UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Escuela Técnica Superior de Ingeniería de Telecomunicación

Extracción de características mediante autocodificadores para el procesado de señales acústicas submarinas producidas por embarcaciones

TRABAJO FIN DE GRADO

GRADO EN INGENIERIA DE SISTEMAS DE TELECOMUNICACION

Autor: Nuria Navarro Martínez

Director: José Luis Sancho Gómez

Codirector: Jorge Pérez Aracil

Cartagena, 10 de septiembre de 2022

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

La identificación de embarcaciones haciendo uso de las señales acústicas submarinas ha despertado un gran interés, especialmente en aplicaciones militares [1]. Sin embargo, este interés se ha extendido a diversos campos, como por ejemplo en la gestión de tráfico marítimo, en la pesca, así como en la protección del entorno marítimo [2]. La detección y clasificación de señales acústicas tiene una especial utilidad en el control del tráfico marítimo [3], [4], y además, permite identificar el origen de ruido submarino.

En el contexto del *Machine Learning* (ML), las señales acústicas se enmarcan en los problemas de series temporales. Éstas tienen diversas aplicaciones, entre las que destacan: medicina [5], la meteorología [6] o en economía [7]. En la mayoría de ellas, los problemas a resolver son problemas de clasificación [8].

Uno de los problemas más comunes a la hora de trabajar con señales temporales de cualquier naturaleza es la cantidad de ruido de las mismas, además de otras restricciones como por ejemplo la disponibilidad de los datos o requerimientos de *hardware* para ejecutar las redes. Para la problemática del ruido, es posible utilizar en muchos casos filtros que mejoren la calidad de las señales. Sin embargo, esta técnicas no son suficientes en muchas situaciones. Esto ha motivado a muchos investigadores a hacer uso de los Autocodificadores (AE, por sus siglas en inglés) junto con técnicas de ML [9, 10]. Este tipo de arquitecturas permiten reducir considerablemente el número de entradas del modelo de ML, haciendo éste más simple. Esto implica una gran ventaja en diversos campos, puesto que a medida que reducimos la complejidad de los modelos, podemos facilitar su implementación, ya que los requerimientos de *hardware* son mucho menores.

1.1 Objetivos

En este Trabajo Fin de Grado (TFG) se aborda la tarea de clasificación de señales acústicas de embarcaciones, todas ellas de la base de datos de señales acústicas submarinas recogidas por los investigadores de la Universidad de Vigo llamada *ShipsEar* [11]. Los objetivos de

este TFG pueden resumirse en los siguientes:

1. Evaluar del rendimiento de las técnicas de ML clásicas en la clasificación de señales acústicas submarinas.

2. Evaluar el rendimiento de los AEs como proceso previo al de clasificación que permita reducir la dimensionalidad de los datos de entrada.

1.2 Estructura del trabajo

La solución propuesta en este trabajo se basa en en aplicar un AE para la extracción de características de las series temporales, y algoritmos de *Shallow Learning* para realizar la clasificación. Se estudiará el efecto del AE, y se comparará el rendimiento con los métodos de ML, aplicando tanto los datos en crudo, los filtrados y las características obtenidas por el AE a los mismos algoritmos de clasificación.

1.2.1 Base de datos

La base de datos empleada en este TFG ha sido cedida por la Universidad de Vigo. Esta base de datos, denominada *ShipsEar* [2], cuenta con un total de 90 archivos de audio, que se corresponden a un conjunto de señales acústicas producidas por embarcaciones y ruidos ambientales captados en la zona, como el ruido del fondo marino u otros fenómenos meteorológicos. En total se diferencian once tipos distintos de embarcaciones, que se agruparán en cuatro grupos distintos de cara a realizar la tarea de clasificación.

1.2.2 Redes empleadas

En cuanto a las redes neuronales empleadas, se trabajará tanto con redes superficiales como con una red profunda. Para la primera parte de este trabajo, la extracción de características, se va a trabajar con AEs convolucionales. Por otra parte, para realizar el problema de clasificación, se emplean modelos de ML como *Random Forest*, árbol de decisión, máquinas de vectores de soporte, *AdaBoost* y redes Perceptrón Multicapa.

1.2.3 Obtención de resultados

Además del análisis del correcto funcionamiento del AE en su función de reconstrucción de señales, se han estudiado la viabilidad y eficiencia de todas las redes de clasificadores mencionadas en un total de cuatro casos de estudio, en los que se varía la entrada de dichas redes y se obtienen los resultados de las predicciones. Estos casos son los siguientes: pasando como entradas las series temporales, introduciendo las series temporales filtradas con un filtro paso bajo, introduciendo las señales filtradas por el codificador del AE para tener como entradas las características extraídas, y pasando las series sin filtrar introducidas previamente por el codificador.

Capítulo 2

Aprendizaje automático aplicado a la clasificación de señales

En este capítulo se recogerá teoría básica de señales y antecedentes teóricos del *Machine Learning*, y se detallarán los tipos de redes neuronales que se van a implementar en este proyecto. De esta forma, se pretende proporcionar una visión general del problema y de las técnicas implementadas para llevar a cabo la solución propuesta.

2.1 Señales y sistemas

Señales. Una señal es una magnitud física (ya sea voltaje, intensidad eléctrica, temperatura, ...) cuya evolución se puede representar matemáticamente gracias a su variación con respecto a alguna variable independiente, como el tiempo o el espacio.

Además, toda señal puede ser clasificada atendiendo a una colección de distintos criterios. Algunos de ellos son los siguientes:

- Señales unidimensionales o multidimensionales, según si varían en función de una o más variables independientes.
- Señales continuas o discretas. Una señal continua es aquella cuya variable independiente puede tomar cualquier valor real. Por otra parte, si la variable independiente toma solo valores de números enteros, se trata de una señal en tiempo discreto. Estas señales también se denominan secuencias.
- Señales analógicas y digitales. Cuando la señal puede tomar valores entre un conjunto infinito de valores, se trata de una señal analógica. Si por el contrario la señal solo puede tomar valores de un conjunto finito, será una señal digital.

Señal acústica. Una señal acústica es la representación eléctrica de una onda sonora. El sonido no es más que la propagación de ondas mecánicas sobre un medio elástico. Es decir, es la propagación de vibraciones o fluctuaciones de presión en un medio. En este trabajo, las vibraciones las generan las embarcaciones que entran y salen del puerto, mientras que

el medio de propagación es el agua.

Cuando grabamos el sonido, estamos recopilando información en forma de tensión eléctrica sobre la variación temporal de las fluctuaciones en el agua. Esta señal se corresponde con una señal analógica de audio.

Ahora bien, para trabajar con estas señales, es necesario tratarlas como señales discretas y digitales, lo que conocemos como audio digital: una secuencia de valores finitos. Para ello, se procesa la señal aplicando un muestreo y una cuantificación digital. El muestreo toma valores de la señal analógica de forma periódica (cumpliendo el teorema del muestreo) mientras que la cuantificación digital cuantifica con bits dichos valores.

Sistemas. Un sistema es un modelo que opera sobre una señal de entrada y genera una señal de salida, cuyo sentido se basa en valer como representación matemática de fenómenos físicos. El sistema queda definido por la transformación T que se aplica a la entrada, siendo la entrada $x[n]$ y la salida $y[n]$:

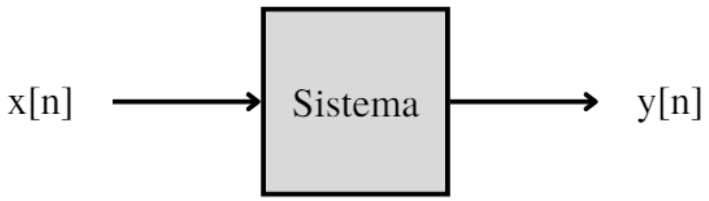


Figura 2.1: Esquema básico de un sistema discreto.

$$y[n] = T\{x[n]\} \tag{2.1}$$

Filtros. Se suele denominar filtro a aquellos sistemas que se diseñan para un propósito determinado. Si hablamos de filtros electrónicos, hablamos de aquellos sistemas que discriminan o eliminan un rango de frecuencias de una señal entrada.

Los filtros electrónicos a su vez se pueden clasificar en:

- Activos o pasivos. Los filtros pasivos no necesitan una fuente de energía externa para funcionar y no tienen componentes activos (como transistores) en su modelo. Los filtros activos, en cambio, pueden estar compuestos por elementos pasivos o activos.
- Analógicos o digitales, según el tipo de señales que puedan tratar, analógicas o digitales.

- Según el tipo de respuesta en frecuencia que proporcionen, podemos estar ante filtros paso bajo, paso alto, filtros paso banda y filtros elimina banda.

En este trabajo vamos a filtrar las secuencias con un filtro paso bajo digital (por sus siglas, FPB). Los filtros paso bajo dejan pasar el conjunto de frecuencias más bajas y eliminan las altas. Al diseñar un filtro paso bajo, se determina la frecuencia de corte, que se trata del valor de frecuencia hasta la cual se va a permitir el paso de información a la señal de salida.

