK, 8PSK

Como se puede ver en la *Figura 4*, la constelación IQ da información sobre el estado de la señal, en este caso, dependiendo de la relación de señal con ruido (SNR).

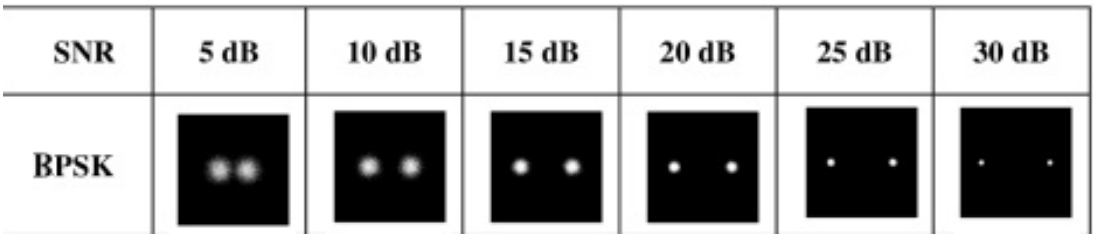


Figura 4: Diferencias modulación BPSK con respecto a señal

Por ello, se decidió investigar el comportamiento de estas señales cuando se les aplicaba una alteración intencionada de señal (*jamming*) y utilizar los datos I, Q y potencia de señal para su detección y capturar una ventana de tiempo de datos para usarlos como el dataset del trabajo. El propósito del trabajo será generar una matriz de datos a partir de estos valores y crear un mapa bidimensional de valores y un modelo que pueda detectar patrones en dichos mapas para clasificar las señales a las que pertenecen.

Para realizar esta tarea, gracias a que se dispone de material de instrumentación, de medición y de generación de señales, se va a crear un entorno en el que mediante un hardware específico se pueden generar y recoger señales de radiofrecuencia como las que se pretenden detectar en el entorno real, tanto limpias como alteradas para y generar un dataset de datos de señal, entre los que están los valores IQ, para entrenar un modelo permitiendo clasificar una señal en limpia o alterada.

En electrónica [4], una señal sinusoide con modulación angular siempre se podrá descomponer en dos señales sinusoides moduladas en amplitud que están desfasadas en un cuarto de ciclo (π/2 radianes o 90 grados). A estas sinusoides moduladas en amplitud se las conocen como componentes en fase y en cuadratura y tienen la misma frecuencia central que la señal original. Esto se puede ver en la *Figura 5* siendo la señal verde la señal original, la señal azul la componente en fase, I(t), y la roja la componente en cuadratura, Q(t):

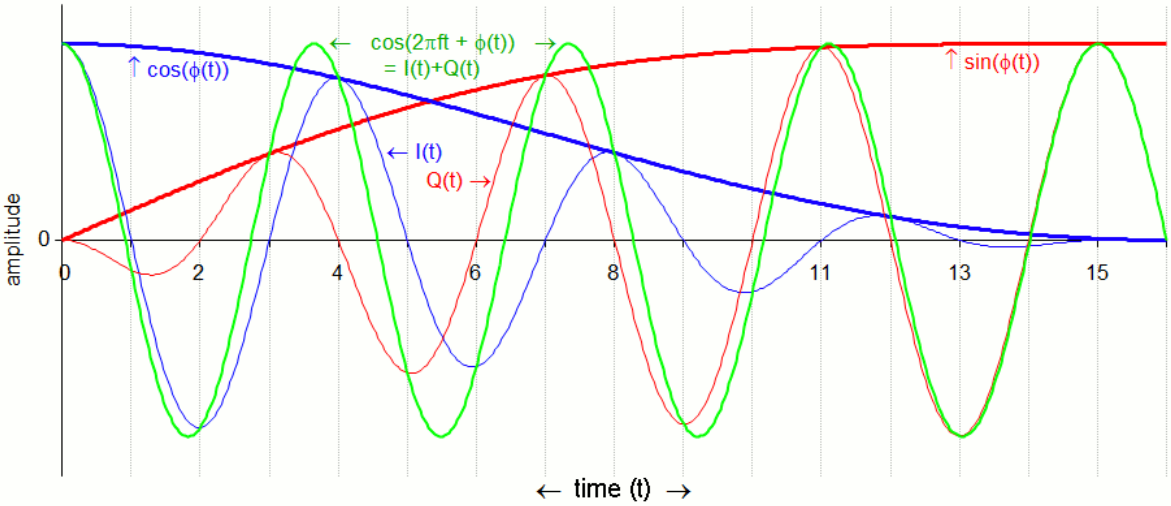
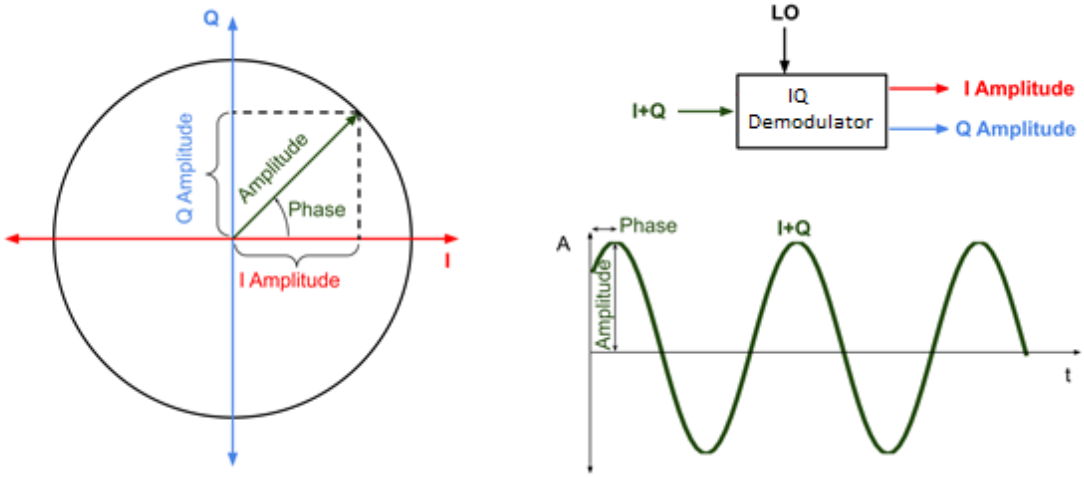


Figura 5: Componente en fase y cuadratura de una señal

De esta manera, los analizadores de señales vectoriales o analizadores de espectro en tiempo real funcionan convirtiendo la señal de RF entrante en muestras de banda base (IQ) en cuadratura en función del tiempo. Todas las mediciones y resultados que se pueden mostrar, desde mediciones de espectro simples hasta análisis de modulación complejos, se calculan a partir de estas muestras de IQ.

Gracias a un hardware que permita desmodular una señal en sus componentes I y Q (proceso simplificado en *Figura 6*), se podrá trabajar con dichos datos y darle el enfoque al proyecto que veremos más adelante.



I+Q		I Amplitude (V)	Q Amplitude (V)
Amplitude (V)	Phase (°)		
1	0	1	0
1	90	0	1
1	180	-1	0
1	270	0	-1

Figura 6: Diagrama IQ

El bloque de la demodulación IQ que se muestra como “caja negra” en la *Figura 6* se ve de manera simplificada en la *Figura 7* (siendo LO *local oscilator* u oscilador local):

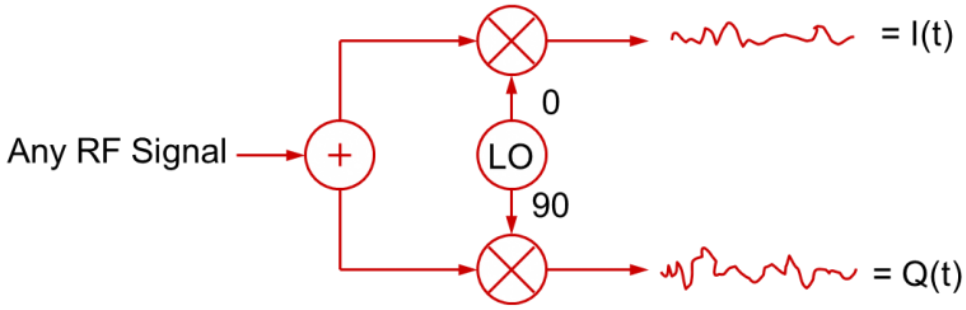


Figura 7: Proceso demodulación IQ

Mediante instrumentación que es capaz de obtener registros de los distintos valores que corresponden a las señales de radiofrecuencia, se pretende analizar y procesar estos datos y buscar una manera eficiente de crear un modelo y entrenarlo para ser capaz de detectar qué tipo de señal se está clasificando.

Además, por estrategias del departamento, trabajar con los mismos parámetros que se utilizan para realizar mediciones y resultados, como se ha visto en los párrafos anteriores, servirán para tener una herramienta mejor integrada en el ecosistema hardware actual del departamento.

Por este motivo, se ha decidido en este trabajo enfocarse en estos parámetros, I y Q, junto con la potencia de la señal para, por un lado, tener un enfoque basado en las variables con los que se miden los parámetros de las señales y permitir una inclusión de la herramienta clasificadora de señales y, por otro, para darle un enfoque en el que se tienen varias variables producto del desglose de una señal con las que entrenar al modelo, lo cual dará un enfoque interesante tanto desde el punto de vista de la obtención de los datos como de su uso para entrenar al modelo. Para ello, se procesarán los datos generando mapas de características bidimensionales que pueden ser abordados mediante técnicas convencionales de Deep Learning basadas en imagen.

En este trabajo se van a utilizar técnicas de aprendizaje supervisado al clasificar y etiquetar los datos que se van a introducir al modelo. El aprendizaje supervisado es una rama de Machine Learning, un método de análisis de datos que utiliza algoritmos que aprenden iterativamente de los datos para permitir que el modelo encuentre información escondida sin tener que programar de manera explícita dónde buscar. Se utilizarán técnicas de Deep Learning para generar y entrenar un modelo convolucional capaz de clasificar el conjunto de muestras correspondientes a señales parciales o totales.

Gracias a la diferencia entre la programación tradicional y las técnicas de Inteligencia Artificial se pueden obtener resultados no basados en un algoritmo, sino que se enseñará a aprender patrones a un modelo que podrá clasificar dichas señales en base a lo aprendido.

1.2 Objetivos

El objetivo general de este trabajo es el de realizar el trabajo en su totalidad de conseguir desarrollar un modelo predictivo basado en redes neuronales convolucionales para discernir entre señales limpias o con *jamming*, a fin de introducirlo en un ecosistema hardware y software. Para lograr la consecución de este objetivo general, se proponen una serie de objetivos específicos que se irán abordando de manera secuencial a lo largo de este TFM:

- Analizar el estado del arte para ofrecer una alternativa o mejora a lo ya implementado en el campo de la detección de *jamming* con Inteligencia Artificial.

- Elaborar un entorno hardware capaz de generar y procesar señales de los tipos que se quieren analizar (limpias o interferidas) y que se consiga generar un número mínimo de distintas señales para que futuro procesado proporcione un set de datos variado.

- Elaborar un entorno software capaz de adquirir, procesar y generar un dataset balanceado de dichas señales para su posterior procesado

- Estudiar el comportamiento de las señales de radiofrecuencia y buscar patrones en sus parámetros que permitan tomar una decisión en cuanto a cómo se puede abordar el problema.

- Elaborar un programa en Python que permita procesar y particionar los datos para poder introducírselos a un modelo y que aprenda de la manera más eficiente posible. Para ello se utilizarán las técnicas de *hold-out* y *cross-validation* para separar los datos en particiones de entrenamiento, test y validación para dotar de robustez y fiabilidad a los modelos que se entrenen.

- En base a un enfoque de aprendizaje supervisado utilizando distintas estrategias de aprendizaje a partir de redes neuronales convolucionales, desarrollar un modelo utilizando las librerías de TensorFlow/Keras usando técnicas de Deep Learning y entrenar dicho modelo de manera que sea capaz de detectar patrones para poder clasificar las señales que se introducirán al modelo.

- Por último, uno de los objetivos fundamentales de este TFM es el de validar y evaluar los resultados obtenidos y dejar abierta una línea de investigación fijada en el refinamiento y posibilidad de un uso profesional del modelo en el campo de la empresa.

1.3 Guía de la memoria

El resto de la memoria del presente TFM se va a distribuir de acuerdo con las siguientes secciones:

1. Marco teórico: En esta sección se va a tratar el trasfondo teórico y el trasfondo de las técnicas de Inteligencia Artificial/Deep Learning que se van a utilizar para la resolución del problema y que inspiran a la realización de este TFM.

2. Estado del arte: Se realiza una exhaustiva revisión bibliográfica sobre los métodos y avances en relación al problema de detección de señales interferidas intencionadamente (o *con jamming*) en el campo de la Inteligencia Artificial. Así como se expondrán las diferencias entre este estado del arte y lo que se pretende aportar en este trabajo.

3. Material y métodos: En este apartado se listarán y se definirán los medios hardware y software utilizados en la elaboración de este proyecto, así como una descripción de cómo se ha generado y gestionado la base de datos utilizada en el proyecto para entrenar, validar y evaluar el modelo de Inteligencia Artificial. Además, en este apartado se definirán los métodos utilizados para generar y procesar los datos adquiridos y utilizados para la elaboración del proyecto. Por último, se explica detalladamente el algoritmo implementado para la resolución del problema.

4. Resultados: Se detallan en profundidad los diferentes experimentos que se realizan durante la elaboración del proyecto, así como un análisis de los resultados obtenidos de forma cuantitativa y cualitativa.

5. Discusión: Se analizan los resultados, explicando cuál es el modelo óptimo y por qué, analizando los resultados de la evaluación de los modelos.

6. Conclusiones y líneas futuras: Finalmente, se discuten las conclusiones del trabajo y se discute sobre la metodología, resultados y calidad del proyecto. Por último se explica y se deja fijada la línea de investigación y desarrollo futura para las siguientes fases del proyecto.

2. MARCO TEÓRICO

En esta sección de la memoria se van a tratar los conceptos fundamentales y las técnicas que se van a emplear para generar la arquitectura del modelo de Inteligencia Artificial, dejando de lado el marco teórico del otro campo que se ha visto, el de la radiofrecuencia, que ya se trató en la introducción pues esta versa sobre el problema en cuestión que se quiere abordar, poniéndolo en contexto. En esta sección se definen los conceptos teóricos necesarios para entender las técnicas que se van a implementar de Inteligencia Artificial.

2.1 Convolutional Neural Networks (CNNs)

Una red neuronal convolucional (o CNN por sus siglas en inglés) [5] es un tipo de red neuronal artificial en la que dichas neuronas se corresponden con campos receptivos de manera parecida a las neuronas que se encuentran en la corteza visual primaria del cerebro humano (*Figura 8*).

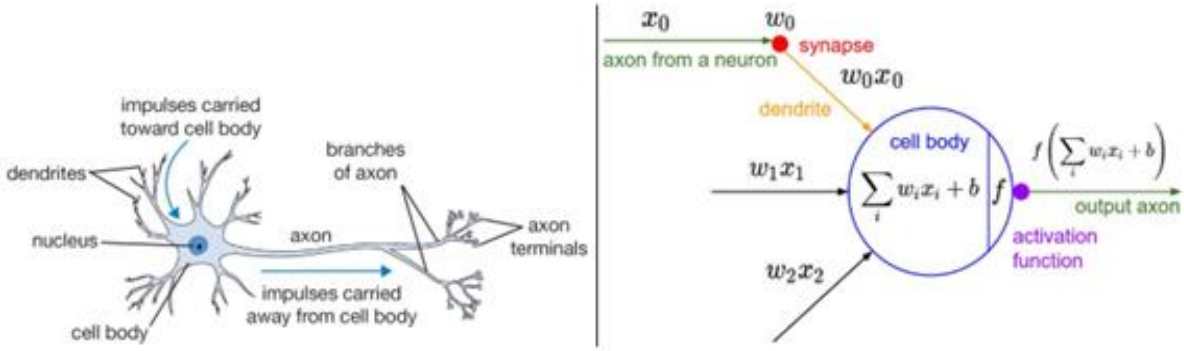


Figura 8: Comparativa entre una neurona biológica y su modelo matemático

Este tipo de red es una variante del perceptrón multicapa (*Figura 9*) el cual es una red neuronal artificial formada por varias capas de perceptrones simples. Estos perceptrones simples [6] son unidades básicas de inferencia en forma de clasificador binario o discriminador lineal, que genera una predicción en base al peso de las entradas combinado con un algoritmo. El perceptrón multicapa añade una o varias capas ocultas y posee como ventaja un algoritmo de aprendizaje, el algoritmo de retropropagación (en inglés *backpropagation*), [7] que emplea un ciclo de propagación en el que se aplica un estímulo que va desde la capa de entrada hasta la capa de salida y se calcula el error de la señal de salida con respecto a la que se esperaba, propagándolo hacia atrás. Esto causa una reordenación de los pesos en las capas anteriores, permitiendo que las neuronas se reorganicen a sí mismas para reconocer las distintas características de las entradas.

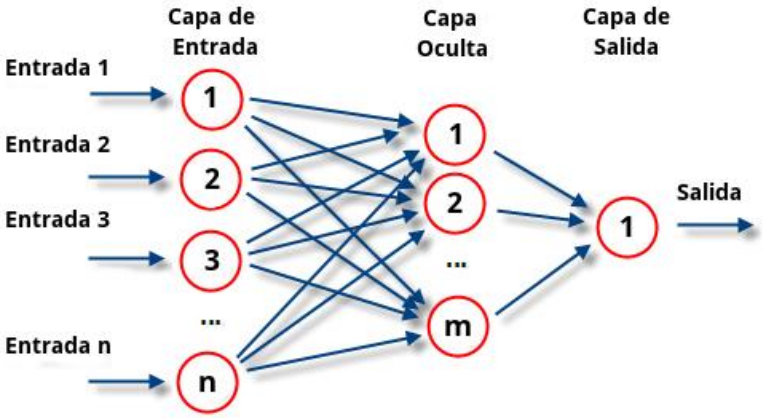


Figura 9: Diagrama Perceptrón Multicapa

En el caso de las redes convolucionales su aplicación se ejecuta en una red bidimensional, esto la hace muy eficaz para tareas de visión artificial como la clasificación y segmentación de imágenes [8].

Una red neuronal convolucional consta de varias capas de filtros convolucionales de una o más dimensiones. Según la rama de análisis funcional, en el campo de las matemáticas, la convolución es una operación entre dos funciones (*f* y *g*) cuyo resultado es una función (*f* * *g*) que expresa cómo se modifica la forma de una por la otra. Por convolución se refiere al proceso de calcular la función de resultado y a la función resultante de dicho proceso. Matemáticamente se define como la integral del producto de las dos funciones después de invertir y desplazar una de ellas. La función de convolución se halla cuando se evalúa la integral para todos los valores en el desplazamiento (*Figura 10*).

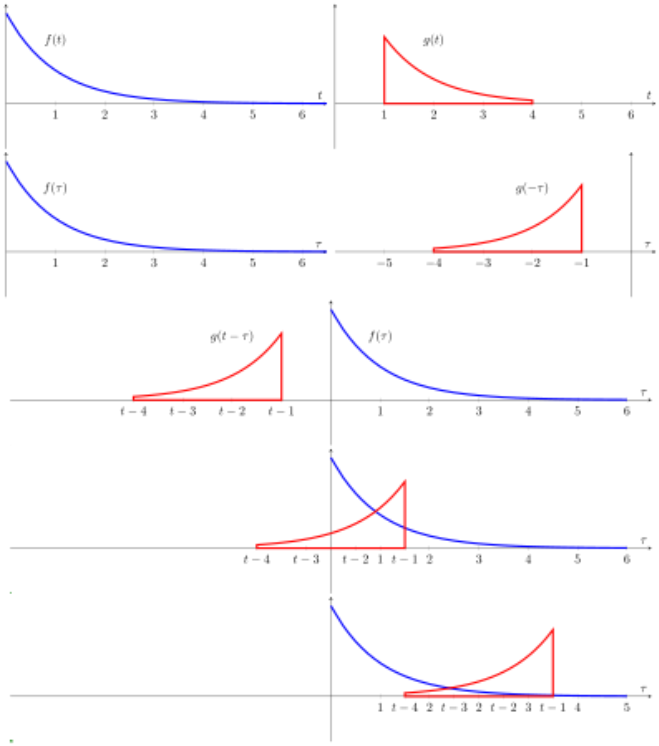


Figura 10: Proceso matemático de convolución

Este proceso en el campo del aprendizaje profundo, se desenvuelve de la siguiente manera [9], donde un *kernel*, filtro o máscara se hará pasar por toda la imagen, cuyos elementos serán multiplicados con los elementos correspondientes de la imagen, a fin de generar una salida. La expresión matemática para la convolución discreta de imágenes es la siguiente:

$$g[x,y] = \sum_{s=-a}^{a} \sum_{t=-b}^{b} w[s,t] f[x-s, y-t]$$

Figura 11: Expresión matemática de la función de convolución

Donde g[x, y] se refiere al resultado de la convolución en la posición x, y; a y b son el tamaño del filtro o *kernel*; w[s, t] es la función de los elementos del filtro y f[x - s, y - t] la región de la imagen en la que aplica el filtro. Este proceso de convolución se puede ver en la *Figura 12*:

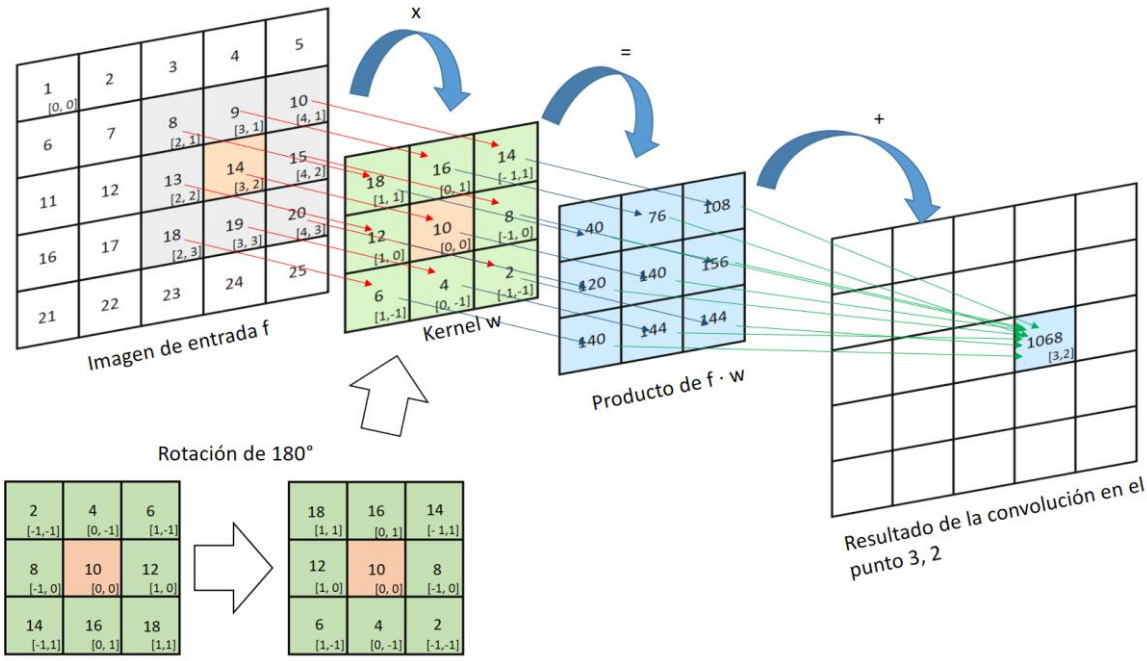


Figura 12: Proceso de convolución de una imagen

Se puede usar una imagen de entrada y un filtro para producir una imagen de salida convolucionando el filtro con la imagen de entrada. Esto consiste en:

1. Superposición del filtro sobre la imagen.
2. Realización de multiplicaciones por elementos entre los valores del filtro y sus valores correspondientes en la imagen.
3. Resumir todos los productos en cuanto a elementos, su suma es el valor de salida para el píxel de destino en la imagen de salida.
4. Repetición para todas las ubicaciones de la imagen.

Tras las capas convolucionales, se suele agregar una función de activación para realizar un mapeo de causalidad no lineal. Se realiza una extracción de características que se compone de neuronas convolucionales y neuronas de submuestreo. En la parte final de la red hay unas neuronas del tipo perceptrón simple que se encargar de la clasificación final de las características extraídas.

Esta etapa de extracción de características se realiza de manera similar al cómo se estimulan las células de la corteza visual. Esta fase consta de capas alternas de neuronas convolucionales y neuronas de submuestreo. Cuando los datos son introducidos por esta fase, las dimensiones de estos disminuyen y las neuronas de la capa exterior son menos sensibles a las variaciones de los datos de entrada, pero al mismo tiempo, se activan con funciones cada vez más complejas.

La arquitectura básica de una red convolucional es [10]:

- **Entrada**: Se corresponde con los pixeles de la imagen según alto, ancho y profundidad. La profundidad podrá ser de 1 sólo color para blanco y negro o de 3 para RGB.
- **Capa De Convolución**: Se aplicará el proceso de convolución. Esto procesará la salida de neuronas que están conectadas en zonas locales de entrada, que se corresponden con los píxeles cercanos, calculando el producto escalar entre el valor de los píxeles y una pequeña región a la que están conectados en el volumen de entrada. El volumen de salida se corresponderá con el número de filtros a aplicar.
- **Función de activación** en los elementos de la matriz para activar o no los coeficientes de las neuronas.
- ***Pool* o *subsampling***: Se trata de reducir las dimensiones de alto y ancho de la imagen, manteniendo la profundidad.
- Opcional (caso del proyecto) **Red de neuronas** "*fully connected*" que conectará con la última capa de subsampling y finalizará con la cantidad de neuronas que queremos clasificar.

En la *Figura 13* se puede ver una representación general de una arquitectura de una CNN en la que los recuadros naranjas representan las convoluciones.

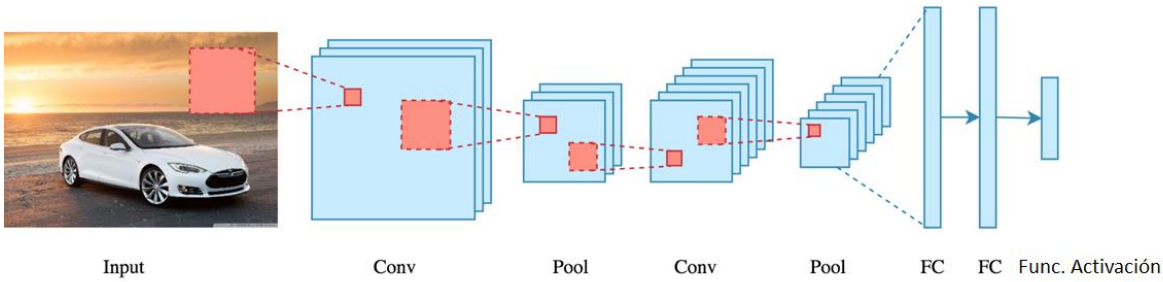


Figura 13: Arquitectura general de una CNN en la que se realiza

Gracias a estas técnicas, se puede crear una red que podrá reconocer cierto tipo de imagen porque ha sido entrenada con ella o con imágenes similares anteriormente muchas veces, pero no solo buscará imágenes semejantes sino que podrá inferir imágenes que no conozca pero que relaciona y en donde podrían existir similitudes, y gracias a esto se puede entrenar a un modelo para que aprenda patrones. Esto se utilizará para la clasificación de las señales que se busca tratar en este trabajo generando mapas bidimensionales como se explicó en la introducción.

Las capas convolucionales permiten extraer características de manera automática al multiplicar los píxeles de la imagen con los valores de *kernel* de los filtros definidos. Las primeras capas convolucionales reconocen patrones más globales, mientras que las capas más profundas son capaces de extraer características más específicas, como los contornos de un determinado objeto, por ejemplo.

Se puede introducir un nodo entre redes neuronales o al final de ellas que se conoce como Función de Activación. Esta función se utiliza para determinar si una neurona se activa o no. Las funciones de activación se usan para propagar la salida de los nodos de una capa hacia la siguiente capa. Se verá esto en el siguiente apartado. Además se tratarán las capas de reducción o *pooling* con mayor detalle.

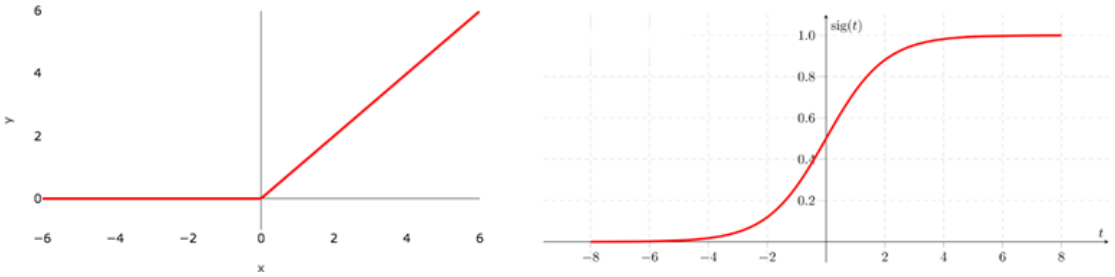
2.1.1 Funciones de activación

Una función de activación es una función que se agrega a una red neuronal artificial para ayudar a la red a aprender patrones complejos en los datos. Cuando se compara con un modelo basado en neuronas que está en nuestros cerebros, la función de activación al final decide qué se disparará a la siguiente neurona. Eso es lo que hace una función de activación red neuronal. Se toma la señal de salida de la celda anterior y la convierte en alguna forma que pueda tomarse como entrada para la siguiente celda.

Hay varios tipos de funciones de activación [11] [12] [13]. En este apartado se verán las usadas en el proyecto:

- **ReLU**: La función ReLU (por sus siglas en inglés *Rectified Linear Unit*) es una función no lineal que se utiliza como función de activación en redes neuronales. Se define como *f(x) = max(0,x)*. Esta es una función de activación ampliamente utilizada, especialmente con redes neuronales convolucionales. Es fácil de calcular y no causa el problema del gradiente de fuga. Dado que la salida es cero para todas las entradas negativas hace que algunos nodos mueran por completo y no aprendan nada. Esto a veces conduce a nodos inutilizables, a cambio, es una función computacionalmente muy eficiente. Su función gráfica se puede ver en la *Figura 14*.

- **Sigmoide**: La función sigmoide es otra función no lineal que se utiliza como función de activación en redes neuronales. Se define como *f(x) = 1/(1+e^-x)*. Una suma ponderada de entradas pasa a través de una función de activación y esta salida sirve como entrada para la siguiente capa. Este método se usa principalmente para problemas de clasificación binaria. Es una función con mayor coste computacional que la RELU. Su función gráfica se puede ver en la *Figura 15*.



Figuras 14 y 15: Función ReLU y Sigmoide

Como regla general, muchas veces la mejor función de activación se encuentra usando la lógica a partir de las nociones que se han visto en este apartado, pero muchas veces se necesita de heurística y probar distintos métodos y funciones hasta que se alcanza una versión optimizada del modelo. Se puede comenzar con el uso de la función ReLU y luego probar otras funciones de activación en caso de que no proporcione resultados óptimos.

2.1.2 Capa de reducción o pooling

La capa de reducción o *pooling* [14] generalmente se coloca tras la capa convolucional. Estas capas permiten reducir el peso de la representación para agilizar el aprendizaje de la red neuronal reduciendo las dimensiones espaciales (ancho x alto) del volumen de entrada para la siguiente capa convolucional sin afectar a la dimensión de profundidad del volumen. A esta operación también se le conoce como reducción de muestreo, pues la reducción de tamaño conduce también a la pérdida de información. A pesar de esta pérdida, se pueden sacar beneficios para la red debido a que la disminución en el tamaño conduce a una menor sobrecarga de cálculo para las próximas capas de la red y también sirve para reducir el sobreajuste.

La operación que se suele utilizar en esta capa es *max-pooling*, que divide a la imagen de entrada en un conjunto de rectángulos y, respecto de cada subregión, se mantiene el máximo valor de la zona, como se ve en la *Figura 18*.

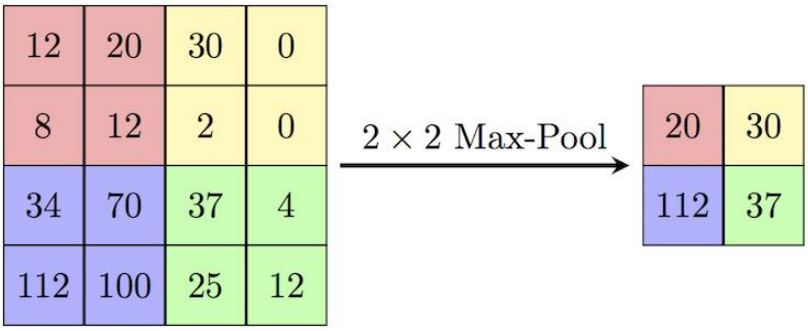


Figura 18: Ejemplo proceso max-pooling 2x2

Además, en este trabajo se va a utilizar la función *global-max-pooling-2D* [15] que realiza exactamente la misma operación que el bloque de *max-pooling* excepto que el tamaño de la agrupación (es decir, el factor de agrupación horizontal x el factor de agrupación vertical) es el tamaño de toda la entrada del bloque, es decir, calcula un solo valor máximo para cada uno de los canales de entrada a modo de función de aplanado (o *Flatten*) como se ve en la *Figura 19*.

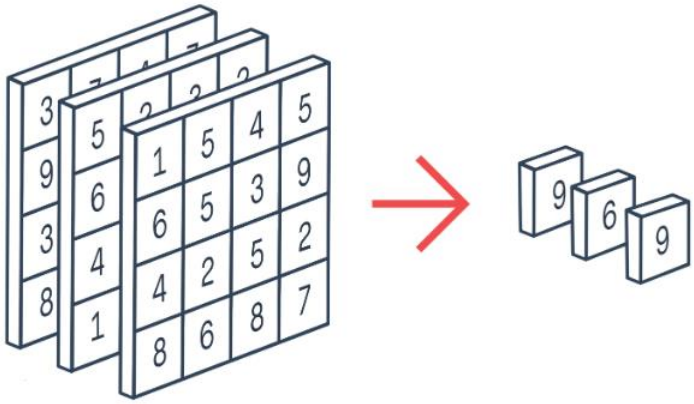


Figura 19: Ejemplo proceso global-max-pooling

3. ESTADO DEL ARTE

En este apartado se va a tratar el estado último sobre el contexto del problema a abordar en este proyecto:

Para ello, se ha realizado una investigación sobre varias fuentes de trabajo distinto. Empezando por la tesis de T. Nawaz*, D. Campo, M. O. Mughal, L. Marcenaro y C. S. Regazzoni titulada *Jammer Detection Algorithm for Wide-band Radios using Spectral Correlation and Neural Networks* [16], en la que se trata el concepto de Radio Cognitiva (CR por sus siglas en inglés de *Cognitive Radio*). Esta tecnología se utilizaría para la asignación automática de bandas de frecuencia donde situarse para transmitir señales. Sin embargo, esto produce un desafío de seguridad pues esta reconfigurabilidad y aprendizaje podría ser explotado por un atacante con una señal de interferencia intencionada. En este trabajo se realiza un algoritmo de detección de inferencias que constan de una señal de un tono de frecuencia utilizando espectros cíclicos y redes neuronales artificiales. En este trabajo se clasifican 20.000 señales que se introducen a una red neuronal artificial para entrenar, validar y testear en cinco tipos de señales: BPSK, QPSK, señal de interferencia (*jamming*), BPSK con *jamming* o QPSK con *jamming*. En este trabajo se compara la capacidad de clasificación de tipo de señal en relación al ratio señal-ruido (SNR por sus siglas en inglés de *Signal-to-Noise Ratio*).