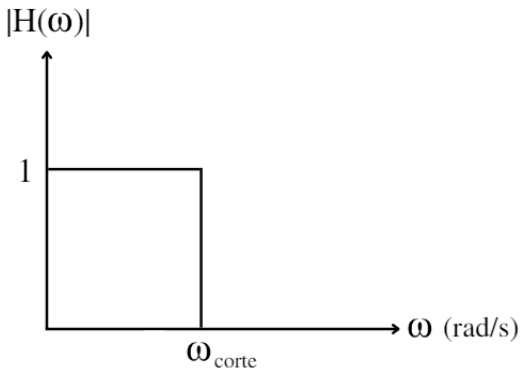


Figura 2.2: Respuesta en frecuencia de un filtro paso bajo ideal.

2.2 Aprendizaje automático

El aprendizaje automático o *Machine Learning* (ML) es un subcampo de la Inteligencia Artificial que se centra en el desarrollo de sistemas que aprendan de forma autónoma de los datos que se le proporcionan con la mínima intervención humana.

Aunque se suelen mencionar juntos, la Inteligencia Artificial, o AI por sus sigas en inglés, es todo aquel software que imita la inteligencia humana y que se desarrolla en un amplio conjunto de áreas. Así pues, no todos los problemas de AI son ML.

Además, aunque este trabajo se centra en técnicas de ML, también nos valemos de algoritmos de aprendizaje profundo o *Deep Learning* (DL). El DL es un subcampo del ML que utiliza redes neuronales profundas, de las que se hablará posteriormente, para analizar el comportamiento de los datos a un nivel de abstracción mayor.

Debido a que el ML es un campo reciente, existe discrepancia respecto a su significado exacto y cada autor difiere en leves matices a la hora de definirlo. Russell y Norvig, dos

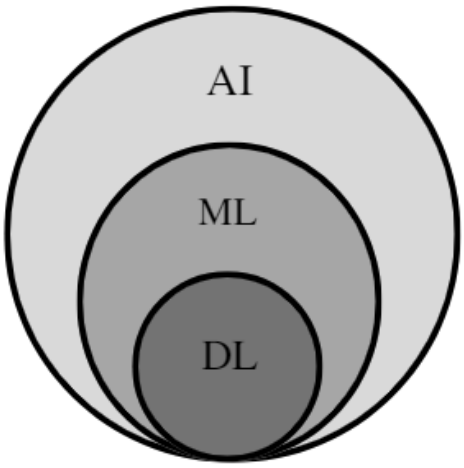


Figura 2.3: Esquema relación AI, ML, DL.

expertos en ciencias de la computación, escriben lo siguiente sobre el ML: "En el aprendizaje máquina un computador observa datos, construye un modelo basado en esos datos y utiliza ese modelo a la vez como una hipótesis acerca del mundo y una pieza de software que puede resolver problemas"[12].

Un modelo genérico de aprendizaje automático consta de las siguientes partes, tal y como se muestra en la Figura 2.4: recopilación y preparación del conjunto de datos, selección de características, elección del algoritmo, selección del modelo y los parámetros, entrenamiento y evaluación del rendimiento.

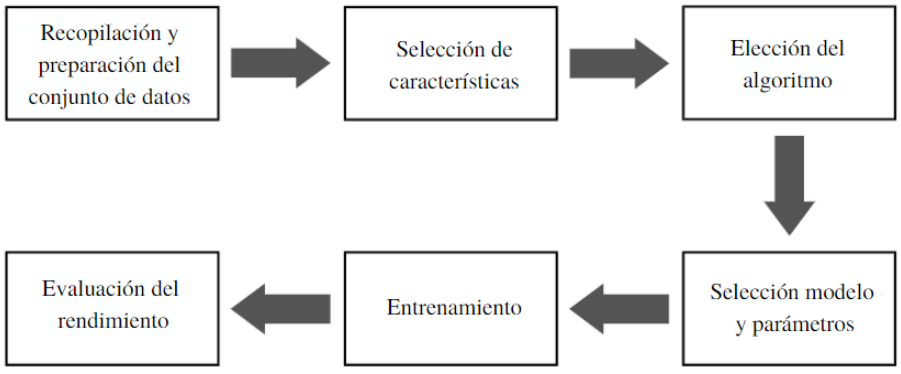


Figura 2.4: Componentes de un modelo de ML genérico. Adaptado de [13].

Para una tarea pueden extraerse grandes volúmenes de información, que pueden pre-

sentarse como datos no estructurados o tener inconsistencias. En estos casos, es necesario hacer una recopilación, limpieza y análisis de todos los datos antes de poder servir como entrada de cualquier algoritmo. Una vez procesado el conjunto de datos, se debe elegir aquellas características de entre todo el conjunto que sean relevantes para el problema.

Cuando se tiene una buena base de datos, se elige el algoritmo de aprendizaje automático que mejor encaje con el problema planteado, además de determinar los parámetros y estructura del modelo según el algoritmo.

Tras ello, se pasa a entrenar el modelo con un subconjunto de datos. En el entrenamiento, se ejecuta el modelo una serie de veces para aprender el comportamiento de los datos y ajustar los parámetros según dicho comportamiento.

Finalmente, se prueba el modelo con otro conjunto de datos y se miden distintos parámetros de rendimiento que verifiquen el buen funcionamiento, entendiéndose éste como el número de aciertos en función del total procesado.

En función del objetivo del problema de estudio se distinguen distintos tipos de aprendizaje máquina: el aprendizaje supervisado y el no supervisado. Otra clase de aprendizaje es el semi-supervisado, que no va a ser de interés en este trabajo.

En el aprendizaje no supervisado se caracteriza por ser modelos donde las etiquetas son desconocidas en todo momento. El problema más importante es el de agrupamiento o *clustering*, que agrupa conjuntos de datos en base a relaciones estadísticas [14] y obteniendo como salida los distintos subgrupos de datos, en los que los datos integrantes compartan características que los diferencian del resto de agrupaciones.

El aprendizaje supervisado emplea datos etiquetados como entrada en el entrenamiento del modelo. Conocer de antemano la respuesta correcta a las entradas permite calcular la precisión e ir ajustando los pesos en base a este cálculo. En función del objetivo de salida diferenciamos entre problemas de regresión, que generará una salida de valores continuos y que permiten establecer una mejor relación causa-efecto, y problemas de clasificación, cuya salida serán números enteros o variables categóricas.

En este trabajo, se va a desarrollar un problema de clasificación, por lo tanto se trata de un problema de aprendizaje supervisado con salida entera o categórica. Existen diversos tipos de redes que permiten abordar esta clase de problemas dentro del ámbito del ML, y que se clasifican en *Shallow Networks* y *Deep Networks*. Dentro de las *Shallow Networks* vamos a ver ejemplos de redes tales como el árbol de decisión, *Random Forest*, máquinas de vectores de soporte, AdaBoost y el Perceptrón Multicapa, mientras que dentro de las *Deep Networks* emplearemos una red neuronal convolucional para implementar un autocodificador.

2.2.1 *Shallow Networks*

El término de redes neuronales superficiales o *Shallow Networks* se emplea para describir aquellas redes neuronales que generalmente tienen una sola capa oculta. El concepto de capa oculta se detalla en los siguientes apartados de esta memoria. Esta estructura simple permite a las redes superficiales aprender características generales, lo que se adapta bien a tareas de aprendizaje con datos de baja dimensión.

A continuación se describen algunas redes neuronales superficiales:

- Perceptrón multicapa

  El primer modelo de una red neuronal fue definido por Warren McCulloch y Walter Pitts en su artículo "Cálculo lógico de ideas inherentes en la actividad nerviosa" (1943). A partir de su trabajo, el científico Frank Rosenblatt desarrolló y construyó el primer perceptrón, dando pie al posterior desarrollo de las redes neuronales artificiales.

  Las redes neuronales se inspiran en la funcionalidad de las redes neuronales del cerebro humano, de forma que al igual que éstas tienen neuronas para transmitir información en forma de impulsos eléctricos, las redes neuronales artificiales cuentan con nodos interconectados que hacen el papel de neuronas.

  Así pues, el perceptrón es un modelo matemático que representa el funcionamiento de una neurona y, por tanto, la mecánica básica de una red neuronal. Este funcionamiento se centra en relacionar un vector de variables independientes, las entradas $\vec{x}$ del perceptrón, con otra variable independiente y, la salida o predicción, como se muestra en la Figura 2.5.

  El vector de pesos $\vec{w}$ pondera los valores de las entradas y los resultados se suman en un único valor z. Dicho de otra forma, se obtiene la combinación lineal de los valores de las entradas y sus pesos:

$$z = b_0 + \sum_{i=1}^{N} w_i x_i \tag{2.2}$$

  donde el peso b_0, conocido como *bias*, es el término independiente, que puede entenderse como aquel que multiplica a una entrada de valor siempre igual a 1; w_i, x_i, representan los pesos y entradas, respectivamente, para cada una de las neuronas i.