Por otro lado, gracias al estudio realizado por Héctor Iván Reyes and Naima Kaabouch titulado *Jamming and Lost Link Detection in Wireless Networks with Fuzzy Logic* [17] se ve un acercamiento al problema del *jamming* mediante el uso de lógica difusa. El sistema que propone el proyecto utiliza los parámetros CCA (*Clear Channel Assessment*), BPR (*Bad Packet Ratio*), PDR (*Packet Delivery Ratio*) y RSS (*Received Strength Signal*) como entradas para evaluar el estado del enlace y, en caso de que se pierda, determinar la causa de la falla del enlace. Este acercamiento al problema aporta la capacidad de determinar que según los parámetros de señal, se puede determinar si hay una alteración en la señal.

Además, el estudio de Furqan, Haji M., Aygül, Mehmet A., Nazzal, Mahmoud y Arslan, Hüseyin titulado *Primary user emulation and jamming attack detection in cognitive radio via sparse coding* [18] trata sobre la implantación de un algoritmo para la detección de emulación de usuario principal y ataques de interferencia en una radio cognitiva. El algoritmo propuesto se basa en la codificación escasa de la señal comprimida recibida sobre un diccionario dependiente del canal. Más específicamente, los patrones de convergencia en la codificación dispersa según dicho diccionario se utilizan para distinguir entre un agujero de espectro, un usuario primario legítimo y un emulador o un *jammer*. El proceso de toma de decisiones se lleva a cabo como una máquina de clasificación basada en el aprendizaje automático. El éxito de los resultados se valida en términos de la métrica de calidad de la matriz de confusión.

Otro estudio a aportar son los papers de Silvija Kokalj-Filipovic, Rob Miller, Nicholas Chang y C. L. Lau con nombre *Mitigation of Adversarial Examples in RF Deep Classifiers Utilizing AutoEncoder Pre-training* [19] y *Adversarial Examples in RF Deep Learning: Detection of the Attack and its Physical Robustness* [20] que trata sobre cómo un algoritmo puede clasificar erróneamente ilustraciones por entradas ligeramente perturbadas y cómo se puede asemejar esto a detección errónea de señales de frecuencia (p. ej., BPSK se confunde con 8-PSK). En este paper se definen mecanismos de defensa basados en el pre-entrenamiento del clasificador objetivo utilizando un autocodificador como método de mitigación para subvertir ataques adversarios contra profundidad sistemas de detección de radar y comunicaciones basadas en el aprendizaje automático.

En el artículo de Ozan Alp Topal, Selen Gecgel, Ender Mete Eksioglu y Gunes Karabulut Kurt titulado *Identification of Smart Jammers: Learning based Approaches Using Wavelet Representation* [21], se trata cómo los *jammers* inteligentes pueden interrumpir la comunicación entre un transmisor y un receptor en una red inalámbrica y cómo el rastro que dejan es indetectable para las técnicas clásicas de identificación de bloqueadores, ocultos en el plano de tiempo-frecuencia. Se propone un método de identificación de bloqueadores que incluye un paso de pre-procesamiento para obtener una imagen de resolución múltiple, seguido del uso de un clasificador utilizando máquinas de vectores de soporte (SVM) y redes neuronales convolucionales profundas (DCNN). De esta manera se extraen automáticamente las características de las señales transformadas y se clasifican. Se consideran tres ataques de interferencia diferentes, la interferencia de bombardeo que tiene como objetivo el ancho de banda de transmisión completo, el ataque de interferencia de la señal de sincronización que tiene como objetivo las señales de sincronización y el ataque de interferencia de la señal de referencia que tiene como objetivo las señales de referencia en un escenario de transmisión de enlace descendente (LTE).

El estudio titulado *A Robust Jamming Signal Classification and Detection Approach Based on Multi-Layer Perceptron Neural Network* [22] realizado por Sahar Ujan, Mohammad Hossein Same, Rene Jr Landry trata sobre cómo la calidad de servicio (QoS) y la seguridad de un sistema de comunicación inalámbrico se pueden mejorar mediante un método de detección de interferencias de radiofrecuencia (RFI), que a su vez podría conducir a un proceso de mitigación efectivo. Debido a la gran variedad de señales RFI que existen se explica cómo se ha desarrollado una técnica de red neuronal de perceptrón multicapa (MLP) multiclase para reconocer señales de interferencia en un escenario de transmisión de video digital en tiempo real. El algoritmo que proponen puede clasificar la interferencia en su grupo relacionado o reconocer la Señal de interés (SoI). Además de analizar diferentes enfoques de aprendizaje (como aprendizaje en línea, lotes completos y mini lotes), la técnica de análisis de componentes principales (PCA) se implementa para seleccionar características más informativas para mejorar el rendimiento del clasificador. Se evalúa la robustez del clasificador entrenado para detectar señales desconocidas en diferentes relaciones señal/ruido (SNR) comparando el rendimiento del diseño propuesto con la técnica SVM en términos tanto de clasificación como de detección de señales de interferencia.

Por último, el artículo de Bikalpa Upadhyaya, Sumei Sun y Biplab Sikdar con nombre *Machine Learning-based Jamming Detection in Wireless IoT Networks* [23] se propone abordar este el problema de detección de *jamming* mediante el uso de algoritmos de aprendizaje automático para desarrollar un mecanismo de detección de interferencias de red pasivo implementando una serie de nodos dedicados para recopilar la información de la red para la detección de interferencias. Primero se considera la información de intensidad

de la señal (o RSSI por sus siglas en inglés *Received Signal Strength Indicator*) de radio para la detección de interferencias y después se valida con datos de red reales y simulados. Para ello utilizan los algoritmos de Machine Learning SVM, árboles de decisión y *Random Forest*.

Tras la revisión del estado del arte actual se puede realizar un análisis para aportar un avance en este campo, diferenciándonos de lo propuesto hasta ahora. Para ello se generará un modelo de Inteligencia Artificial propio para resolver el problema de detección de *jamming* para un ecosistema hardware y software.

La primera aproximación a esta línea de investigación y desarrollo, que es la que se ve en este TFM, se centrará solamente en las señales tipo BPSK. Por otro lado, las señales que se utilizarán no estarán limitadas a canales de banda estrecha (o *narrowband*) sino que podrán ser de banda ancha (o *wideband*). La diferencia es que no se está limitado por un ancho de banda de señal en concreto (menor a 1MHz), sino que se utilizarán señales que pueden tener un ancho de banda variable.

Además, las señales de interferencia o *jamming* no estarán limitadas a un tono de señal (por ejemplo 1kHz), sino que pueden tener un ancho de banda variable (entre 1kHz y 50MHz). Por otro lado, las señales de *jamming* no estarán limitadas a amplitudes mucho mayor que las de las señales que interfieren, sino que tendrán una amplitud variable.

Para la detección de interferencia se utilizarán los parámetros de I, Q y la potencia de señal RSSI. Para ello, entrenará a un modelo los datos de señales propiamente generadas tipo BPSK con diferentes anchos de banda y potencias de señal, de las cuales se tomarán valores sincronizados que se almacenarán en una base de datos para su posterior procesado.

El número de parámetros del modelo y tiempo de procesado de las señales será un factor crucial en la creación y elección de los modelos debido a que este proyecto se incluirá en un ecosistema hardware y software que captará y clasificará imágenes en tiempo real.

Por último, el acercamiento al problema que se realizará desde el punto de vista de la creación de una red neuronal artificial, para diferenciarse de lo visto en este apartado, será el de crear un modelo de red neuronal convolucional que permita detectar si la señal está siendo alterada intencionalmente a partir de la detección de patrones de un mapa de valores bidimensional creados con los parámetros de las señales que se han generado. Se compararán los resultados de precisión y número de parámetros del modelo creado, se compararán con un modelo ya pre-entrenado al que se le realiza la técnica de *fine tuning*, que consiste en utilizar un modelo pre-entrenado, cargar sus pesos y congelar el entrenamiento de sus capas excepto la capa de salida para entrenarlo con los datos del proyecto en cuestión.

4. MATERIAL Y MÉTODOS

En el presente apartado se va a tratar el material y los métodos que se han utilizado para el desarrollo de este proyecto. Al estar hecho en colaboración con una empresa, hay material sensible y confidencial, como qué dispositivos en concreto se utilizan o la propia base de datos, por lo que dicho material no se podrá tratar con mucho grado de detalle o especificar de qué se trata con total transparencia.

4.1 Material

Se ha utilizado una serie de material, herramientas dispositivos, tanto hardware como software para la generación de las señales y la obtención de los datos que se van a procesar para entrenar y probar el modelo, para guardar dichos datos en una base de datos, el procesado de dichos datos y la creación y entrenamiento del modelo de Inteligencia Artificial. Se van a ver a continuación, por tanto, el distinto material que se ha empleado para la elaboración del TFM.

4.1.1 Hardware

Para generar las señales de comunicación y las señales de interferencia (o *jammers*) se han utilizado dos radios definidas por software (o SDR). A diferencia de las radios tradicionales que tradicionalmente se han implementado vía hardware (como por ejemplo, filtros, mezcladores, amplificadores, moduladores o demoduladores, etc.) un SDR [15] es un sistema de comunicación por radio en el que los componentes se implementan mediante software en una computadora personal o sistema embebido equipada con algún tipo de interfaz de radiofrecuencia y un convertidor de señal de analógico a digital. Este diseño permite tener una radio que puede recibir y transmitir protocolos de radio (denominados formas de onda) muy diferentes entre sí basándose únicamente en el software utilizado. En la *Figura 20* se puede ver la arquitectura de estos dispositivos.

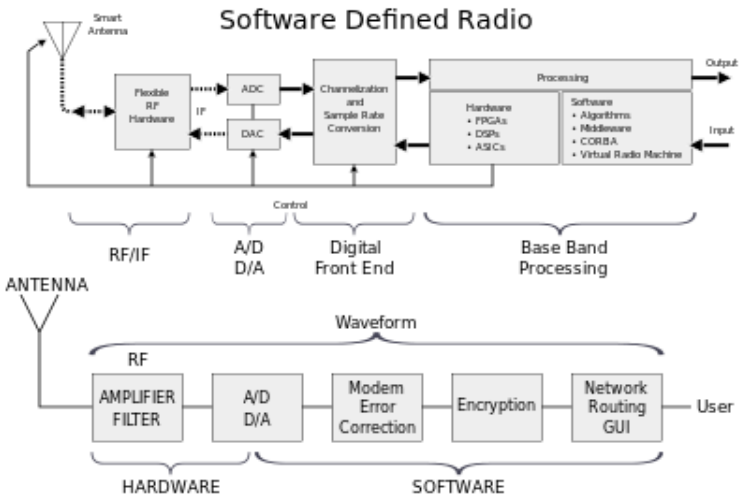


Figura 20: Diagrama general de una SDR

Para combinar las dos señales (la señal de comunicación que se va a clasificar) y la señal de *jamming* (o *no jamming*, dependiendo de la forma y potencia de esta), se ha usado como controlador para el mezclador de radiofrecuencia un dispositivo *USRP* (Universal Software Radio Peripheral) controlada mediante un software de control hecho con la interfaz gráfica LabVIEW. Esto, junto con un mezclador de radiofrecuencia se usa para sumar la señal de comunicación junto con la señal de *jamming* o con señal de ruido (caso en el que se considere que no hay señal aparte de la limpia). En la *Figura 21* se puede ver la arquitectura de estos dispositivos.

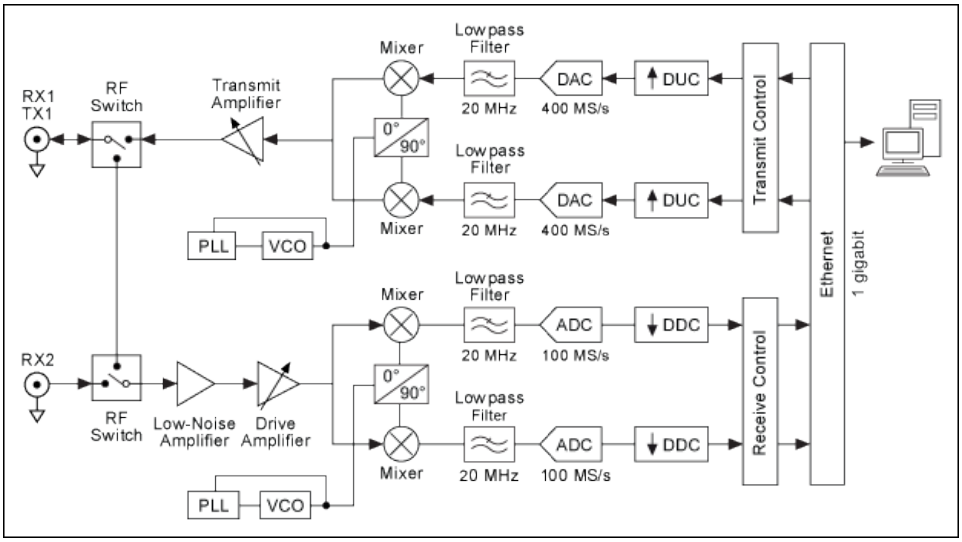


Figura 21: Diagrama general de una USPR

Además, se utilizarán varios PCs para controlar las dos SDRs y generar las señales deseadas, controlar la USRP y para recoger los datos de salida. También se utilizarán cables Ethernet para la interconexión de todos los dispositivos hardware y cables de radiofrecuencia para conectar las SDRs con la USRP. Por último se dispondrá de un servidor donde almacenar la base de datos donde se guardan los datos de las señales.

4.1.2 Software

En caso del software, este ha sido usado para controlar o programar los dispositivos hardware, así como para poder capturar los datos y generar una base de datos con ellos.

Para controlar la USRP se ha utilizado el software LabVIEW, el cual es un entorno de programación gráfica que se utiliza principalmente para desarrollar sistemas de producción, validación y automatización. Como se puede ver en la *Figura 22*, la interfaz gráfica permite controlar de manera sencilla y generar automatizaciones de una interconexión de los dispositivos conectados al PC.

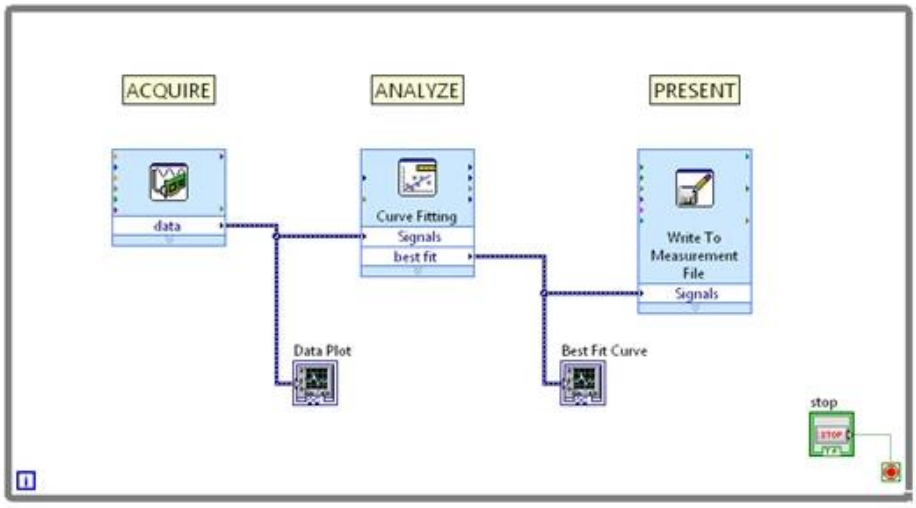


Figura 22: Interfaz ejemplo LabVIEW

Por otro lado, para controlar las SDRs se ha utilizado un software de control propio del laboratorio del departamento de la empresa.

Para poder almacenar los datos recibidos de las SDRs en una base de datos para su posterior procesado, se ha utilizado un servidor web NGINX para gestionar el tráfico del PC al servidor y el uso de una API REST que se encuentra en el servidor y que es el endpoint de entrada de datos y serialización de los mismos a la base de datos. Una API REST [19] es una API, la cual es una interfaz de programación de aplicaciones, que se ajusta a los límites del tipo de arquitectura REST. Entre otros, se encuentran los criterios de que ha de ser una arquitectura cliente-servidor cuya gestión de solicitudes es a través de HTTP.

Se ha utilizado el software WMware para poder ejecutar la máquina virtual donde se monte la base de datos dentro del servidor. VMware es una herramienta software que sirve como hipervisor alojado que se ejecuta en sistemas operativos. Esto nos permite generar máquinas virtuales en una sola máquina física y usarlas simultáneamente junto con la máquina que las aloja para capturar datos y subirlos a la base de datos.

Se ha utilizado Python 3.7 como lenguaje de programación para acceder a la base de datos, utilizando el framework Django. Django es [20] es un marco web Python de alto nivel que se ha usado para desarrollar la herramienta de gestión entre el NGINX y la base de datos.

Se ha usado Python 3.7 también para la aplicación de control de las SDRs y para la obtención y gestión de los datos de las señales que estas generan.

De la misma manera, se ha utilizado Python 3.7 como lenguaje de programación para generar el código de preparación, procesado, particionado de los datos y para generar el modelo final de Inteligencia Artificial que clasificará las señales y para el entrenamiento, validación y testeo del mismo.

Para la preparación y procesado de los datos se ha utilizado principalmente la librería de Python Numpy 1.21.4 y Pandas 1.3.5, para la partición de los datos se ha utilizado la librería sklearn y, finalmente, para la generación del modelo y su entrenamiento y validación de resultados, se ha utilizado las librerías TensorFlow 1.15.0 y Keras 2.3.1. TensorFlow es una librería de código abierto desarrollada por Google que se utiliza principalmente para desarrollar y entrenar modelos de aprendizaje automático. Sobre TensorFlow, se usará Keras, que es una librería que nos permitirá implementar bloques constructivos en la red neuronal a diseñar, como capas, funciones de activación y optimizadores. El hardware y software encargado de y las funcionesPara entender mejor el proceso que ocurre en cada capa, se explica la arquitectura y las capas donde ocurre esto en la *Figura 23*:

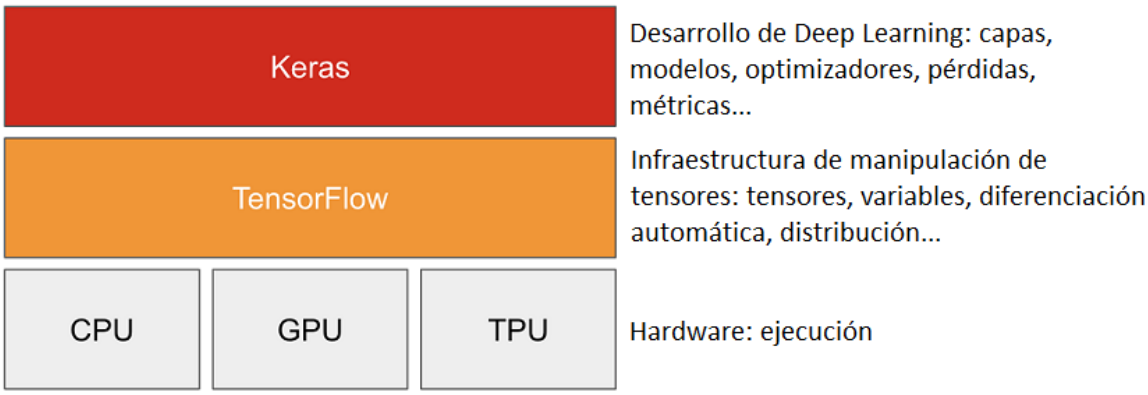


Figura 23: Infraestructura en la implementación de TensorFlow/Keras

Por último, para diseñar, implementar y ejecutar el código del procesado y partición de los datos y la creación, entrenamiento y validación del modelo, se ha utilizado la plataforma informática interactiva basada en web Jupyter Notebook.

4.1.3 Base de datos

La base de datos de este proyecto se obtiene de los datos recolectados de un total de 80 señales (45 señales limpias y 35 señales alteradas). Esta base de datos se compone de 160 ficheros de datos (2 tipos de ficheros .csv con diversos datos por cada señal) junto con un fichero .csv que clasifica y etiqueta cada una de esas 80 señales. Estos ficheros son descargados de una base de datos que se crea desde cero en un servidor físico del laboratorio del departamento.

Para no volcar todos los datos que transmite continuamente la SDR a un texto plano, poder darle formato .csv para diferenciar y poder elegir los datos que son relevantes de toda la cantidad de datos que proporciona la radio SDR, se decide crear una base de datos propia alojada en un servidor para generar y guardar los datos de una manera determinada para una mejor recopilación de los datos y futuro procesamiento de los mismos.

La base de datos está montada como estructura *full stack* con *docker compose* en un servidor externo, comunicándose el PC que recibe los datos con él. Internamente la arquitectura se ha llevado a cabo a través de la base de datos objeto-relacional (ORM) que trae el framework Django incorporado, así como la API se ha hecho con el framework Django REST para gestionar una base de datos gestionada en SQL.

En esta base de datos se van a almacenar en formato .csv los datos de señal con un formato de tiempo determinado, con un número de registros o (instantes de tiempo) concreto y con una disposición y selección de valores de señal determinados.

Se decide no realizar ningún método de detección de *outliers* o de detección de valores erróneos que hayan generado de forma incorrecta las radios digitales, pues se pretende que el modelo sea capaz de obviar esos instantes de medida errónea por sí mismo, a partir de los demás datos que el modelo esté procesando a la vez.

Una vez almacenados y con el formato deseado estos datos en la base de datos, se tienen dos ficheros .csv por cada señal que comparten instantes temporales en los que se almacenan distintos parámetros y datos de señal para un total de 80 señales. Esto resulta en un total de 160 ficheros .csv. Cada fichero consta de 1000 registros en los que cada registro es un instante temporal de señal en el que se ha leído y procesado datos de señal y se han guardado en la base de datos.

Alternativamente, se ha generado manualmente un tercer fichero Excel donde se etiqueta cada una de esas 80 señales como señal con la etiqueta "*No Jamming*" cuando la señal que de la que se capturaron esos datos es una señal considerada limpia y con la etiqueta "*Jamming*" cuando dicha señal se considera como que está alterada tal y como se muestra en la *Figura 24*:

	Signal	Class
2	NameSignal A	No jamming
3	NameSignal B	Jamming
4	NameSignal C	Jamming
5	NameSignal D	No jamming
6	NameSignal E	No jamming

Figura 24: Visualización del fichero clasificador señales

Por tanto, se habrá generado finalmente un dataset de tres tipos de ficheros en los que se encuentran los valores de señal en 80 (señales) * 2 (tipos de ficheros) clasificados como *NombreSeñal-IQ.csv* y *NombreSeñal-Signal.csv (*que hacen un total de 160 ficheros con 1000 registros cada uno) y un tercer fichero llamado *Clasificador_señales.xlsx* donde se etiquetan las 80 señales a analizar. Se adjunta a continuación en la *Figura 25* un esquema que ayuda a visualizar el conjunto de datos que se usa en este TFM, donde *NombreSeñal* se corresponde por el nombre asignado a cada señal de cada una de las 80 señales distintas que se generaron y guardaron

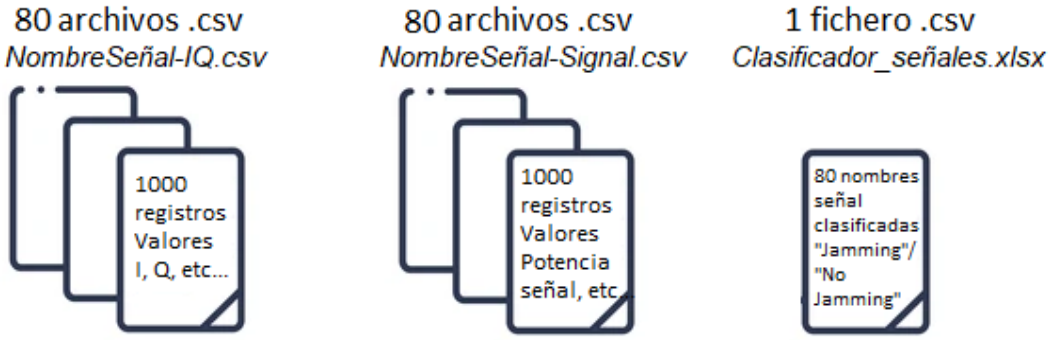


Figura 25: Esquema del conjunto de datos inicial

4.2 Métodos

4.2.1 Preparación de los datos

La obtención y preparación de los datos es uno de los puntos más interesantes de este trabajo. Para conseguir recrear las señales que se desean analizar, se han utilizado varias componentes hardware y software, como se ha visto en el apartado anterior, para la generación de las señales de radiofrecuencia y para generar una base de los datos que se generan de las mismas.

Para abordar el problema de detección de *jamming* en señales de radiofrecuencia se decidió generar las señales de radiofrecuencia a clasificar mediante radios definidas por software y recoger y procesar los datos para generar una base de datos de valores de señal que serán utilizados posteriormente para entrenar el modelo. En la *Figura 26* se puede ver un esquema simplificado del sistema para obtener los datos y guardarlos en una base de datos.

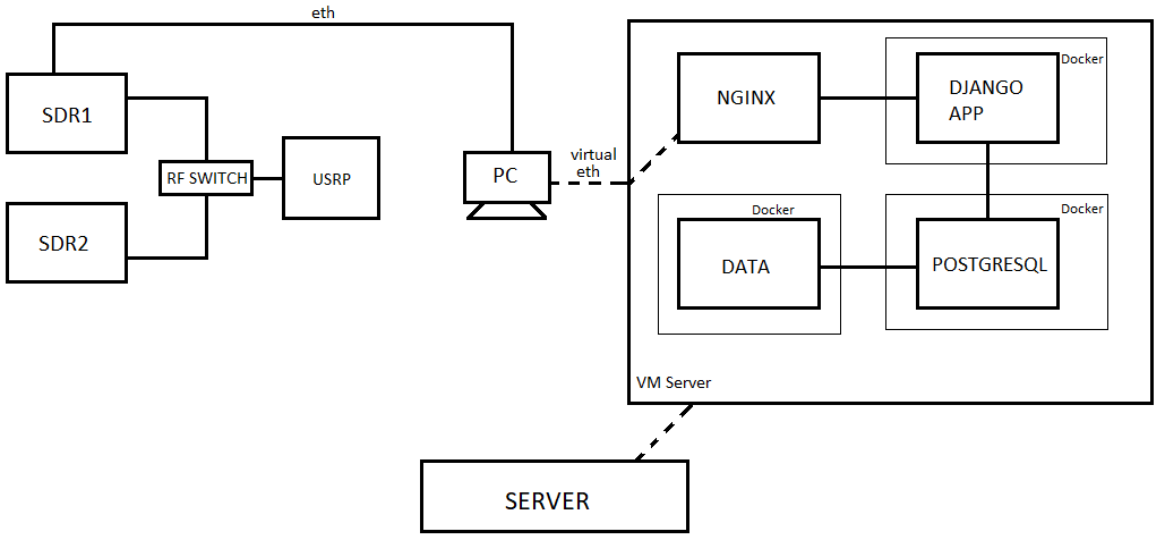


Figura 26: Arquitectura simplificada del sistema de adquisición de datos

Para generar los datos se usaron dos SDRs y una USRP entre las dos para introducir el *jamming* de una a otra. Una vez las radios están generando señales, conectadas mediante cable Ethernet a un PC con el software de control, se recogen los datos de estado de señal que se transmiten por el Ethernet de la SDR que transmite la señal ya sumada a la otra al PC que recogerá los datos. Los datos se transmiten en tramas binarias que son decodificadas para obtener los diferentes campos de señal. En estos datos se encuentran los valores IQ que vimos en la introducción, así como la potencia de la señal, entre otros valores de señal.

Una vez se tienen los valores, entre los que se encuentran los valores IQ, se castean a string y se monta la palabra con todos los IQs y demás valores de la medida que corresponda y, junto con otros datos, se montan en un archivo .json que es lanzado por http a la API REST que se aloja en el servidor. En la API, cuando el mensaje llega, se castea desde .json y usando el ORM que incluye framework Python, Django, se almacenan en una base de datos gestionada mediante el sistema PostgreSQL en una manera similar a la representada en la *Figura 27*.

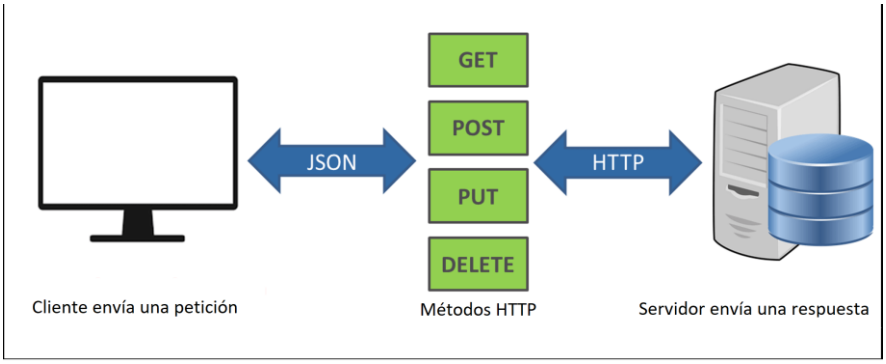


Figura 27: Esquema uso API REST

La última parte para poder obtener el dataset en el formato y forma deseados, es la extracción de los archivos .csv de la base de datos. Para ello, se realiza una consulta a la base de datos que mediante la aplicación hecha con Django y se prepara para servir por http al navegador mediante unas funciones que se encuentran incluidas en el framework. Esto compone una pieza de software que permite traducir las consultas de la base de datos en SQL a código Python, extrayendo de la base de datos los registros e instancia objetos a partir de ellos, que ha permitido ahorrar un trabajo considerable a la hora de tratar con bases de datos.

En este punto del proyecto, ya se tiene el dataset con el que se va a trabajar para entrenar al modelo guardado en una carpeta local con 160 ficheros .csv que corresponden a dos ficheros *NombreSeñal-IQ.csv* y *NombreSeñal-Signal.csv* por cada una de las 80 señales procesadas. Además, se tiene el fichero *Clasificador_señales.xlsx* donde se etiquetan estas 80 señales como se vio en el apartado anterior y en la *Figura 24* y *Figura 25*.

Más adelante, en el apartado de procesado de los datos, se dividirán los 1000 registros de valores de cada una de las señales en 10 sub-señales de 100 registros, dando como resultado un dataset de 800 sub-señales de 100 registros cada una. Esto permitirá tener más datos con los que entrenar el modelo y reducir la ventana de tiempo con la que el modelo podrá hacer las predicciones. De esta manera en vez de tener que analizar 1000 registros de valores por señal, registrará 100, lo que permitirá tener un modelo que necesite de menos registros o muestras temporales para la detección de *jamming* y que el procesado de los datos pueda ser más rápido una vez implementado el modelo en un ecosistema hardware, ya que necesitará de 100 registros de tiempo en vez de 1000.

4.2.2 Pre-procesado de señal

Una vez se han obtenido los datos en formato .csv, como se ha explicado en el apartado anterior, teniendo dos tipos de archivos .csv por cada señal para la extracción de datos de dicha señal y otro archivo para clasificar las señales, se genera un Jupyter Notebook para la extracción y procesado de los datos, llamado "*TFM_1_data_preparation.ipynb*" (*Anexo A1*) para extender el número de datos que se tienen (de 80 señales a 800 sub-señales) para aumentar el dataset que se tiene en el proyecto.

Para ello, se lee cada archivo .csv, *"NombreSeñal-IQ.csv"* y "*NombreSeñal-Signal.csv"*, de cada señal y generar un dataframe con los valores que se quieren procesar. De esta manera se podrá acceder a los valores I, Q y potencia de señal, devolviendo la función un dataframe con 1000 registros (filas) cada uno con 60 valores de I, 60 valores de Q y 1 valor de potencia de señal. En este punto es cuando se dividen las señales de 1000 registros en 10 sub-señales de 100 registros.

Para ello, de estos dataframes se generan y se guardan 10 matrices tipo numpy array de dimensión 100x121. Estas matrices serán usadas para guardar en ellas 100 registros de 60 valores I, 60 valores Q y 1 valor de potencia de señal. Se decide utilizar este formato para generar mapas de características bidimensionales para que el modelo convolucional pueda aprender a detectar patrones en estos datos.

Por tanto, para cada señal de 1000 registros, se han concatenado los valores de I, Q y potencia de señal de cada instante de tiempo y se han dividido en 10 sub-señales de 100 registros para dar lugar a una matriz de "m x n" (donde m = 100 es el número de registros de la sub-señal y n = 121 son los valores de I, Q junto con el valor de potencia de la sub-señal). La representación de estos mapas de datos de se pueden ver a modo de imagen en la *Figura 28*. Con estos datos se generarán los sets de entrenamiento, validación y test para introducir al modelo que se creará para detectar patrones de *jamming* o *no jamming*.