  Este resultado z se evalúa con la función de activación σ, que determina el nivel de activación de la neurona, cuya expresión es:

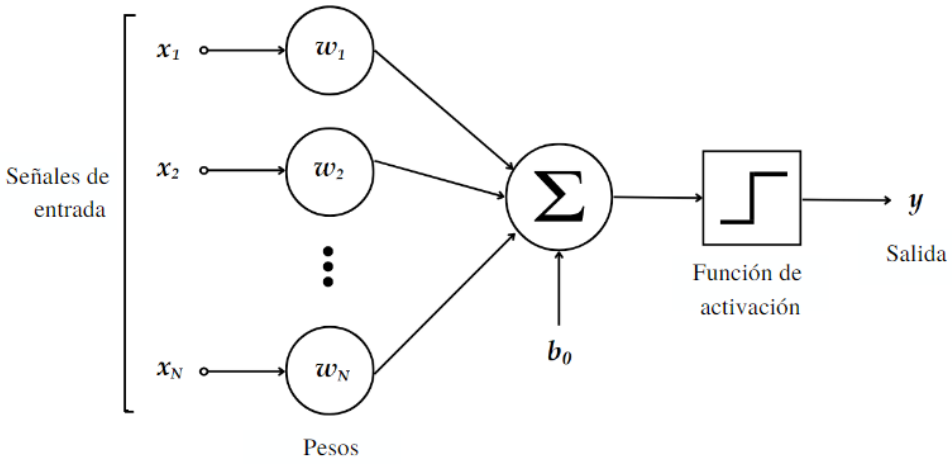


Figura 2.5: Esquema perceptrón simple.

$$y = \sigma\left(b_0 + \sum_{i=1}^{N} w_i x_i\right). \tag{2.3}$$

Al entrenar el perceptrón, se van ajustando todos los pesos para mejorar la predicción, lo que equivale a reducir el valor de la función de pérdidas. La función de pérdidas cuantifica el error que se produce entre la salida obtenida y el valor de entrada que se quería predecir.

Para reducir la función de pérdidas, las redes neuronales emplean un algoritmo iterativo conocido como descenso del gradiente. Se trata de un algoritmo de optimización que intenta encontrar el valor de los pesos que hace converger la función de pérdidas hacia su valor mínimo. El procedimiento se basa en calcular para cada iteración k el gradiente de la función de pérdidas, y actualizar los pesos según la siguiente expresión:

$$\vec{\omega}^{(k+1)} = \vec{\omega}^{(k)} - \gamma \vec{\nabla} J\left(\vec{\omega}^{(k)}\right) \tag{2.4}$$

donde γ es el ratio de aprendizaje, que indica cuán grande serán las actualizaciones de los pesos en cada iteración, $J(\vec{\omega}^{(k)})$ es la función de pérdidas evaluada con los valores de los pesos de dicha iteración, $-\vec{\nabla} J(\vec{\omega}^{(k)})$ es el opuesto del gradiente de la función de pérdidas y $\vec{\omega}^{(k+1)}$ es el nuevo valor que adquiere el vector de pesos tras la iteración k.

Este proceso se repite hasta cumplir el criterio de parada indicado, que bien puede ser un número fijo de iteraciones, alcanzar un valor mínimo del error, o traspasar un

determinado tiempo de ejecución.

Otro algoritmo de optimización es el Adam. Ha sido diseñado para entrenar redes neuronales profundas y es uno de los algoritmos más utilizados debido a su rápida convergencia y buen funcionamiento en la mayoría de los problemas [15].

Es importante establecer una función que aproxime correctamente el comportamiento general de los datos de entrada, ya que es posible que durante el entrenamiento se produzca sobrejuste o subajuste.

El sobreajuste se produce al intentar ajustar el modelo a características particulares, entre las que se puede incluir el ruido, provocando que el modelo no sea generalizable y el rendimiento de la red caiga al ejecutar el modelo final.

Por otra parte, el subajuste se produce cuando el modelo no es lo suficientemente complejo para ajustarse al comportamiento de los datos, de forma que no se puede obtener una buena aproximación.

La Figura 2.6 muestra los distintos estos casos al intentar aproximar la función coseno, usando regresión lineal con características polinómicas. Los polinomios de grado 1 y grado 15 corresponden a un caso de subajuste y sobreajuste respectivamente, mientras que el de grado 4 obtiene una buena aproximación.

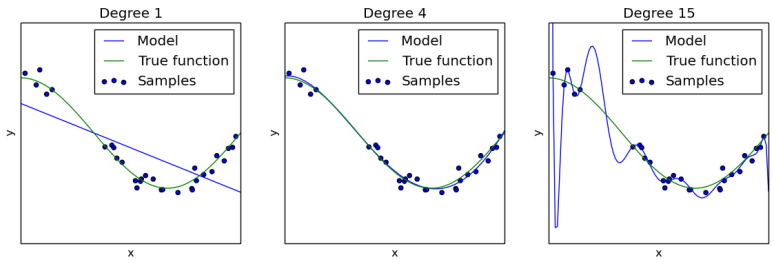


Figura 2.6: Grado 1: subajuste. Grado 4: buena aproximación. Grado 15: sobreajuste. Imagen extraída de [16].

El perceptrón simple o de una capa sirve para resolver tareas de clasificación con dos clases de datos. Para resolver problemas más complejos donde los datos no se puedan separar simplemente por una línea recta, se aumentaría el número de perceptrones creando una red. Sin embargo, todo el conjunto se podría entender como un único perceptrón combinación lineal del resto, por lo que para dotar de la no linealidad

necesaria para dividir más de dos clases, se utilizan funciones de activación no lineales.

Por otra parte, el perceptrón multicapa o *MultiLayer Perceptron* (MLP) es una red neuronal capaz de resolver problemas que no son linealmente separables mediante el uso de funciones de activación no lineales. Su arquitectura consta siempre de tres partes: una capa de entrada, una o más capas ocultas y una capa de salida.

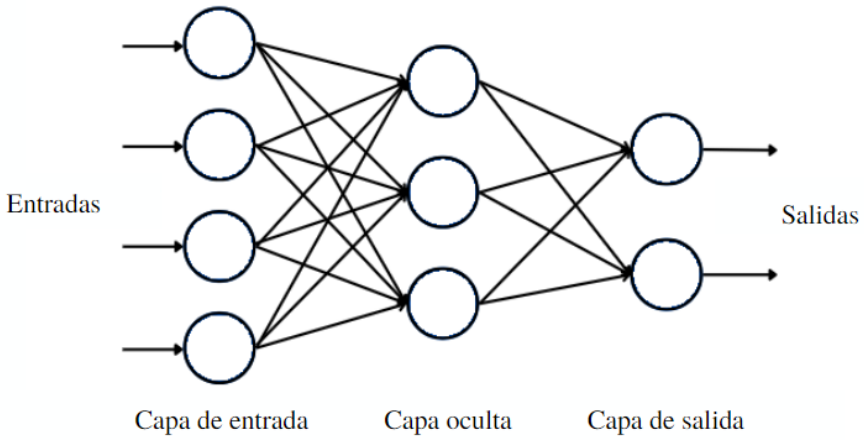


Figura 2.7: Arquitectura de un Perceptrón Multicapa.

La capa de entrada lee el conjunto de datos que se proporciona al modelo. Este conjunto puede ser datos en crudo o bien características obtenidas por distintos métodos. Los nodos de la capa de entrada se interconectan con los nodos de la siguiente capa, una capa oculta, para transmitir la información al siguiente nivel.

Todas las capas intermedias entre la capa de entrada y la de salida se denominan capas ocultas. Los nodos de estas capas realizan los cálculos para encontrar características y patrones ocultos en los datos de entrada.

Finalmente, la capa de salida es la predicción del modelo tras procesar la entrada, será la respuesta de la red al problema planteado.

Que este tipo de modelos pueda tener más de una capa oculta aporta complejidad al problema, ya que usualmente cada neurona estará conectada a todas las neuronas de la capa siguiente. Por ello, en problemas de aprendizaje supervisado, estas redes se entrenan con algoritmos de propagación hacia atrás o retropropagación (del inglés *backpropagation*). Este algoritmo se emplea para actualizar o auto-adaptar los pesos

de la red, y su funcionamiento consta de dos fases:

1. En la primera fase o etapa hacia adelante, se introduce la entrada al modelo y se van ejecutando todas las capas intermedias hasta llegar a la capa de salida. La respuesta obtenida se compara con la salida deseada y se calcula el error producido.
2. En la segunda fase o etapa hacia atrás, el error calculado se propaga de la capa de salida a todas las capas ocultas, sin embargo, cada neurona solo recibe una fracción del error total en base a su contribución a la respuesta obtenida. De esta forma, cada peso se actualiza en distinta medida según si su neurona ha contribuido más o menos al error generado.

Respecto a las funciones de activación no lineales que se emplean en el perceptrón multicapa, algunas de las más empleadas son las siguientes:

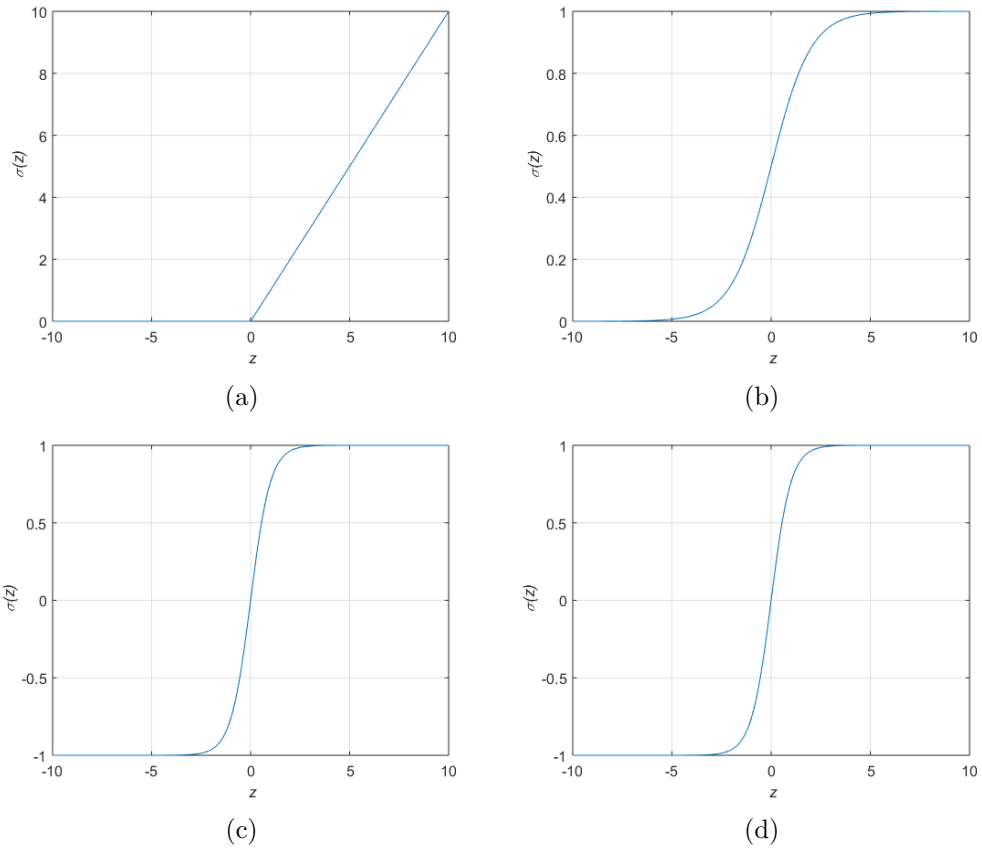


Figura 2.8: Funciones de activación: (a) ReLU, (b) Sigmoide, (c) Softmax, (d) Tangente hiperbólica.

– Función rectificadora, ReLU por sus siglas en inglés *(Rectified Linear Unit)*:

$$\sigma(z) = max(0, z) \tag{2.5}$$

Esta función devolverá 0 como resultado para valores negativos, y el propio valor para valores positivos. Es decir, la respuesta de esta función es dejar inactivas las neuronas con valores negativos y activar los de valores positivos.

La función rectificadora se emplea con frecuencia en redes neuronales debido a su capacidad de rápido aprendizaje.

– Función sigmoide:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.6}$$

Esta función limita su respuesta a valores entre 0 y 1. Para valores más negativos se aproxima con una curva suave a 0, y para valores más positivos, a 1.

La función sigmoide se suele usar en las capas finales de problemas de clasificación cuando contamos con dos clases distintas de datos. Para problemas con más de dos clases, esta función puede no ser eficaz debido a que para cada clase realiza cálculos de probabilidad independientes entre sí, es decir, la probabilidad de pertenecer a una clase no está condicionada a las probabilidades de pertenecer a las restantes, lo que puede provocar que se clasifique una entrada como perteneciente a dos clases distintas.

– Softmax:

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \tag{2.7}$$

Esta función se utiliza para problemas de regresión logística multiclase ya que aporta el cálculo de la probabilidad relativa de cada clase, y por tanto el total de la suma de todas las probabilidades es igual a 1.

– Tangente hiperbólica:

$$\sigma(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} \tag{2.8}$$

Esta función tiene el mismo comportamiento que la función de activación sigmoide, cambiando el rango de valores de salida por 1 para valores positivos y -1 para negativos.

Autocodificadores. Un autocodificador o *AutoEncoder* (AE) es un tipo de red que se emplea en tareas de aprendizaje no supervisado en su versión original [17].

Su objetivo se basa en la obtención de una representación de menor dimensión de los datos de entrada, de forma que sea capaz de generar, a partir de dicha representación, una salida lo más parecida posible a la entrada.

La arquitectura típica de un autocodificador consta de tres partes:

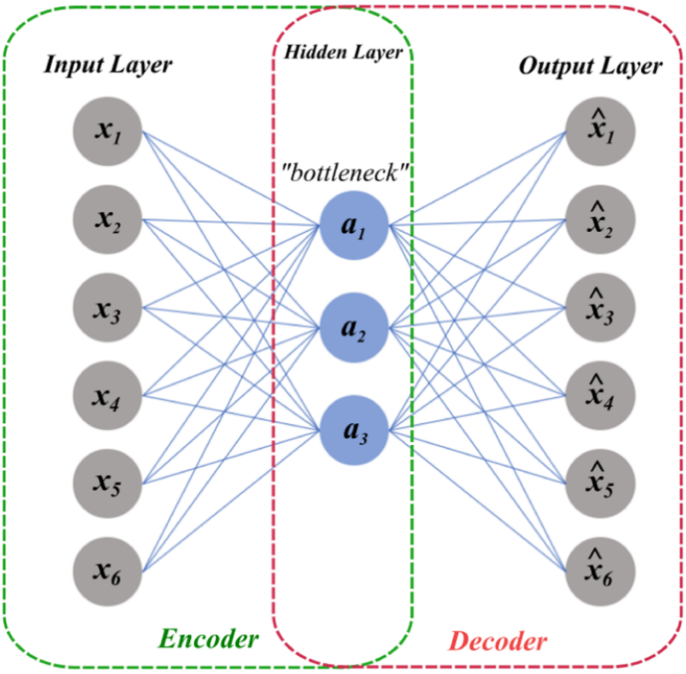


Figura 2.9: Arquitectura de un Autocodificador. Imagen extraída de [18].

- Codificador. Este módulo aprende a interpretar la entrada y la comprime en una representación interna definida por el espacio latente.
- Espacio latente (*bottleneck* o cuello de botella). Representa el espacio de nivel más bajo en el que se reducirá la entrada. Es la parte más importante de la red neuronal ya que restringe el flujo de información entre codificador y decodificador.
- Decodificador. Este módulo tiene el efecto contrario al codificador,"descomprime" la representación del espacio latente y genera una salida similar a la entrada.

Como se aprecia en la Figura 2.9, se trata de una red simétrica, donde el tamaño de entrada es el mismo que el de salida, siendo el número de nodos menor en las capas intermedias. La menor dimensionalidad de las capas ocultas asegura una extracción eficaz de las características más representativas, de forma que se establezcan correlaciones útiles entre los datos de entrada. De esta forma, al decodificador solo llegan las características críticas para hacer una reconstrucción válida [19].

La dimensión del espacio latente es el parámetro más importante en el diseño de un AE. Si la dimensión es demasiado grande, puede producirse sobreajuste. Si por el contrario es demasiado pequeña, puede perderse parte de información relevante por producirse subajuste.

Otros parámetros que se definen antes de entrenar un AE son el número de capas, que indica la profundidad del modelo, y el número de nodos por capas. Si un AE tiene más capas ocultas que entradas, es posible que el algoritmo aprenda la función identidad en el entrenamiento, provocando que la salida sea exactamente igual a la entrada. Aunque se busca poder reconstruir la señal de entrada de la forma más exacta posible, esto es un inconveniente porque significa que el AE no está aprendiendo de las entradas, sino que las "copia" a la salida.

Como función de pérdidas, para tener una medida de acierto que ayude al AE a ajustar sus pesos para mejorar durante el entrenamiento, se obtiene el error de reconstrucción, que será la diferencia entre la entrada y la entrada reconstruida (el *output*).

Cuando el AE ha sido bien entrenado, el subconjunto que se encarga de la extracción de características (codificador y espacio latente) puede emplearse en etapas previas de procesado de datos, siendo la salida del AE la entrada de la tarea específica, tal como puede ser un problema de clasificación.

Generalmente, los AE se utilizan en tareas de reducción de dimensionalidad, es decir, la reducción del número de variables aleatorias, pero tienen otras muchas aplicaciones, algunas de las cuales son las siguientes:

1. Eliminación del ruido.

   Para un autocodificador, es posible "filtrar" o ignorar componentes ruidosas de forma inherente debido a la reducción de dimensionalidad de las entradas.

   Existe un tipo de AE, el *Denoising Autoencoder*, o autocodificador de eliminación de ruido, cuyo funcionamiento se basa en la corrupción de los datos de entrada para intentar reconstruirlos. Es decir, se crearán copias ruidosas de los datos, el AE deberá eliminar este ruido y así se obtendrá a la salida los datos originales.

   De esta forma, el AE genera mejores representaciones en el espacio latente de los datos, lo que asegura una buena reconstrucción de los datos ruidosos. Además, los AE de eliminación de ruido tienen menos riesgo de aprender la función identidad, ya que sus entradas nunca serán los datos originales, sino su versión ruidosa.