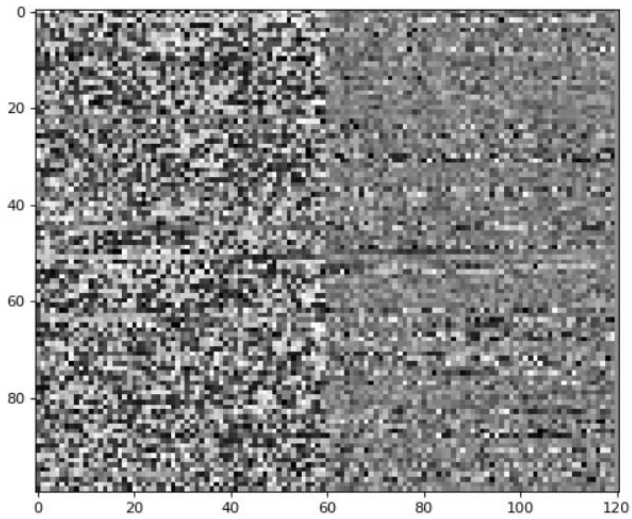


Figura 28: Representación gráfica de un mapa de valores 100x121

4.2.3 Partición de datos

Para realizar la partición de los datos que se han obtenido en el apartado anterior se ha generado un segundo notebook llamado "*TFM_2_data_partition.ipynb*" (*Anexo A2*). En él se ha desarrollado el código encargado de comprobar el balanceo de las clases del problema a raíz de comprobar la frecuencia de las clases que contiene el fichero clasificador *Clasificador_señales.xlsx*, como se puede ver en la *Figura 29*. Se puede ver que se tienen 45 señales clasificadas como *no jamming* y 35 señales clasificadas como *jamming*. Con esto podemos decir que tenemos el dataset lo suficientemente balanceado como para no necesitar recurrir a técnicas adicionales para el balanceo de datos.

```
No jamming    45
Jamming       35
Name: Class, dtype: int64
```

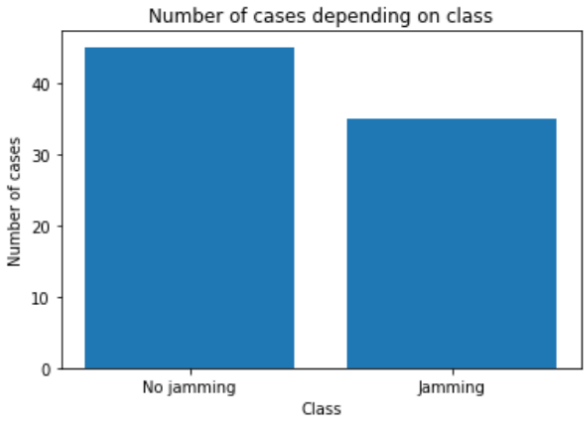


Figura 29: Balanceo de las clases del problema

Se genera un dataframe leyendo del fichero *Clasificador_señales.xlsx* que contiene 2 columnas: NombreSeñal y Clase (*Jamming/No jamming*). Se realiza una selección aleatoria de muestras del dataframe para particionar en un 60% - 40% los datos de entrenamiento y test. Además se fuerza a que el conjunto de test quede balanceado mediante un bucle de validación como se puede ver en la *Figura 30*. De esta manera que quedan 32 señales para test y 48 para entrenamiento y validación.

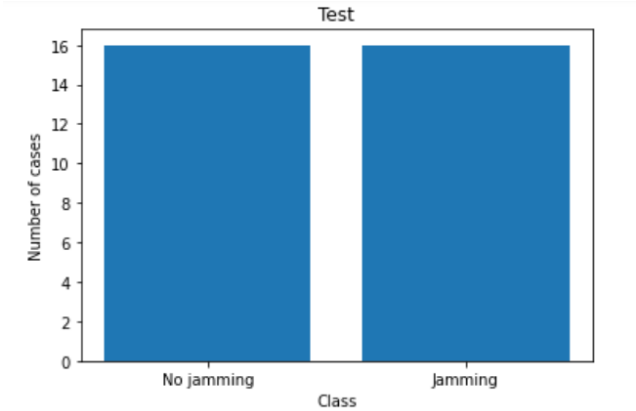


Figura 30: Balanceo de las clases en el conjunto de test

Para dividir las 48 señales de entrenamiento y validación se implementa la técnica de validación cruzada (o cross-validation). La validación cruzada [21] es una técnica que se basa en un proceso iterativo para validar y evaluar un modelo de Inteligencia Artificial, garantizando que los resultados de la validación son independientes del entrenamiento. Consiste en dividir los datos de manera aleatoria en k grupos del mismo tamaño en los que k - 1 grupos se utilizan para entrenar el modelo y el grupo restante para la validación. Esto se realiza k veces como se ve en la *Figura 31* con un caso de k = 4, como se decide hacer en nuestro caso:

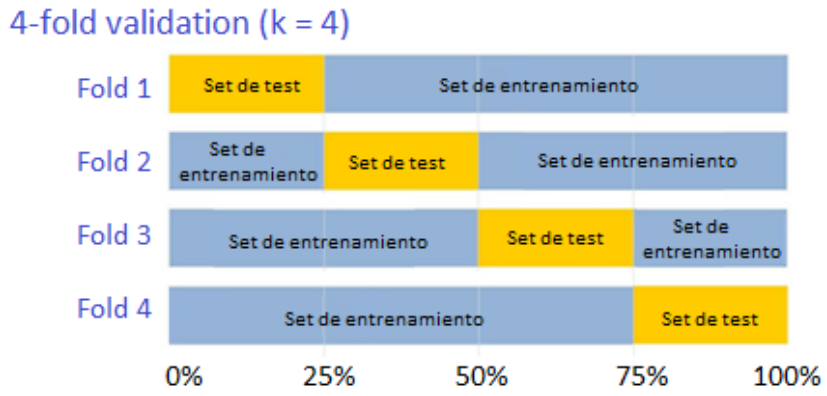


Figura 31: Técnica Cross-validation para k=4

Por tanto, se generan otros 8 datasets correspondientes a las particiones realizadas de los sets de entrenamiento y validación (cuatro de cada uno). Se ha usado la función KFold de la librería de Sklearn, con k = 4, para la validación cruzada de los datos, por lo que tenemos 4 sets de datos de entrenamiento de 36 señales cada uno y 12 señales para la validación, generando 9 dataframes que se guardarán en archivos .csv (uno para test, 4 para entrenamiento y 4 para validación) cuyas cabeceras son el nombre de señal y clase (J*amming*/No *jamming*) teniendo localizada cada señal y clase en su partición correspondiente.

En este punto del trabajo ya se tienen hechas las particiones con los nombres de las 80 señales, unas para test y otras para el entrenamiento y validación. A esto se suman las 800 matrices de datos de dimensiones (100x121) que se han generado en el apartado anterior que se corresponden con los datos de 800 sub-señales 100 registros con los datos de señal. Estas matrices se interpretarán como unos mapas de valores bidimensional que serán las entradas al modelo convolucional.

Para enlazar por un lado las particiones de qué nombres de señal que se han generado y los archivos .npy que tienen las matrices de datos de sub-señales, se generan tres matrices que contienen todas las matrices de 100x121 de cada sub-señal que pertenece a cada conjunto (test, entrenamiento o validación). Quedando por tanto las matrices de:

- Test: 32(señales) x 10(divisiones de señal) x 100(registros) x 121(valores I,Q,Pot)
- Train: 4(csv) x 36(señales) x 10(div señal) x 100(registros) x 121(valores I,Q,Pot)
- Val: 4(csv) x 12(señales) x 10(div señal) x 100(registros) x 121(valores I,Q,Pot)

Se muestra la siguiente tabla (*Tabla 1*) como resumen de las particiones y preparación de los datos que se realizan en este apartado:

	Test	Entrenamiento	Validación
Señales	32	36	12
Sub-señales de 100 registros (.npys)	320	360	120
Particiones (.csv)	1	4	4
Dimensiones matriz final	32x10x100x121	4x36x10x100x121	4x12x10x100x121

Tabla 1: Resumen de las particiones y preparación de los datos

Estas tres matrices finales se guardarán en ficheros .npy separados bajo el nombre *matrix_test.npy*, *matrix_train.npy* y *matrix_valid.npy*, respectivamente. Estos ficheros se corresponderán con los X_train, X_val y X_test con los que se entrenará y pondrá a prueba el modelo clasificador de señales que se implementará en el siguiente apartado. La estructura de carpetas y ficheros utilizados en este proyecto se queda entonces como se muestra en la *Figura 32*:

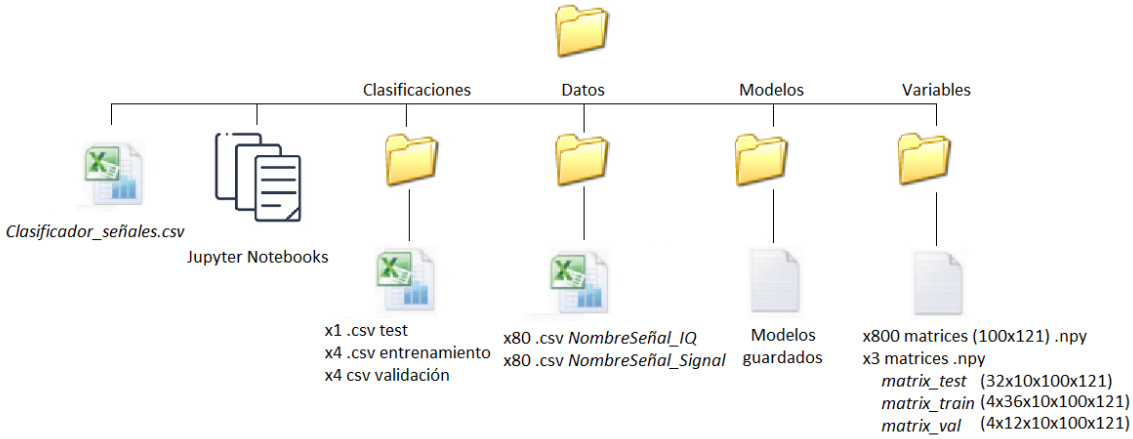


Figura 32: Estructura de carpetas y ficheros del proyecto

Siendo la carpeta */Clasificaciones* la carpeta que aloja los 9 dataframes de las particiones realizadas para *test, train_0… train_3, validation_0… validation_3* en ficheros .csv. La carpeta */Datos* contiene los 160 archivos .csv que se descargaron de la base de datos con los parámetros de señal a procesar. La carpeta */Variables* donde se guardan los 800 ficheros .npy que contienen las matrices de las submuestras de señal con los valores I, Q y potencia de señal y los 3 ficheros *matrix_test.npy*, *matrix_train.npy* y *matrix_valid.npy* donde están las matrices de sub-señales unificadas y clasificadas según la partición que se ha hecho. También se tiene el fichero Excel *Clasificador_señales.xlsl* donde se clasifican las señales y los *notebooks* donde se ejecuta el código de preparación y partición de datos así como la instanciación, entrenamiento y evaluación de los modelos. En la carpeta modelos se guardan los modelos entrenados para su almacenaje y su evaluación.

4.2.4 Algoritmo implementado

Se va a realizar una comparación entre dos estrategias de aprendizaje. Por un lado se entrenarán redes *from scratch* (o modelo hecho desde cero) y por otro lado se entrenarán redes pre-entrenadas utilizando la técnica de *fine tuning*. Se prueban ambas técnicas para tener en cuenta las ventajas e inconvenientes que ofrece cada una. Un modelo pre-entrenado aplicándole *fine tuning*, aunque conlleva un gran número de parámetros entrenables, suele reportar mejores resultados que una red entrenada *from scratch*, ya que permite utilizar los coeficientes de arquitecturas preentrenadas. No obstante, se pretende definir un modelo from scratch donde el número de parámetros entrenables se presente muy reducido con respecto a finetuning y que los resultados de clasificación sean similares a los que se pueden obtener con las redes pre-entrenadas. Se genera un tercer notebook llamado "*TFM_3_model.ipynb*" (*Anexo A3*) para la creación y el entrenamiento del modelo.

Para generar un modelo de Deep Learning *from scratch* que permita clasificar las matrices de datos que se le van a pasar como entrada, se decide utilizar la librería TensorFlow y su API Keras. Se decide utilizar un modelo basado en capas convolucionales para basar el aprendizaje del modelo en aprender a reconocer patrones en base a los vectores de características que saque de los datos que se le introduzcan como entrada.

Se define la longitud para el vector de características en 121 (que se corresponde con los 60 valores I, 60 valores Q y 1 valor de potencia) y se definen las dimensiones de la entrada del modelo como (100x121x1), ya que las entradas al modelo serán unos mapas de datos de 100 registros de 121 parámetros cada una.

La base del modelo (o *base model*) se construye mediante la clase *sequential* de Keras, un modelo al que se le añaden de forma secuencial las siguientes capas:

- Una capa convolucional mediante la clase *Conv2D* a la que se le pasa como parámetro de entrada el definido en el párrafo anterior de (100, 121, 1) y que está compuesta por 32 filtros de 3x3. Se aplica una función de activación ReLu y se le añade un *pool* de dos dimensiones mediante la clase *MaxPooling2D* reduciendo las dimensiones a la mitad.

- Se añaden otras dos capas convolucionales con 64 y 520 filtros de 3x3 con una activación y un *pool* de igual manera que en el párrafo anterior para acabar reduciendo las dimensiones a 25x30x520.

Tras la construcción de la base del modelo, para la parte final del modelo (o *top model*) se decide:

- Como primera aproximación del *top model* se decidió añadir una capa *Flatten* que unidimensualice la entrada multidimensional para interconectar la base del modelo con la implementación de las *fully connected*. De esta manera se pasa de 25x30x520 dimensiones a un vector unidimensional de 101400 valores. Despuñes, una capa *fully connected* empezando por una capa densa para poder conectar cada neurona de una capa con todas las salidas de la capa anterior. Con esta capa,

se aplica una transformación de rotación, escala y traducción a su vector pasando a tener una dimensionalidad de salida de 520. A esta capa se le añade una función de activación ReLu. Finalmente se aplica una capa *Dropout* con un valor de 0.5 para que cada neurona tenga solo el 50% de posibilidad de no activarse durante la fase de entrenamiento y una capa densa con función de activación sigmoide.

En la *Figura 33* se puede ver el sumario del modelo construido en esta primera aproximación.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 98, 119, 32)       320
_________________________________________________________________
activation (Activation)      (None, 98, 119, 32)       0
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 49, 60, 32)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 49, 60, 64)        18496
_________________________________________________________________
activation_1 (Activation)    (None, 49, 60, 64)        0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 25, 30, 64)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 25, 30, 520)       300040
_________________________________________________________________
activation_2 (Activation)    (None, 25, 30, 520)       0
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 13, 15, 520)       0
_________________________________________________________________
flatten (Flatten)            (None, 101400)            0
_________________________________________________________________
dense (Dense)                (None, 520)               52728520
_________________________________________________________________
activation_3 (Activation)    (None, 520)               0
_________________________________________________________________
dropout (Dropout)            (None, 520)               0
_________________________________________________________________
dense_1 (Dense)              (None, 2)                 1042
_________________________________________________________________
activation_4 (Activation)    (None, 2)                 0
=================================================================
Total params: 53,048,418
Trainable params: 53,048,418
Non-trainable params: 0
```

Figura 33: Sumario del modelo (primera aproximación)

Tras ver que el modelo de esta manera tiene alrededor de 53 millones de parámetros y una de las expectativas del proyecto es que se pueda implementar este detector de *jamming* en tiempo real, se decidió reducir el número de parámetros para aumentar la velocidad del modelo implementando en el *top model*:

- Una capa *Global_max_pooling2D* como sustituto a la capa *Flatten* para unidimensionalizar los parámetros de entrada a la capa. De esta manera se pasa de 25x30x520 dimensiones a un vector unidimensional de 520 valores

- Finalmente se le añade una capa densa donde le pasamos como parámetro el número de clases a determinar (2) y una función de activación sigmoide para clasificar binariamente los mapas de datos que entren al modelo.

Con ello se ha pasado de tener un modelo de más de 53 millones de parámetros entrenables a un total de 319.898 parámetros, consiguiendo un modelo mucho más eficiente en términos de performance y rapidez de procesamiento, lo cual será imprescindible para un futuro en el que este modelo reciba datos reales para su procesado. En la *Figura 34* se puede ver el sumario del modelo construido que se da por óptimo.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_24 (Conv2D)           (None, 98, 119, 32)       320
_________________________________________________________________
activation_32 (Activation)   (None, 98, 119, 32)       0
_________________________________________________________________
max_pooling2d_20 (MaxPooling (None, 49, 60, 32)        0
_________________________________________________________________
conv2d_25 (Conv2D)           (None, 49, 60, 64)        18496
_________________________________________________________________
activation_33 (Activation)   (None, 49, 60, 64)        0
_________________________________________________________________
max_pooling2d_21 (MaxPooling (None, 25, 30, 64)        0
_________________________________________________________________
conv2d_26 (Conv2D)           (None, 25, 30, 520)       300040
_________________________________________________________________
activation_34 (Activation)   (None, 25, 30, 520)       0
_________________________________________________________________
global_max_pooling2d_12 (Glo (None, 520)               0
_________________________________________________________________
dense_10 (Dense)             (None, 2)                 1042
_________________________________________________________________
activation_35 (Activation)   (None, 2)                 0
=================================================================
Total params: 319,898
Trainable params: 319,898
Non-trainable params: 0
_________________________________________________________________
None
```

Figura 34: Sumario final del modelo from scratch

De esta manera se consigue que se propague información relativa a las características morfológicas del mapa de valores que se introduce, para que sea capaz de detectar patrones y conseguir diferenciar entre un mapa que corresponde con una señal limpia a una con *jamming* con una reducción de parámetros significativa con respecto al primer modelo creado. La representación gráfica de cómo está diseñado el modelo definitivo y cómo varían las dimensiones de los datos de entrada de cada capa se puede ver con un diagrama en la *Figura 35*.

Por último, el diagrama completo de la arquitectura del modelo final se puede ver en el *Anexo A4*.