2. Detección de anomalías.

   Es posible emplear un autocodificador como detector de anomalías. En este caso, el AE se entrena solo con datos que no presenten anomalías, de forma que se obtiene una representación latente de los datos normales, y se generan salidas regulares. Cuando se introduzcan datos con anomalías como entradas, el AE no podrá reconstruirlos adecuadamente y el error de reconstrucción se incrementará, marcando entonces la presencia valores anómalos.

3. Modelos generativos.

Gracias al *Variational autoencoder* (VAE) o autocodificador variacional, es posible hablar de generación de datos sintéticos mediante AE, especialmente en aprendizaje supervisado y semi-supervisado.

Los autocodificadores variacionales comparten arquitectura con los AE pero en lugar de obtener correlaciones entre los datos de entrada, se obtiene la distribución de probabilidad que los modela. El decodificador se alimenta con muestras de puntos de dicha distribución para generar datos nuevos.

Máquina de vectores de soporte. En un algoritmo de vectores de soporte, en inglés *Support Vector Machine* (SVM), los datos se representan en puntos de un espacio n-dimensional (siendo n el número de características del conjunto de datos) para segregar las distintas clases mediante un hiperplano. Este hiperplano tiene la propiedad de máximo margen, es decir, maximiza su distancia con las muestras más cercanas de cada clase y definido por los vectores soporte, que son los puntos extremos de cada clase. La Figura 2.10 muestra un ejemplo de vectores de soporte para el caso de división lineal (2 clases de datos).

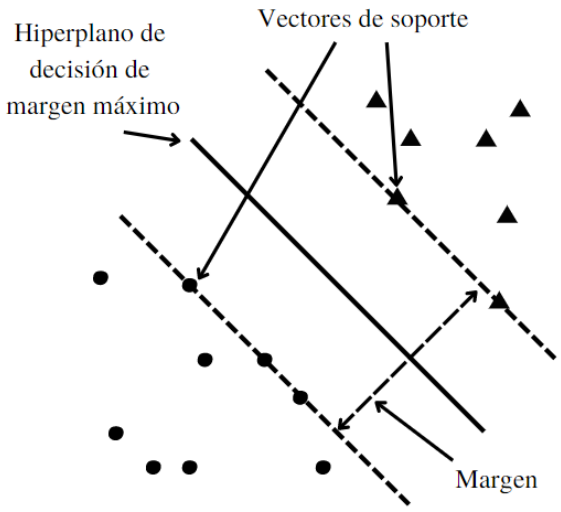


Figura 2.10: Esquema *Support Vector Machine*. Adaptado de [20].

Árbol de decisión. Este algoritmo se usa especialmente en problemas de clasificación. Su nombre viene de la forma de árbol que toma su estructura, que está formada por un nodo raíz, varios nodos de decisión, sus ramas y nodos hoja, como se ilustra en la Figura 2.11.

En el nodo raíz se concentran la totalidad de características, que se van dividiendo según se adapten a las reglas de decisión indicadas en los nodos intermedios o de decisión. Las conexiones entre nodos se conocen como ramas. Cada nodo intermedio tiene dos ramas

que pueden conducir a otro nodo intermedio o a un nodo hoja, y recorrerá una u otra rama según la entrada se adecúe o no a las características indicadas en el nodo decisión. Los nodos hoja corresponden al último nivel de desglose, y representa el resultado de la decisión.

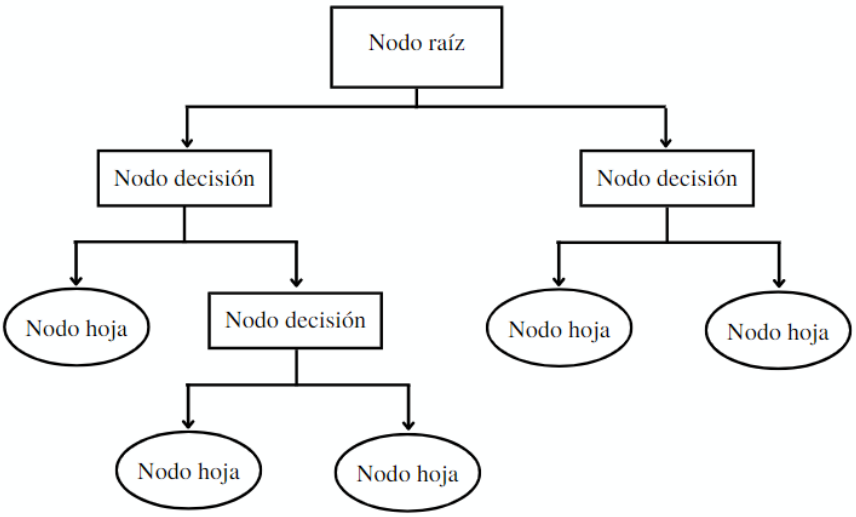


Figura 2.11: Esquema clasificador árbol de decisión.

Una vez formado el árbol se puede proceder a la clasificación de un conjunto de datos dado. Para ello, y empezando por el nodo raíz, se van comparando los registros de los nodos con las entradas, y se va eligiendo el mejor atributo aplicando la Media de Selección de Atributos o *Attribute Selection Measures* (ASM por sus siglas en inglés) hasta que no se puede desglosar más y se llega a un nodo hoja.

Random Forest. Este algoritmo es un método de aprendizaje de conjuntos (*Ensemble Learning*). Este tipo de aprendizaje surge para optimizar algoritmos simples predictivos en situaciones donde el umbral de decisión sea ambiguo, combinando varios de estos algoritmos simples y agregando todas las salidas a un bloque predictor final.

En concreto, un *Random Forest* combina múltiples árboles de decisión de forma paralela, dividiendo aleatoriamente el conjunto inicial de datos T en tantos subconjuntos T_k como árboles tenga, tal y como se esquematiza en la Figura 2.12. La salida final se corresponde con el promedio de todas las decisiones finales de los subárboles.

De este modo, se reduce el tiempo de entrenamiento y la salida tiene gran precisión en comparación con otros algoritmos.

AdaBoost. El algoritmo *AdaBoost* o *Adaptative Boosting* forma parte de los métodos del aprendizaje de conjuntos (*Ensemble Learning*). El objetivo de este algoritmo es implementar un predictor fuerte a partir de los errores de los algoritmos débiles (por defecto

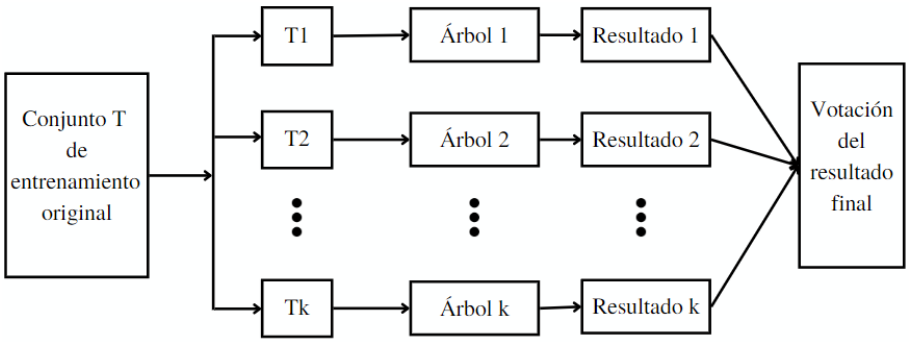


Figura 2.12: Esquema *Random Forest*. Adaptado de [21].

.

este algortimo suele ser un árbol de decisión). El funcionamiento del algoritmo es crear modelos que ajusten sus pesos de forma iterativa y secuencial a partir del modelo anterior, es decir, el modelo N ajustará sus pesos en base a los errores del modelo N-1. La Figura 2.13 ilustra este comportamiento.

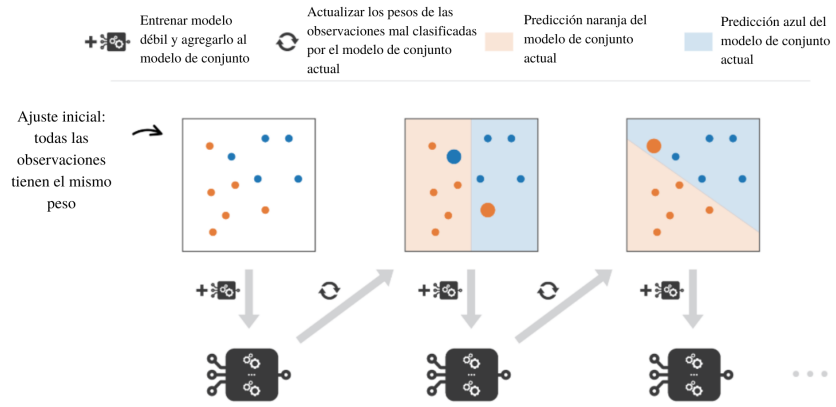


Figura 2.13: Esquema *AdaBoost*. Adaptado de [22].

Dada la naturaleza de las entradas del ejercicio propuesto, y del potencial mostrado recientemente por los métodos de aprendizaje profundo [23] y [24], además de los métodos mencionados de *Shallow Learning* vamos a utilizar técnicas de aprendizaje profundo en el desarrollo de este trabajo.

2.2.2 *Deep Networks*

Las redes profundas o *Deep Networks* son un subconjunto de redes del *Machine Learning* que tienen cierto nivel de profundidad, dotado por la existencia de múltiples capas en su arquitectura, tal y como se muestra en la Figura 2.14, donde *Input Layer* es la capa de entrada, *Output Layer* la capa de salida, y *Multiple hidden layers* las múltiples capas ocultas de la red. Este tipo de redes, denominadas *Deep Neural Networs* o DNN, tienen un rendimiento superior pero, en contraparte, tienen un gran costo computacional debido a la complejidad de las redes.

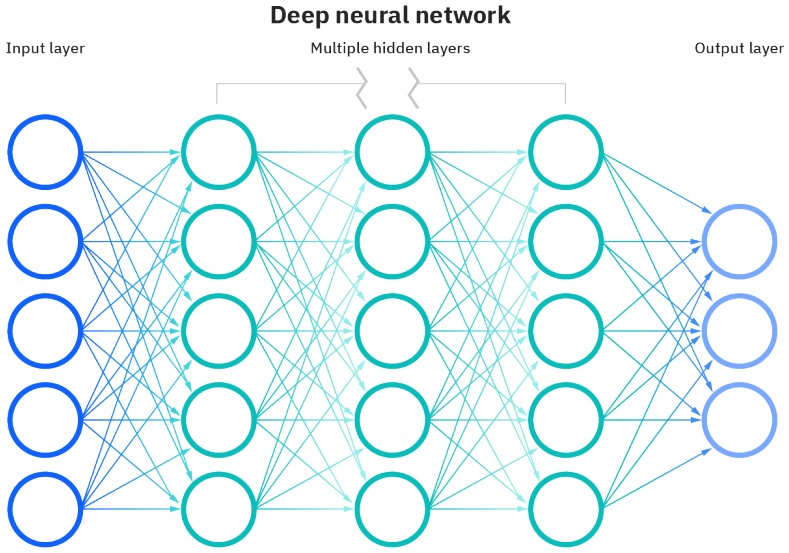


Figura 2.14: Arquitectura DNN. Imagen extraída de [25].

Las redes profundas han cobrado importancia porque pueden ser entrenadas con el descenso de gradiente cambiando la función de activación Sigmoide por ReLU. Este cambio es significativo debido al problema de saturación del gradiente que ocurre con funciones de activación Sigmoides. Este problema se produce cuando las neuronas generan valores (gradientes) que están muy cerca de los límites de un rango de valores compacto. Si el gradiente es constantemente 0, no se produce aprendizaje en la red neuronal, mientras que si el gradiente es constantemente 1, puede provocar sobreajuste.

Otro tipo de redes profundas son las redes neuronales convolucionales o *Convolutional Neural Network* (CNN). Las CNN son un tipo redes neuronales que suelen emplearse en tareas de clasificación de imágenes o detección de objetos, ya que su rendimiento es superior frente a otras redes. Además, pueden emplearse en modelos de AE, tal y como se ha

realizado en este trabajo.

Las arquitecturas de las CNN se componen por la combinación de tres tipos de capas:

1. Capa convolucional. En esta capa o capas, ya que pueden existir capas convolucionales seguidas, se realizan convoluciones sobre los datos de entrada con un número de filtros. Estos filtros son un conjunto de matrices denominadas kernel, y tienen asignados unos pesos que se van actualizando en el entrenamiento mediante el descenso de gradiente y el *backpropagation*. Estos filtros funcionan como un detector de características. Al ir recorriendo las entradas en la convolución, se va creando una matriz de salida, conocida como mapa de características. El procedimiento se detalla en la Figura 2.15, ejemplo de una convolución 2D con una entrada $7x7x1$ y un filtro de un kernel de $3x3$.

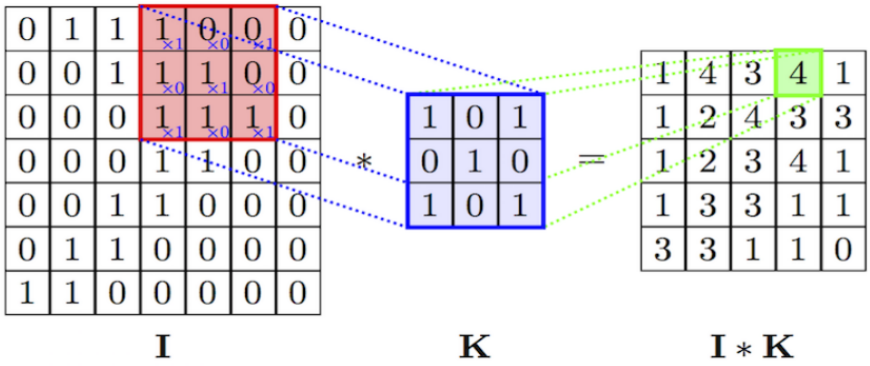


Figura 2.15: Ejemplo de una convolución en 2D. Imagen extraída de [26].

El mapa de características variará en su tamaño según el número de filtros aplicados, ya que por cada filtro se produce un mapa de características, según el *Stride* indicado, que es la distancia que se va moviendo el kernel sobre la entrada, o según el *zero-padding*, que es una técnica que ajusta el tamaño de salida según las especificaciones que se requieran.

2. Capa de agrupación. Esta capa es la encargada de reducir la dimensionalidad manteniendo las características más importantes. Al igual que en la capa convolucional, la entrada se barre con un filtro, sin embargo, este filtro no tiene pesos. En este caso, el kernel aplica una función de agregación para generar la salida. Las funciones de agregación posibles son el *Max pooling*, donde la matriz de salida se va construyendo con el valor máximo de que detecta el filtro mientras recorre la entrada, y el *Average pooling*, que calcula el valor promedio del área que va recorriendo para constituir la salida.

3. Capa *fully-conected*, que será la capa final. El nombre de completamente conectada viene de que, a diferencia de las capas anteriores, cada nodo de esta capa se conecta

a un nodo de la capa anterior. Es aquí donde se realiza la tarea de clasificación según las características extraídas en las capas anteriores.

Un ejemplo de arquitectura de una red neuronal convolucional, con la unión de los tres tipos de capas explicadas, es el ilustrado en la Figura 2.16.

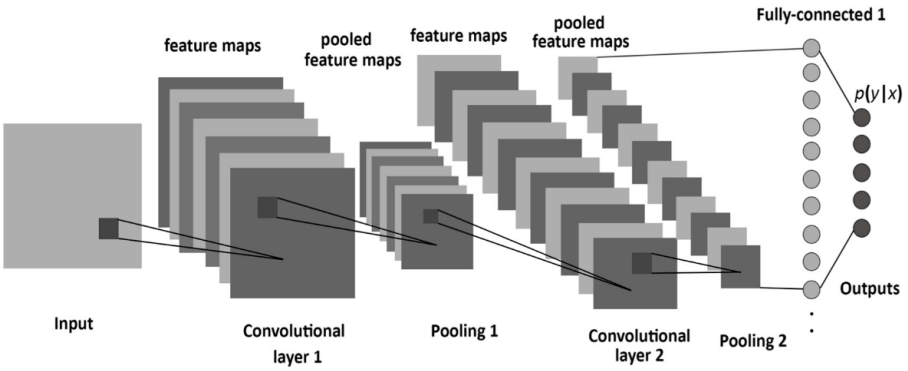


Figura 2.16: Ejemplo de arquitectura de una red neuronal convolucional. Imagen extraída de [27].

Capítulo 3

Metodología

Como se ha introducido previamente, en este trabajo nos hemos centrado en el desarrollo de un problema de *Machine Learning* de clasificación de secuencias temporales, procesándolas previamente con un autocodificador. Las secuencias se obtienen a partir de ficheros de audio de señales acústicas submarinas, que recogen los sonidos emitidos por una gran diversidad de embarcaciones, y que se pueden clasificar en cuatro clases distintas atendiendo a la envergadura de las naves.

En primer lugar, se ha creado un modelo funcional de un autocodificador para la extracción de características de las secuencias obtenidas de los audios. Estas características se emplearán como entradas de distintos modelos de ML, entre los que se encuentran un conjunto de algoritmos que pertenecen a los conocidos *Shallow Methods* y una red MLP. En la Figura 3.1 se visualiza el esquema planteado. Finalmente, se ha estudiado el comportamiento y desempeño de todos los modelos mencionados, así como del efecto de aplicar un filtro paso bajo a las series temporales antes de servir como entradas para el autocodificador.

El desarrollo de este proyecto se ha realizado en la herramienta Google Colaboratory, que permite escribir y ejecutar código *Python* directamente desde el navegador, mediante máquinas virtuales asociadas a la cuenta de Google. Además, se ha optado por realizar la suscripción al servicio de Colab Pro, que da acceso a GPUs más rápidas y mayor capacidad de memoria RAM. Esto ha sido necesario para entrenar y ejecutar correctamente las redes implementadas, además de reducir el tiempo necesario para ello.