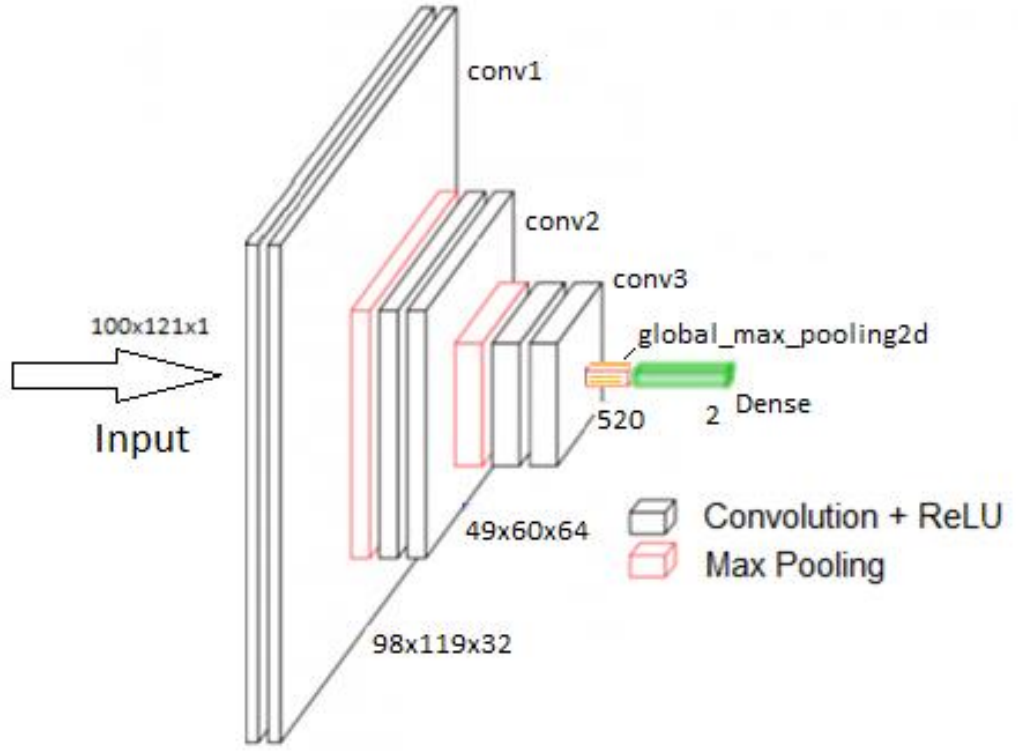


Figura 35: Diseño final del modelo from scratch

Por otro lado, para comprobar la calidad del modelo que se ha implementado la técnica de *fine tuning*. Para ello, se ha utilizado el modelo pre-entrenado VGG16. Este es un modelo de red neuronal convolucional (CNN) que se compone de 5 bloques convolucionales como se puede ver en la *Figura 36* pre-entrenados con un set de imágenes llamado "*imagenet*".

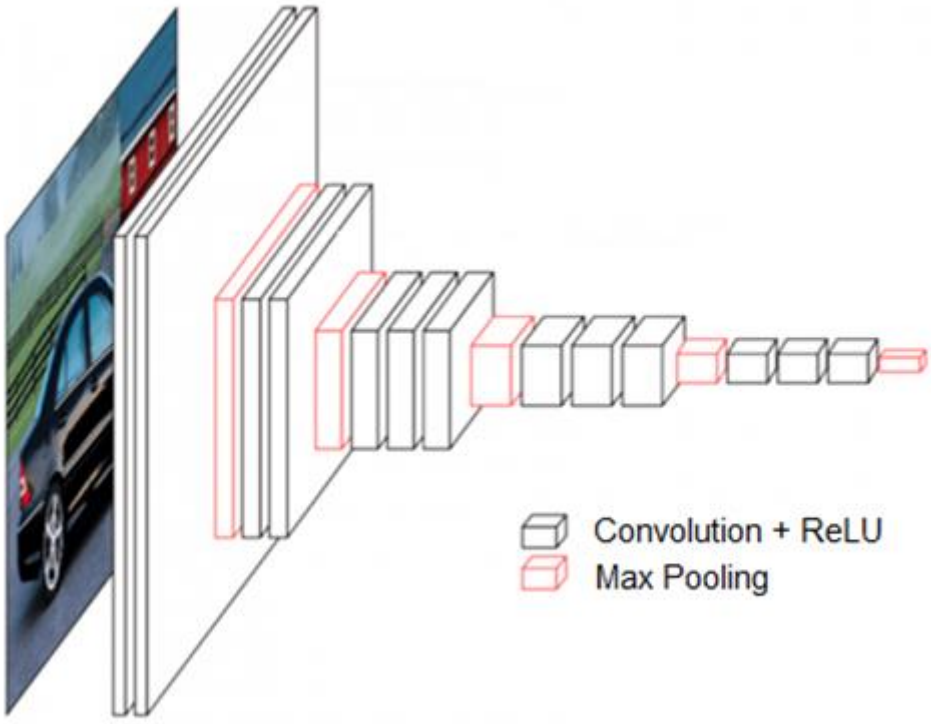


Figura 36: Arquitectura modelo base VGG16

Como *base model* se tienen los 5 bloques convolucionales predefinidos que se pueden ver en la *Figura 37* para imágenes (o mapas de datos) de dimensión 100x121x3, ya que este modelo solo funciona con imágenes tridimensionales (o RGB). Para ello se copiará tres veces para las tres dimensiones los mapas bidimensionales de dimensión 100x121x1 que se tienen como datos para el entrenamiento de los modelos para obtener mapas de dimensiones 100x121x3 en las tres dimensiones que requiere el modelo.

Como *top model*, para cuadrar con el modelo propuesto que se implementó en el apartado anterior, se implementa una capa *global_max_pooling2d* y una capa densa activada por una función sigmoide. Esto hace que se tenga un modelo con un total de 14.715.714 parámetros, de los cuales dejamos congelados 14.714.688 que corresponden con las capas convolucionales del modelo base y 1.026 parámetros entrenables como se puede ver en la *Figura 37*.

Layer (type)	Output Shape	Param #
input_12 (InputLayer)	(None, 100, 121, 3)	0
block1_conv1 (Conv2D)	(None, 100, 121, 64)	1792
block1_conv2 (Conv2D)	(None, 100, 121, 64)	36928
block1_pool (MaxPooling2D)	(None, 50, 60, 64)	0
block2_conv1 (Conv2D)	(None, 50, 60, 128)	73856
block2_conv2 (Conv2D)	(None, 50, 60, 128)	147584
block2_pool (MaxPooling2D)	(None, 25, 30, 128)	0
block3_conv1 (Conv2D)	(None, 25, 30, 256)	295168
block3_conv2 (Conv2D)	(None, 25, 30, 256)	590080
block3_conv3 (Conv2D)	(None, 25, 30, 256)	590080
block3_pool (MaxPooling2D)	(None, 12, 15, 256)	0
block4_conv1 (Conv2D)	(None, 12, 15, 512)	1180160
block4_conv2 (Conv2D)	(None, 12, 15, 512)	2359808
block4_conv3 (Conv2D)	(None, 12, 15, 512)	2359808
block4_pool (MaxPooling2D)	(None, 6, 7, 512)	0
block5_conv1 (Conv2D)	(None, 6, 7, 512)	2359808
block5_conv2 (Conv2D)	(None, 6, 7, 512)	2359808
block5_conv3 (Conv2D)	(None, 6, 7, 512)	2359808
block5_pool (MaxPooling2D)	(None, 3, 3, 512)	0
global_max_pooling2d_19 (Glo	(None, 512)	0
dense_17 (Dense)	(None, 2)	1026

Total params: 14,715,714
Trainable params: 1,026
Non-trainable params: 14,714,688

VGG16 base model

Figura 37: Sumario del modelo VGG16

5. RESULTADOS

Tras haber implementado el modelo para poder analizar y detectar patrones de los mapas de valores que se le introducen, se realizan una serie de experimentos y pruebas para conseguir entrenar al modelo de manera que consiga una buena precisión a la hora de detectar los patrones que definan si una señal es limpia o está alterada.

Antes de nada, como cada señal se subdividió en 10 subseñales, se genera un nuevo dataframe a partir del fichero clasificador de señales *Clasificador_señales.xlsx* del cual por cada señal, se generan 10 etiquetas con la misma clasificación, consiguiendo así unas listas que convertimos en categóricas *y_train*, *y_validation* e *y_test*¸ en las que '0' será "*No Jamming*" y '1' "*Jamming*" quedando acorde a la subdivisión de las señales que se usan en *X_train*, *X_validation* y *X_test*.

5.1 Experimentos

Teniendo el modelo implementado y las particiones de los datos hechas, se decide utilizar el modelo implementado en el apartado anterior con diferentes optimizadores para comprobar los distintos resultados que nos pueden ofrecer dichos optimizadores. Estos procesos de entrenamiento se pueden ver en el notebook "*TFM_3_model.ipynb*" (*Anexo A3*). Se utilizan para probar distintas aproximaciones los optimizadores de:

- Descenso estocástico de gradiente (o **SGD** por sus siglas en inglés por *Stochastic Gradient Descent*). Este es el método básico de optimización de redes neuronales. La actualización de los pesos sigue el siguiente algoritmo: $W = W - \alpha * \Delta W$ (donde W es la matriz de pesos, α el *Learning Rate* (o ratio de aprendizaje) y ΔW la derivada de la matriz de pesos.

- **Adam** es un método de descenso de gradiente estocástico que se basa en la estimación adaptativa de momentos de primer y segundo orden. El método es computacionalmente eficiente, tiene pocos requisitos de memoria y es invariante al reajuste diagonal de gradientes. A continuación se muestra su implementación:

$$\nu_t = \beta_1 * \nu_{t-1} - (1 - \beta_1) * g_t$$
$$s_t = \beta_2 * s_{t-1} - (1 - \beta_2) * g_t^2$$
$$\Delta\omega_t = -\eta \frac{\nu_t}{\sqrt{s_t + \epsilon}} * g_t$$
$$\omega_{t+1} = \omega_t + \Delta\omega_t$$

Siendo n el *Learning Rate* inicial, g_t el gradiente en tiempo t v_t el promedio exponencial de gradientes a lo largo de w y s_t el promedio exponencial de cuadrados de gradientes a lo largo de w.

- **Adadelta** es otro método de descenso de gradiente estocástico que se basa en la tasa de aprendizaje adaptativo por dimensión, adaptando las tasas de aprendizaje en función de una ventana móvil de actualizaciones de gradientes. Este es el optimizador que mejor resultado ha dado en este proyecto. La inicialización del algoritmo de optimización es la siguiente:

$$E[g^2]_0 := 0 \text{ (Inicializa vector de momento de segundo orden de gradiente)}$$

$$E[\Delta x^2]_0 := 0 \text{ (Inicializa actualización de variable de segundo orden de gradiente)}$$

$$t := t + 1$$

$$E[g^2]_t := \rho * E[g^2]_{t-1} + (1 - \rho) * g^2$$

$$\Delta x_t = -RMS[\Delta x]_{t-1} * g_t / RMS[g]_t$$

$$E[\Delta x^2]_t := \rho * E[\Delta x^2]_{t-1} + (1 - \rho) * \Delta x_t^2$$

$$x_t := x_{t-1} + \Delta x_t$$

Donde RMS representa la raíz cuadrada media del gradiente. Y ε una constante utilizada para un mejor acondicionamiento del gradiente. $RMS[g_t] = \sqrt{E[g^2]_t + \epsilon}$

Junto a esos tres optimizadores se prueban distintos valores de *Learning Rate* utilizando métodos heurísticos, probando valores de 10^{-4} hasta 10^{-2} para el entrenamiento del modelo. Además, se utiliza la función *binary_cross_entropy* como función de pérdidas.

Tras varias pruebas, en las que se decide utilizar 500 épocas por cada iteración (4 iteraciones por los 4 sets de entrenamiento y validación que se crearon en el apartado de partición de datos) y una *Callback EarlyStopping* para pasar de iteración si el valor de pérdidas del set de validación no cambia en 30 épocas. Por último, se utiliza un batch_size de 32 para que el set de entrenamiento se introduzca en el modelo en lotes de 32 mapas de valores (o matrices). Para compilar el modelo se utiliza la función de pérdidas "*binary_crossentropy*".

Con esto, se entrenan tanto el modelo propuesto en este trabajo *from scratch* (el modelo hecho desde cero propuesto en el apartado anterior) como el modelo VGG16 para comparar la eficacia, precisión y performance de cada uno.

Para finalizar, se guardan los modelos mencionados una vez son entrenados para ser evaluados en el notebook "*TFM_4_model_evaluation.ipynb*" (*Anexo A5*). En los siguientes apartados se mostrarán los resultados cualitativos y cuantitativos de la evaluación de los modelos una vez entrenados y se discutirá cuál es la aproximación más óptima para la resolución de este TFM.

5.2 Resultados cualitativos

Para mostrar los resultados cualitativos del proyecto se enseñan las curvas de precisión y pérdidas de los modelos entrenados.

Tras varias pruebas de entrenamiento del modelo con los optimizadores mencionados anteriormente, se aprecia que el optimizador Adadelta es el que mejores resultados obtiene según las curvas de precisión y de pérdidas. Se ha decidido mostrar los resultados del mejor entrenamiento, que se corresponde con el modelo entrenado con el optimizador Adadelta y un *Learning rate* de 0.001.

Aportando la experiencia de las pruebas realizadas, el optimizador SGD es un optimizador que funciona bien y que es estable en sus pasos por cada época y que consigue casi un 100% de precisión en entrenamiento y validación. A pesar de que en alguna de las pruebas realizadas no conseguía superar el 70% de precisión, en la gran mayoría se ha desarrollado con resultados satisfactorios.

Por otro lado, el optimizador Adam es el que peores resultados consiguió. Variando mucho los valores de *Learning rate* en cada prueba, se consiguió sacar alguna situación en la que la precisión conseguía valores aceptables, pero la mayoría de las pruebas no conseguían una precisión mayor del 60-70%.

El optimizador Adadelta es sin duda el optimizador que mejor ha funcionado, consiguiendo un 100% de precisión y con unas pérdidas muy bajas para todas las pruebas realizadas. Por este motivo, el modelo entrenado con este optimizador se ha considerado el modelo definitivo.

Se muestran a continuación los resultados del entrenamiento con el optimizador que mejor resultados ha dado (Adadelta) con el *Learning rate* que mejor resultados ha sacado, con valor 0.001, para los entrenamientos del *cross validation* (*Figura 38*) y del modelo final (*Figura 39*).

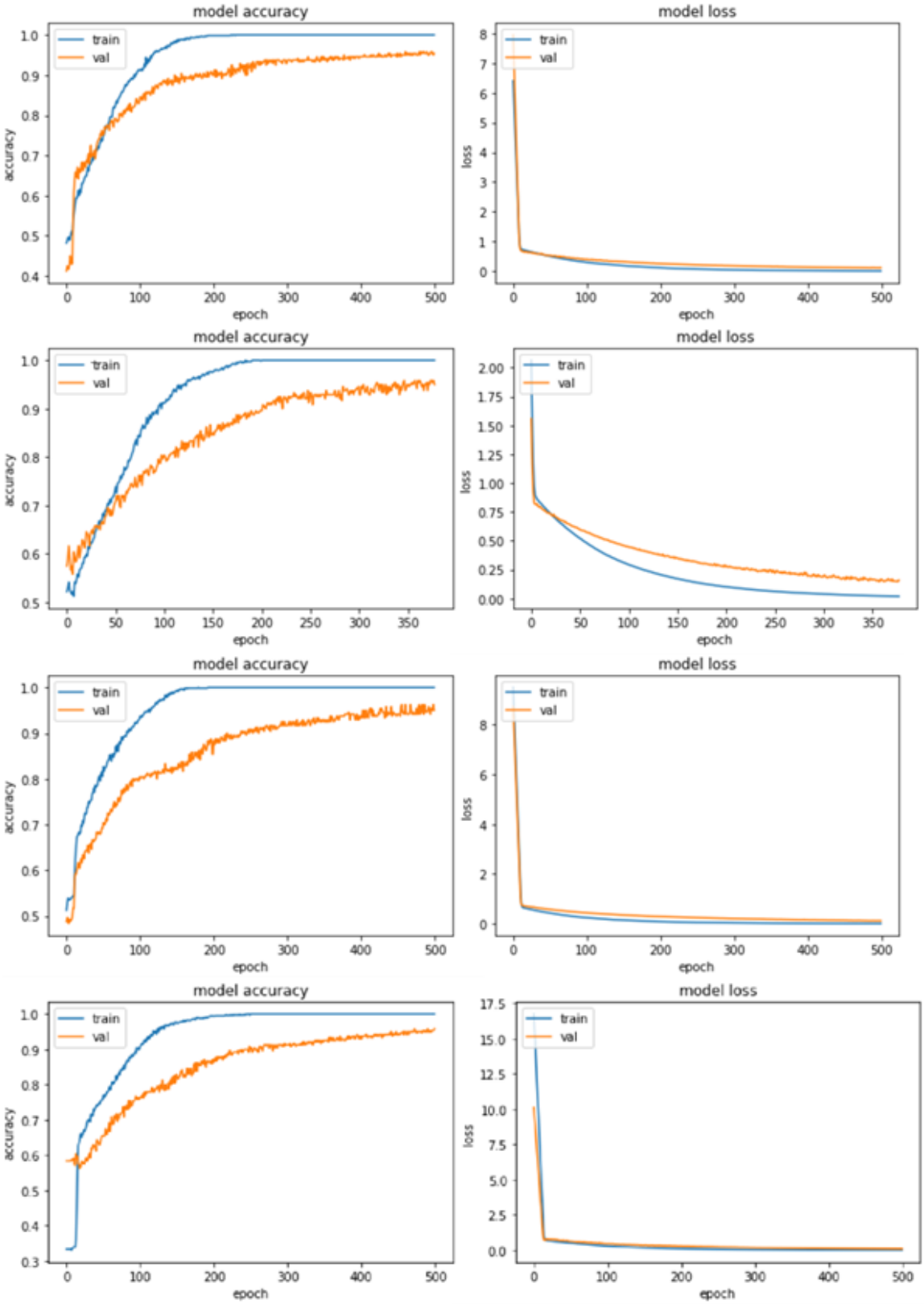


Figura 38: Curvas de precisión y pérdidas del modelo from scratch, opt Adadelta, lr = 0.001 (cross validation)