También se ha utilizado la herramienta de Google Drive para almacenar la base de datos empleada, de forma que, durante el desarrollo y ejecución del código, se ha trabajado directamente desde la nube vinculando Drive a Colab.

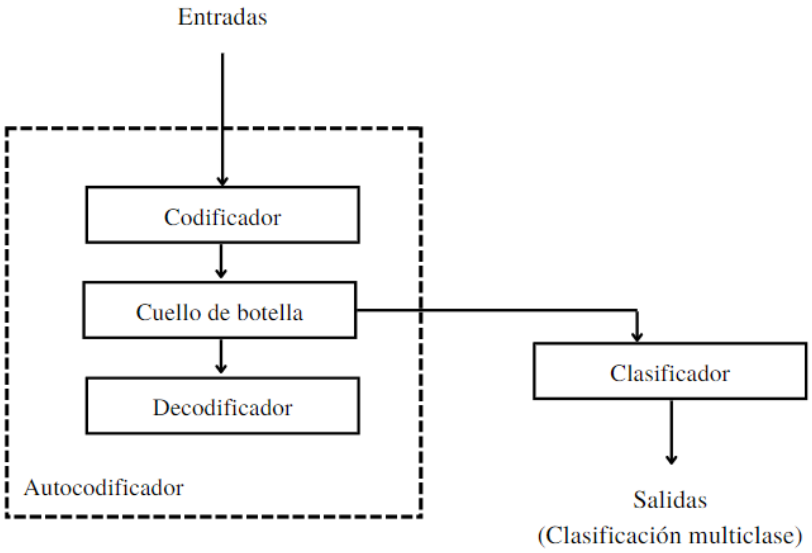


Figura 3.1: Esquema del problema planteado.

3.1 Base de datos ShipsEar

Se ha elegido la base de datos *ShipsEar*, que cuenta con un conjunto de grabaciones submarinas de sonidos de barcos, cedida por la Univerdidad de Vigo.

Las grabaciones fueron realizadas en la Ría de Vigo, lugar con un gran tráfico marítimo, y tienen duraciones de entre 15 segundos a 10 minutos. Se grabaron con un sistema que cuenta con tres hidrófonos submarinos [2], que detectan los sonidos transmitidos por el agua y los transforma en señales eléctricas. Además de los ruidos de los barcos, las grabaciones contemplan el ruido antropogénico y otros ruidos ocasionales de mamíferos marinos.

La base de datos se compone de un total de 90 archivos de audio de diferente duración divididos en 11 clases distintas de embarcaciones y una clase extra de ruido ambiental. Además, también se adjunta un fichero Excel con la nomenclatura seguida para nombrar los archivos de audio y otros parámetros de relevancia sobre la grabación de los mismos.

El número de audios según el tipo de embarcación se recoge en la Tabla 3.1. Los 12 audios restantes se corresponden con los audios de ruido ambiental, que son excluidos del conjunto de datos, ya que, al no tener información sobre sonidos procedentes de barcos, no se van a emplear para el problema de clasificación.

En los problemas de clasificación es importante que el número de datos por cada clase o resultado esté balanceado, ya que en el aprendizaje, el algoritmo podría tender a sobre-

Tipo de embarcación	Cantidad
Embarcación de pasajeros	30
Lancha motora	13
Transatlántico	7
Draga	5
Ro-Ro	5
Embarcación para cultivo del mejillón	5
Pesquero	4
Velero	4
Remolcador	2
Barco piloto	2
Arrastrero	1
TOTAL	78

Tabla 3.1: Número de audios por tipo de embarcación

ajustar hacia aquellos con más cantidad de datos. Así pues, al igual que se ha hecho en el problema de clasificación propuesto por los investigadores de la Universidad de Vigo [2], los 11 tipos de embarcaciones se han agrupado en un total de 4 clases según el tamaño de la embarcación, quedando recogido el total de audios por clase en el Tabla 3.2.

Clase	Embarcaciones	Cantidad
A	Pesqueros, arrastreros, barcos cultivo mejillón, remolcadores y dragas	17
B	Lanchas a motor, lanchas piloto y veleros	19
C	Embarcación de pasajeros	30
D	Transatlánticos y buques Ro-Ro	12
	TOTAL	78

Tabla 3.2: Número de audios por clases

3.2 Preprocesamiento

La cantidad y calidad de las señales empleadas para entrenar y evaluar los modelos de ML son factores clave para un buen rendimiento. Es por ello que el primer paso a la hora de desarrollar una solución es conocer en profundidad el conjunto de datos que componen la base de datos elegida para, según su forma y particularidades, aplicar el preprocesamiento más indicado.

En primer lugar, se ha empleado la librería *pydub* de *Python* para convertir todos los

ficheros de audio del formato wav inicial a formato FLAC (*Free Lossless Audio Codec*). Este formato permite obtener ficheros de audio comprimidos sin pérdidas, de forma que el tamaño del archivo se reduce considerablemente sin perder información en el proceso. Este cambio de formato ayuda a almacenar y cargar la base de datos de forma más rápida.

Una vez cargados los ficheros en Colab, se utiliza la librería *librosa* para leerlos y obtener la información de la serie temporal, así como de la tasa de muestreo empleada, que es de 52734 Hz.

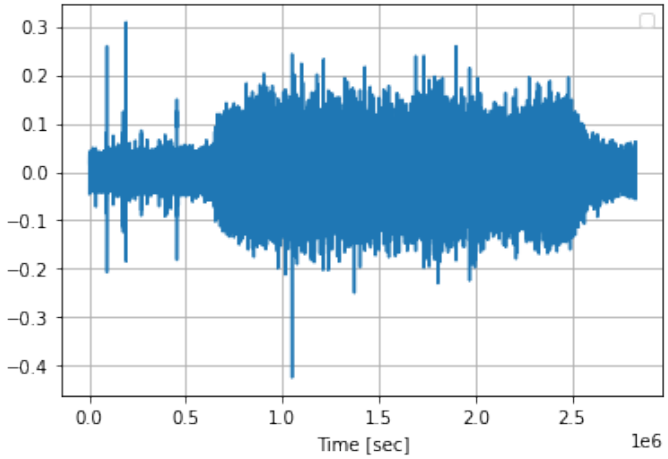


Figura 3.2: Señal completa de audio.

En este momento también se cargan las etiquetas. Según la clase a la que pertenezca el audio que se lea, se añade $(0, 1, 2, 3)$ a la variable que recoja todas las etiquetas para las clases (A, B, C, D) respectivamente.

En este punto se define la función *butter_lowpass_filter*, que filtra las secuencias con un filtro paso bajo Butterworth de orden 5 y frecuencia de corte 5 kHz. La frecuencia de corte se ha determinado en base a que la información útil de los sonidos de embarcaciones se concentran a bajas frecuencias, mientras que por encima de 8 kHz solo hay ruido [2]. De esta forma, eliminamos gran parte de ruido captado en las grabaciones.

El siguiente paso en el preprocesamiento es quedarnos con las muestras centrales de las series. Las grabaciones de audio comienzan momentos antes de percibir los ruidos que provocan las embarcaciones a fin de no perder información útil. Sin embargo, para nuestros problemas de clasificación, estos márgenes se traducen en ruido. Por tanto, los datos que introducimos a nuestro conjunto serán la mitad central de todas las series temporales.

A continuación se define la función *cut_seq*, que parte cada serie en divisiones de 13180

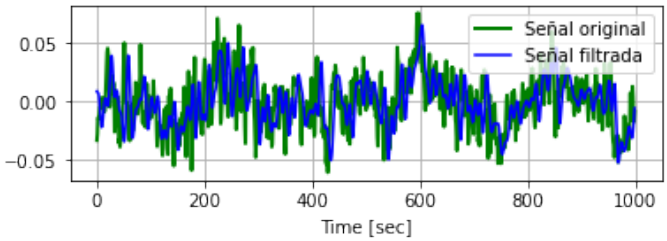


Figura 3.3: Efecto temporal del filtro paso bajo.

muestras y las recopila en la variable asignada al conjunto de datos. Con esta función hacemos que todos los datos introducidos a los modelos posteriores tengan el mismo tamaño, además de generar un mayor número de series temporales. En la Figura 3.4 se recoge el número total de series temporales que se han obtenido para cada una de las cuatro clases. Como se puede observar, el mayor conjunto de datos lo proporciona la Clase C, esto se debe a que la base de datos está ligeramente desbalanceada y cuenta con más minutos de grabación para esta clase.

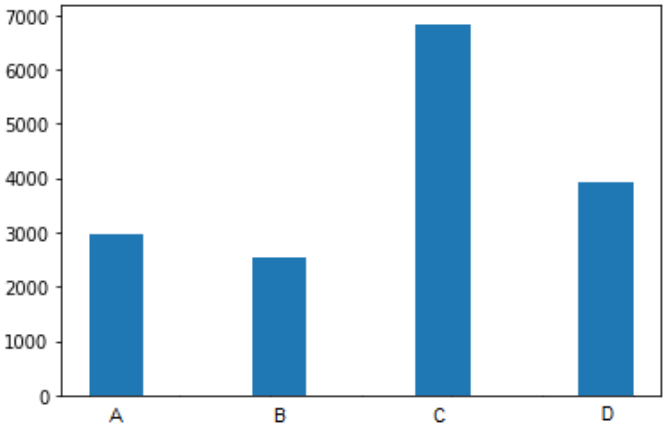


Figura 3.4: Distribución de los datos en clases.