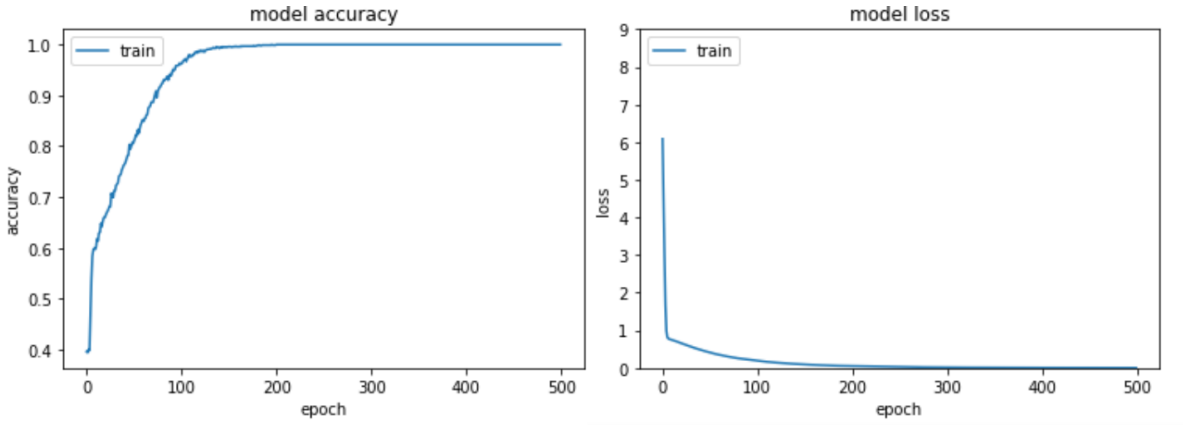


Figura 39: Curvas de precisión y pérdidas del modelo from scratch, opt Adadelta, lr = 0.001 (final)

Para realizar una comparativa, se realizan los mismos experimentos para el modelo VGG16 al que se le aplica la técnica de *fine tuning*. Se muestran las curvas de precisión y pérdidas del modelo pre-entrenado VGG16, el cual se entrena con el set de datos del proyecto al que se ha congelado su modelo base para que no cambien los pesos, junto con las capas *fully connected* implementadas en el apartado 5.1 que serán las que se puedan entrenar. Se utiliza un *Learning Rate* de 0.01 tras observar los resultados de los experimentos del *cross validation* y por último se entrena el modelo con todos los datos de entrenamiento obteniendo las curvas de precisión y pérdidas que se pueden ver en la *Figura 40*:

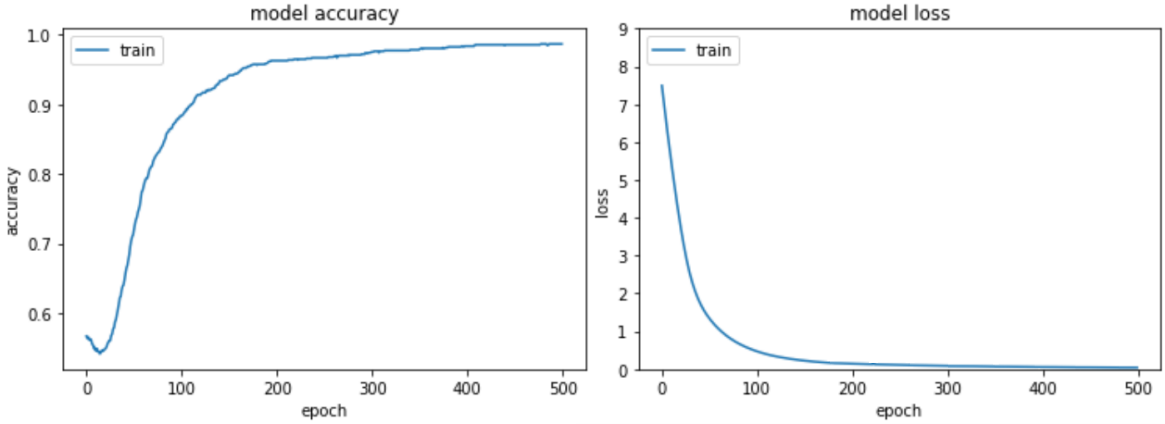


Figura 40: Curvas de precisión y pérdidas del modelo pre entrenado, opt Adadelta, lr = 0.01 (final)

Con estas curvas podemos ver que ambos modelos adquieren valores de precisión altos, llegando al 100%. Lo mismo pasa con las curvas de pérdidas, que se ve cómo cada época van disminuyendo en ambos modelos de forma muy similar. En los siguientes apartados se verán los resultados cuantitativos que definan lo bueno que es cada modelo.

5.3 Resultados cuantitativos

Tras las pruebas y los entrenamientos mencionados, obtenemos que el modelo entrenado con el optimizador Adadelta con un *Learning rate* de 0.001 es el que mejores resultados obtuvo, llegando a obtener en el entrenamiento una precisión con el set de entrenamiento y de validación del 100%.

Para poder analizar los resultados de manera cuantitativa, se utiliza el notebook "*TFM_4_model_evaluation.ipynb*" (*Anexo A5*) cargando el modelo hecho *from scratch* y se evalúa con el conjunto de test utilizando el método *evaluate()* obteniendo los resultados de precisión que se pueden ver en la *Figura 41* tardando en procesar cada muestra 2ms:

```
320/320 [==============================] - 1s 2ms/sample - loss: 0.1400 - acc: 0.9547
adadelta_model_lr_0.001:
accuracy: 95.94%
```

Figura 41: Precisión del modelo from scratch adadelta_model_lr_0.001

Por otro lado, se saca su matriz de confusión que se puede ver en la *Figura 42*:

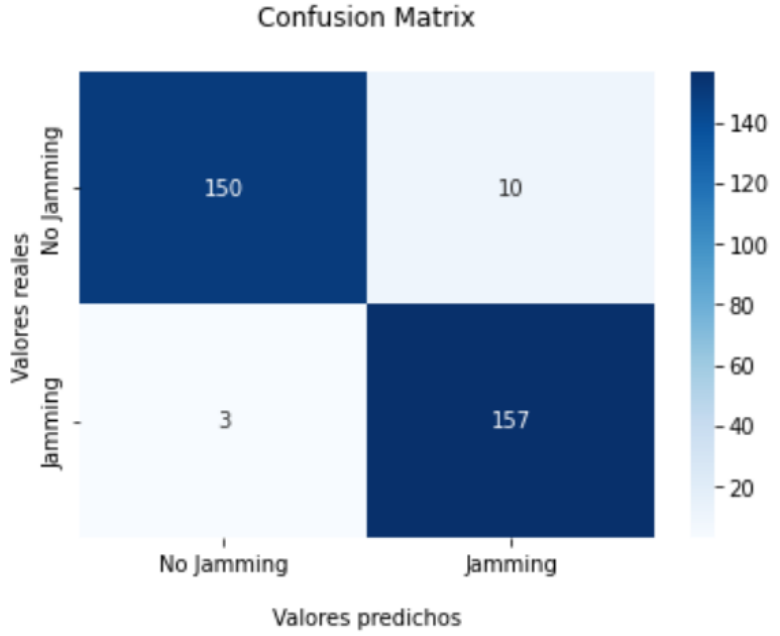


Figura 42: Matriz de confusión modelo from scratch adadelta_model_lr_0.001

Para analizar cuantitativamente los resultados, se van a analizar tres variables. Por un lado la precisión del modelo, el *recall* y el resultado F1.

La precisión describe cómo de bien el modelo puede predecir las etiquetas correctamente. Su fórmula es la siguiente:

$$Precision = \frac{TruePositive}{TruePositive + FalsePositive}$$

Recall describe cómo el modelo puede recuperar todas las etiquetas correctamente. Su fórmula es la siguiente:

$$Recall = \frac{TruePositive}{TruePositive + FalseNegative}$$

La puntuación F1 es la proporción entre la multiplicación de precisión y recuperación dividida por la suma de precisión y recuperación y luego ponderada por 2. Esta métrica combina precisión y recuperación. Si el valor aumenta, el modelo mejor es.

$$F1 = 2 * \frac{precision * recall}{precision + recall}$$

Se sacan los resultados a partir de la librería de *sklearn* usando el método *classification_report()*, obteniendo así los resultados que se pueden ver en la Figura 43, siendo '0' la correspondencia a la etiqueta "*No jamming*" y '1' a la de "*Jamming*":

	precision	recall	f1-score	support
0	0.98	0.94	0.96	160
1	0.94	0.98	0.96	160
accuracy			0.96	320
macro avg	0.96	0.96	0.96	320
weighted avg	0.96	0.96	0.96	320

Figura 43: Resultados precisión, recall y puntuación F1 del modelo from scratch adadelta_model_lr_0.001

Con esta información podemos concluir con que los resultados de precisión del modelo que se ha entrenado en este trabajo son muy satisfactorios, teniendo una precisión general del 95.94% con una performance rápida, en la que se tardan 2ms en evaluar una muestra. Estos resultados fueron similares a los obtenidos utilizando la primera aproximación del modelo (mencionado en el apartado 4.2.2 Algoritmo implementado) con una velocidad de procesado mucho mayor, por lo que se decide hacer definitivo este modelo, pues la velocidad de procesado es un parámetro crítico en la resolución del problema.

Para comprobar cómo de bueno es el modelo en comparación con el modelo pre-entrenado VGG16 al que le hemos re-entrenado utilizando la técnica de *fine tuning* volvamos a realizar los mismos resultados. Para ello volveremos a evaluar el modelo, sacar su matriz de confusión y sus resultados de precisión, *recall* y puntuación F1.

Para comprobar cómo de bueno es el modelo en comparación con el modelo pre-entrenado VGG16 al que se le ha aplicado el *fine tuning*, se obtienen los mismos resultados evaluando dicho modelo de la misma manera que se evaluó el modelo *from scratch*:

```
320/320 [==============================] - 5s 16ms/sample - loss: 0.2296 - acc: 0.9563
vgg_adadelta_model_lr_0.01:
acc: 95.63%
```

Figura 44: Precisión del modelo pre-entrenado adadelta_modelVGG16_lr_0.001

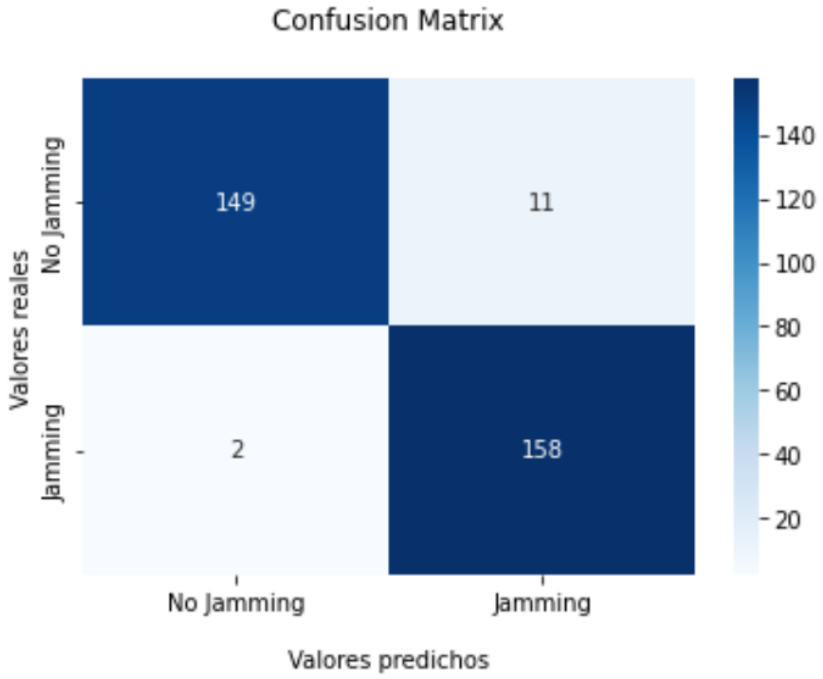


Figura 45: Matriz de confusión modelo pre-entrenado adadelta_modelVGG16_lr_0.001

	precision	recall	f1-score	support
0	0.99	0.93	0.96	160
1	0.93	0.99	0.96	160
accuracy			0.96	320
macro avg	0.96	0.96	0.96	320
weighted avg	0.96	0.96	0.96	320

Figura 46: Resultados precisión, recall y puntuación F1 del modelo pre-entrenado adadelta_modelVGG16_lr_0.001

Con esta información podemos afirmar que los resultados de precisión del modelo VGG16 son muy satisfactorios, llegando a adquirir un 99% de precisión en la detección de señales no alteradas y un 93% en las alteradas.

En el siguiente apartado se analizarán y compararán los resultados de ambos modelos.

6. DISCUSIÓN

Tras observar los resultados cuantitativos y cualitativos, se procede a analizar si el modelo diseñado en este proyecto es un modelo eficiente y más válido para su puesta en producción para resolver el problema de clasificación de señales que un modelo pre entrenado al que se le ha efectuado la técnica del *fine tuning*. Para ello, veremos una comparativa en la *Tabla 2* entre sus parámetros y los resultados que han ofrecido en su evaluación en la siguiente tabla con los valores más significativos.

----------	Modelo from scractch	Modelo VGG16
Precisión (general)	95.94%	95.63%
Precisión (por clases)	98% (No jamming) 94% (Jamming)	99% (No jamming) 93% (Jamming)
Recall	94% (No jamming) 98% (Jamming)	93% (No jamming) 99% (Jamming)
Puntuación F1	96% (No jamming) 96% (Jamming)	96% (No jamming) 96% (Jamming)
Número de parámetros	319.898	14.714.688
Velocidad de procesamiento (Jupyter Notebook)	2 ms/muestra (subseñal 100x121)	16 ms/muestra (subseñal 100x121)

Tabla 2: Comparativa resultados y valores modelo from scratch vs VGG16 con fine tuning

Comparando los resultados, se puede ver que ambos modelos tienen una precisión similar, lo que les hace a ambos modelos potenciales para la resolución del problema. Si bien, la velocidad de procesado de 16ms/muestra del modelo VGG16 con respecto a 2ms/muestra del modelo diseñado es un factor que hace que el modelo diseñado *from scratch* en este proyecto sea una aportación interesante, pues permitirá poder analizar más muestras en menos tiempo, pudiendo detectar antes el *jamming* y permitiendo que la ventana de muestras a analizar para un mismo tiempo de procesado sea mayor permitiendo procesar más datos en menos tiempo que con el modelo VGG16.

Por tanto, gracias a la velocidad de procesamiento del modelo diseñado en este trabajo, se dispondrá de un modelo capaz de procesar muestras de señal en un tiempo suficientemente corto como para poder implementar ventanas de detección que validen cada X muestras si la señal que se está analizando está alterada o no, implementando de esta manera una protección a falsos positivos/negativos.

En la *Figura 47* se puede apreciar las diferencias entre los valores que se han obtenido de los dos modelos.

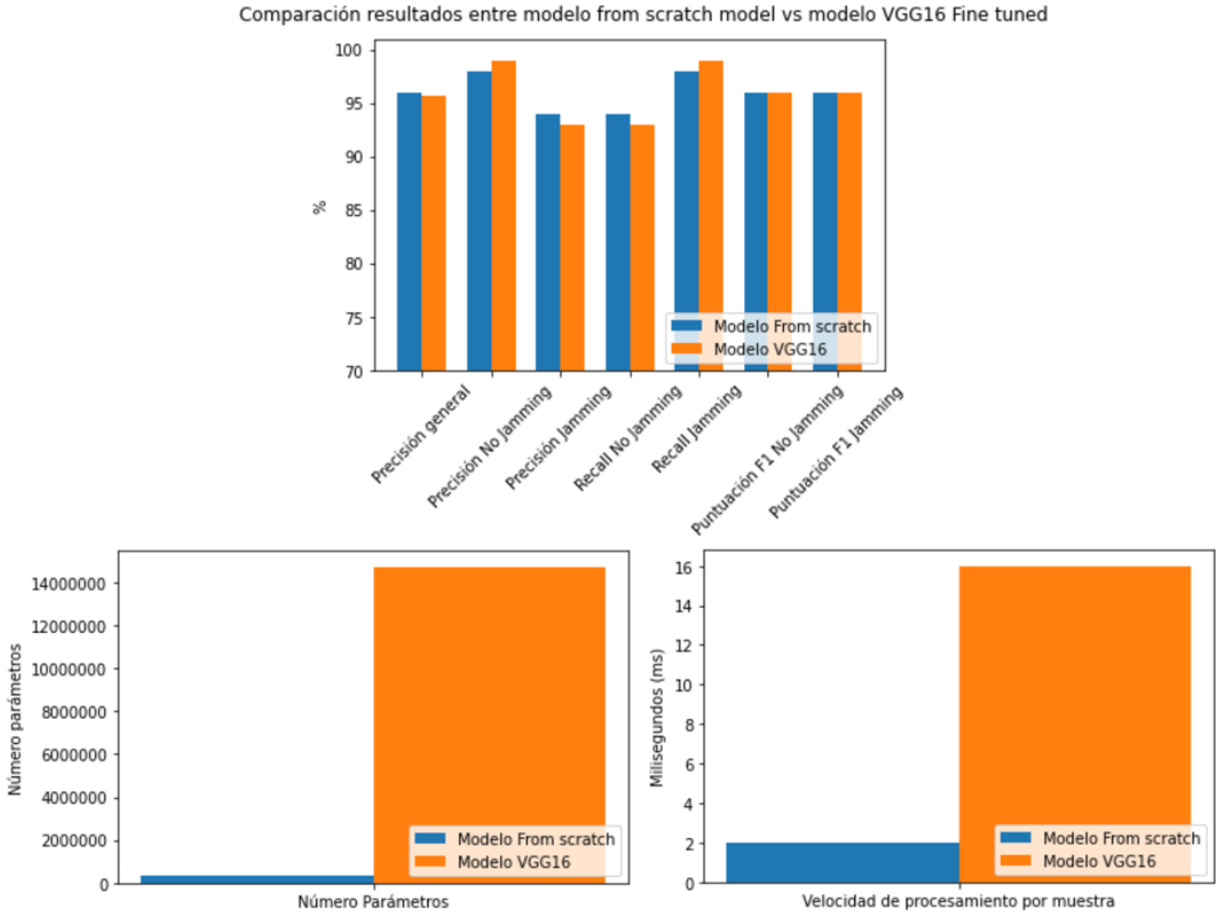


Figura 47: Comparación resultados y parámetros modelo from scratch vs modelo VGG16

Se concluye por tanto que para el problema que se está tratando, el modelo creado *from scractch* es un modelo mucho más adecuado que el VGG16 entrenado, pues la precisión del modelo es similar pero la diferencia de parámetros, y por tanto, velocidad de procesamiento de señales, son mucho mejores en el modelo *from scratch*, característica que se considera vital en la situación que atañe el problema.

7. CONCLUSIONES Y LÍNEAS FUTURAS

En este TFM se ha realizado un proyecto completo de generación y adquisición de datos, generación de base de datos, procesado de datos, partición de datos, diseño de modelo de Deep Learning, entrenamiento, validación y test del modelo con los datos y comparación del modelo diseñado y entrenado con respecto a un modelo pre-entrenado al que se le ha aplicado la técnica *fine tuning*.

Gracias a esto se ha podido desarrollar un proyecto de Machine Learning mediante Aprendizaje Supervisado utilizando un algoritmo con técnicas de Deep Learning desde cero, permitiendo generar todos los recursos necesarios para la resolución del problema y entender y participar en todas las fases de un proyecto de esta categoría. Lo más crítico en un proyecto de este tipo es el cómo se prepara y se pasan los datos al modelo para entrenarse, lo cual es el punto fuerte de este trabajo, permitiendo tener un ecosistema de generación de datos propio y controlado que permita aumentar en el futuro el número de casos y poder dotar al modelo de mayor conocimiento para etapas futuras.

Con este trabajo se ha conseguido resolver el problema que se planteaba al conseguir un modelo de Inteligencia Artificial capaz de clasificar las señales que se le introducen en las categorías de *Jamming* o *No Jamming* con una velocidad de procesado mucho mayor que la de un modelo pre-entrenado y con una precisión similar. De esta manera, se podrá implementar este modelo en un entorno real en el que el tiempo de procesado es crítico.

En cuanto a la metodología del trabajo y la calidad del resultado final, se puede considerar que se ha conseguido llegar a un resultado muy satisfactorio habiendo realizado todos los procesos necesarios de manera correcta. Esto permite poder aumentar las dimensiones del problema de manera sencilla ya que se ha seguido un orden y una metodología de generación de datos, volcado de datos en una base de datos, extracción, procesado y partición de los datos para el entrenamiento del modelo que es modular y con capacidad de crecimiento y modificación para soportar un aumento de las pretensiones de este proyecto.

Como última conclusión, se puede decir que este proyecto es un acercamiento académico a un problema del mundo real que puede ser una futura línea de investigación de la empresa utilizando Inteligencia Artificial para detectar alteraciones de señal con unos resultados iniciales muy satisfactorios.

A continuación, se plantean las líneas futuras del proyecto para determinar las posibilidades de crecimiento en dimensiones o en complejidad:

1. Implementar el modelo entrenado en una FPGA mediante, por ejemplo, una placa PYNQ. Esta placa está diseñada para usarse con el marco de código abierto PYNQ que permite a los programar un sistema en chip (o *SoC*) integrado con Python, implementando el modelo en una placa que permita introducirse en el ecosistema Hardware y Software donde se quiera introducir para poner a prueba el modelo actual en un entorno real.

2. Aumentar el número de casos pasando de tener 80 señales (u 800 subseñales) a un número mucho mayor para tener un modelo mucho mejor entrenado.

3. Refinar el modelo de tal manera que pueda ser de detección multiclase, permitiendo no solamente conseguir detectar si hay *Jamming* o no, sino también aumentar los posibles casos consiguiendo detectar no solo para señales BPSK, sino también para el resto de señales de radiofrecuencia PSK definidas en el apartado 1, como las QPSK o 8PSK. Para ello se deberán generar y procesar las nuevas señales de la misma manera que se han hecho en este trabajo, generar el nuevo fichero de etiquetas para cada señal procesada y crear y entrenar un nuevo modelo que permita la clasificación multiclase.

4. Determinar varios tipos de alteración de señal no solamente centrándose en *Jamming* o *No Jamming*, sino también detectando qué tipo de *jamming* es y pudiendo determinar si hay *Jamming* de un tono, multibanda, si es de amplitud menor o mayor que la señal original, etc. Para ello se puede crear un modelo con diferentes salidas para que pueda determinar no solamente el tipo de señal (como hemos visto en el párrafo anterior) sino también el tipo de *Jamming*. Para ello se puede implementar un modelo *MultiOutput* y *MultiLayer* que sea capaz de identificar por un lado qué tipo de señal es y por otro qué tipo de *Jamming* tiene o si no tiene *Jamming*.

5. Por último, como una posibilidad de llevar el proyecto a un siguiente nivel, se plantea la posibilidad de crear un sistema de redes generativas adversas (o GAN por sus siglas en inglés *Generative Adverse Network*) de tal manera que se acabe creando un sistema multiagente que compita entre sí. Por un lado, un agente se debe encargar de generar una señal limpia que y generarle un *Jamming* en un momento determinado o aleatorio durante una ventana de tiempo y, por otro lado, otro agente se encargue de recibir el resultado de esta señal y detectar si hay *Jamming* o no. Para ello se debería crear primero un modelo que aprenda a generar señales con y sin *Jamming* y luego utilizar un modelo para detectarlas. De esta manera se puede crear un sistema de recompensas en las que el modelo generador reciba recompensa por cada ventana de tiempo en la que genere *Jamming* y pierda cada vez que se lo detectan, mientras que el otro modelo pierda puntos por cada ventana de tiempo que no está detectando *Jamming* o cada vez que detecte mal. Con esto se podría conseguir un proyecto muy interesante en el que no solamente se tenga a una red entrenándose para detectar *Jamming* cada vez más complejo, sino que otra red se entrenará para poder generar interferencias cada vez más enrevesadas que permitan tener un sistema de defensa-como-ataque con técnicas no vistas hasta ahora.

Con esto se da por terminado este Trabajo de Fin de Máster, en el que se ha realizado un ciclo de trabajo completo, pasando por todas sus fases, para un proyecto de clasificación con Inteligencia Artificial utilizando Machine Learning y Deep Learning y en el que se han dejado fijadas las líneas futuras de investigación y de desarrollo del proyecto.

BIBLIOGRAFÍA

[1] Alfred O. Hero (2014). *The radio-frequency spectrum*. *Encyclopædia Britannica*

[2] Diccionario Español de Ingeniería - Definición de Interferencia intencionada https://diccionario.raing.es/es/lema/interferencia-intencionada (*consultado en Mayo de 2022*)

[3] John G. Proakis (1995) - *Digital Communications*.

[4] Matthew Gast (2005). *802.11 Wireless Networks: The Definitive Guide. Vol. 1 (2 ed.)*

[5] Wikipedia - Definición de Red Neuronal Convolucional https://es.wikipedia.org/wiki/Red_neuronal_convolucional (*consultado en Mayo de 2022*)

[6] Fran Ramírez (2018). Historia de la IA: Frank Rosenblatt y el Mark I Perceptrón, el primer ordenador fabricado específicamente para crear redes neuronales en 1957

[7] Wikipedia - Definición de retropropagación https://es.wikipedia.org/wiki/Propagaci%C3%B3n_hacia_atr%C3%A1s (*consultado en Mayo de 2022*)

[8] Y.J. Cruz; M. Rivas; R. Villalonga Quiza; R.E Haber.; Beruvides, G. (2021) - *Ensemble of convolutional neural networks based on an evolutionary algorithm applied to an industrial welding process*

[9] Bryan Medina - Procesamiento digital de imágenes e Inteligencia Artificial. https://bryanmed.github.io/conv2d/ *(consultado en Mayo de 2022)*

[10] Victor Zhou - *CNNs, Part 1: An Introduction to Convolutional Neural Networks* https://victorzhou.com/blog/intro-to-cnns-part-1/ (*consultado en Mayo de 2022*)

[11] Vandit Jain - *Everything you need to know about "Activation Functions" in Deep learning models* https://towardsdatascience.com/everything-you-need-to-know-about-activation-functions-in-deep-learning-models-84ba9f82c253 (*consultado en Mayo de 2022*)

[12] Fundamentals of Deep Learning – Activation Functions and When to Use Them? https://www.analyticsvidhya.com/blog/2020/01/fundamentals-deep-learning-activation-functions-when-to-use-them/ (*consultado en Mayo de 2022*)

[13] Keras API- Layer activation functions - https://keras.io/api/layers/activations/ (*consultado en Mayo de 2022*)

[14] Raul E. Lopez Briega - Redes neuronales convolucionales con TensorFlow https://relopezbriega.github.io/blog/2016/08/02/redes-neuronales-convolucionales-con-tensorflow/ (*consultado en Mayo de 2022*)

[15] Knowledge Center - Global max pooling 2D https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/blocks/global-max-pooling-2d (*consultado en Mayo de 2022*)

[16] T. Nawaz; D. Campo; M. O. Mughal; L. Marcenaro; C. S. Regazzoni (2017) - *Jammer Detection Algorithm for Wide-band Radios using Spectral Correlation and Neural Networks*

[17] Héctor Iván Reyes; Naima Kaabouch (2013) - *Jamming and Lost Link Detection in Wireless Networks with Fuzzy Logic*

[18] Haji M. Furqan; Mehmet A. Aygül; Mahmoud Nazzal; Hüseyin Arslan (2020). *Primary user emulation and jamming attack detection in cognitive radio via sparse coding*.

[19] Silvija Kokalj-Filipovic, Rob Miller, Nicholas Chang y C. L. Lau (2019) - *Mitigation of Adversarial Examples in RF Deep Classifiers Utilizing AutoEncoder Pre-training*.

[20] Silvija Kokalj-Filipovic, Rob Miller (2019) - *Adversarial Examples in RF Deep Learning: Detection of the Attack and its Physical Robustness.*

[21] Ozan Alp Topal, Selen Gecgel, Ender Mete Eksioglu, y Gunes Karabulut Kurt (2019) - *Identification of Smart Jammers: Learning based Approaches Using Wavelet Representation*

[22] Sahar Ujan, Mohammad Hossein Same, Rene Jr Landry (2020) - *A Robust Jamming Signal Classification and Detection Approach Based on Multi-Layer Perceptron Neural Network*

[23] Bikalpa Upadhyaya, Sumei Sun y Biplab Sikdar (2019) - *Machine Learning-based Jamming Detection in Wireless IoT Networks*

[24] Markus Dillinger, Kambiz Madani, Nancy Alonistioti (2003) - *Software Defined Radio: Architectures, Systems and Functions*.

[25] Red Hat (mayo 2020) - *What is a REST API?* https://www.redhat.com/en/topics/api/what-is-a-rest-api (*consultado en Mayo de 2022*)

[26] MDN web docs - *Django introduction* https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Introduction (*consultado en Mayo de 2022*)

[27] Joaquín Amat Rodrigo (noviembre 2016, última actualización noviembre 2020) - *Validación de modelos predictivos: Cross-validation, OneLeaveOut, Bootstraping* https://www.cienciadedatos.net/documentos/30_cross-validation_oneleaveout_bootstrap (*consultado en Mayo de 2022*)

A. ANEXOS

A1. Anexo 1: *TFM_1_data_preparation.ipynb*

Link Github:
https://github.com/AlexCanoMoya/TFM_Master_AI/blob/main/TFM_1_data_preparation.ipynb

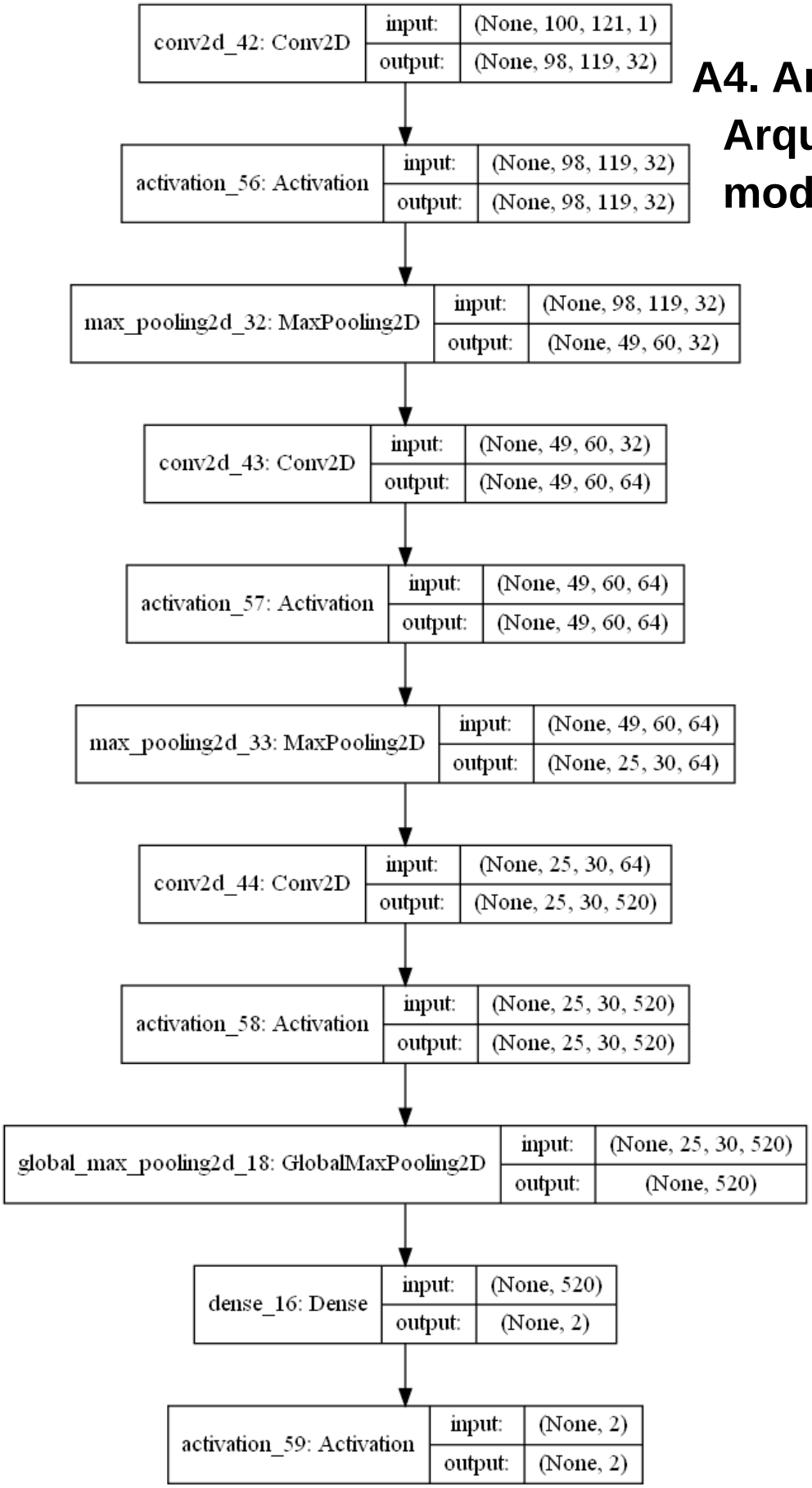
A2. Anexo 2: *TFM_2_data_partition.ipynb*

Link Github:
https://github.com/AlexCanoMoya/TFM_Master_AI/blob/main/TFM_2_data_partition.ipynb

A3. Anexo 3: *TFM_3_model.ipynb*

Link Github:
https://github.com/AlexCanoMoya/TFM_Master_AI/blob/main/TFM_3_model.ipynb

A4. Anexo 4: Arquitecturadel modelo implementado.



A5. Anexo 5: *TFM_4_model_evaluacion.ipynb*

Link Github:
https://github.com/AlexCanoMoya/TFM_Master_AI/blob/main/TFM_4_model_evaluation.ipynb