Por último, se divide el conjunto de datos en dos grupos: un 60 % del total serán datos de entrenamiento, un 20 % datos de validación, y el 20 % restante, datos de prueba. Los datos de entrenamiento se usarán para entrenar la red y los datos de validación sirven para controlar el buen funcionamiento del modelo sin necesidad de ejecutarlo. Por otra parte, los datos de prueba sirven para obtener medidas finales de su eficiencia o desempeño. Cabe mencionar que como los datos estaban ordenados por clases (debido a la forma en la que se cargan los ficheros), la partición en entrenamiento, validación y prueba se ha programado

para tomar datos aleatoriamente de cada clase. Este punto es importante ya que cuando entrenemos la red, los datos no pueden presentarse ordenados por clases.

3.3 Extracción de características

A continuación se detalla la arquitectura del autocodificador que hemos utilizado para la extracción de características que servirán de entradas a los otros modelos de clasificación.

Se trata de un modelo funcional de autocodificador de reconstrucción convolucional, que se compone, como ya hemos adelantado, de dos subredes: codificador y decodificador.

El codificador comprime las entradas reduciendo la dimensionalidad con el cuello de botella y el decodificador vuelve a aumentar las dimensiones e intenta generar salidas lo más parecidas posibles a la entradas, partiendo de los datos comprimidos que le llegan.

Para obtener lo que serán las entradas del problema de clasificación, se entrena el autocodificador y luego se guarda el modelo del codificador. Las entradas van a ser esas representaciones que se obtienen en el espacio latente, por lo que la subred del decodificador se puede descartar una vez entrenado el autocodificador.

Esta red se va a realizar con la librería *Keras* de *Python*, que es una librería de redes neuronales que puede ejecutarse sobre *TensorFlow* y permite implementar redes profundas.

Arquitectura del codificador. El modelo de codificador convolucional tendrá entradas de tamaño 13180 muestras. La primera de las capas convolucionales tendrá 32 filtros, la segunda 16 y la tercera 8. En todas las capas se emplea kernel de tamaño 7 y un número de *strides* igual a 2. El *stride* indica el número de muestras que saltan los filtros mientras va recorriendo la entrada. Además, se indica "*same*" como parámetro de *zero-padding*. Este parámetro mantiene el tamaño de salida durante todas las capas, no se añaden ceros en los extremos. Como función de activación, se utiliza ReLU en todas las capas.

Cuello de botella. En el cuello de botella se quita dimensionalidad, se condensa el conjunto de características extraídas en un array. Para ello aplicamos una capa *Flatten* a la salida del codificador, y una capa *Dense* con función de activación ReLU. El factor de compresión resultante es 0.25, pasamos de tener una entrada de dimensión 13180 a una salida de 3295.

Arquitectura del decodificador. El decodificador cuenta con la estructura inversa al codificador. En primer lugar, la salida del cuello de botella se pasa por una capa de *Reshape* para volver a dotar de la dimensionalidad necesaria para poder aplicar los filtros convolucionales. Después de esta capa, se pasa por tres nuevas capas que realizar una

deconvolución o convolución inversa, la primera con 8 filtros, la segunda con 16 y la tercera con 32. El tamaño del kernel es 7, al parámetro *padding* se le indica "*same*", el *stride* tiene un valor de 2 y la función de activación, ReLU. Por último, como introdujimos en las CNN, se añade una última capa con solo 1 filtro, tamaño de kernel igual a 7 y *padding* igual a "*same*".

Entrenamiento. Una vez modelado codificador y decodificador, se modela el autocodificador. Las entradas del AE serán las entradas del codificador, mientras que las salidas serán las salidas del decodificador.

Para actualizar iterativamente los pesos de la red, se ha escogido el algoritmo de optimización Adam con un *learning rate* de 0,001.

Para obtener el error de reconstrucción se emplea la función correspondiente al error cuadrático medio (MSE), que se calcula como el promedio de los errores de reconstrucción al cuadrado [28]:

$$L_{mse} = \frac{1}{N}\sum_{i=1}^{N}(X_i - \hat{X}_i)^2 \tag{3.1}$$

Donde N representa el número de muestras, X_i es la secuencia de entrada y $\hat{X}_i$ la secuencia reconstruida.

La arquitectura final del autocodificador será la detallada en la Figura 3.5.

Como parte del entrenamiento, se ha aplicado el método de *Callback Early Stopping* para que la red finalice el entrenamiento cuando las *val_loss* no mejoren. Estas pérdidas proporcionan una guía para saber si se está produciendo sobreajuste.

Finalmente, la arquitectura del codificador se guarda y se utilizará en como primer paso en los problemas de clasificación para realizar la extracción de características para servir como entradas a los distintos clasificadores.

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 13180, 1)]        0

 conv1d (Conv1D)             (None, 6590, 32)          256

 conv1d_1 (Conv1D)           (None, 3295, 16)          3600

 conv1d_2 (Conv1D)           (None, 1648, 8)           904

 flatten (Flatten)           (None, 13184)             0

 dense (Dense)               (None, 3295)              43444575

 reshape (Reshape)           (None, 3295, 1)           0

 conv1d_transpose (Conv1DTra  (None, 6590, 8)          64
 nspose)

 conv1d_transpose_1 (Conv1DT  (None, 13180, 16)        912
 ranspose)

 conv1d_transpose_2 (Conv1DT  (None, 13180, 32)        3616
 ranspose)

 conv1d_transpose_3 (Conv1DT  (None, 13180, 1)         225
 ranspose)

=================================================================
Total params: 43,454,152
Trainable params: 43,454,152
Non-trainable params: 0
_________________________________________________________________
```

Figura 3.5: Modelo AE.

3.4 Técnicas de clasificación

Como se ha comentado, el problema de clasificación consiste en predecir etiquetas continuas a partir de un conjunto de datos de entrada. Para dicho fin, existen diversas técnicas que pueden ser empleadas. A continuación se detallan las características de cada uno de los modelos implementados para el problema de clasificación de series temporales. Dichos métodos pertenecen a lo que se conoce como *shallow methods*. Sin embargo, también se ha utilizado una técnica de DL, el perceptrón multicapa, el cual ha sido explicado en la Sección 2.

Para definir en *Python* los *shallow methods*, utilizamos la librería Sklearn. Esta librería cuenta con funciones para solucionar problemas de aprendizaje supervisado y no supervisado, además de funciones para el apartado de preprocesamiento y otras funciones para problemas de optimización.

Es necesario tener que en cuenta que en todos los métodos, se dividen los datos en dos grupos: entrenamiento y prueba. Por lo tanto, dicho paso no se explica en cada uno de ellos.

Árbol de decisión

Las entradas del modelo pueden ser tanto numéricas como categóricas. En nuestro caso, hemos seleccionado entradas numéricas, pues son las características obtenidas del AE. Las salidas del modelo son las etiquetas, definidas anteriormente: $(0, 1, 2, 3)$.

Creamos el modelo de árbol de decisión con la clase *DecisionTreeClassifier* del módulo *tree* de Sklearn. Este clasificador permite configurar una serie de parámetros, que son los siguientes:

- *criterion*: función para medir la calidad de una división. Por defecto se escoge la función de impurezas Gini.
- *splitter*: indica la estrategia que se sigue para elegir la división en cada nodo. Los valores posibles son "*best*" y "*random*"para elegir la mejor división o la mejor división aleatoria. Se escoge *best*.
- *max_depth*: profundidad máxima del árbol. En este caso no se ha indicado profundidad máxima, por lo que los nodos se expanden hasta que todos los nodos sean nodos hoja, o no se alcance el número mínimo de muestras requeridas para dividir el nodo.
- *min_samples_split*: número mínimo de muestras requeridas para dividir un nodo interno. Se toman 2 muestras (valor por defecto).
- *min_samples_leaf*: número mínimo de muestras requeridas en un nodo hoja. Por defecto, se inicializa a 1.
- *min_weight_fraction_leaf*: fracción mínima ponderada de la suma total de pesos requerida para un nodo hoja. Por defecto se indica 0,0.
- *max_features*: número de características a considerar cuando se busca la mejor división. Se tomará el número de características que se le pasa al modelo.
- *random_state*: Controla la aleatoriedad del estimador, ya que las funciones siempre se permutan aleatoriamente en cada división. Para obtener un comportamiento determinista durante el ajuste, se debe fijar a un número entero.

- *max_leaf_nodes*: número máximo de los mejores nodos hoja. Este parámetro no se completa, por lo que no hay un número limitado de nodos hoja.
- *min_impurity_decrease*: valor que limita si un nodo se divide o no según si produce una disminución de la impureza mayor o igual a este valor. Por defecto se indica 0.
- *class_weight*: pesos asociados a las clases. Se suponen todos los pesos igual a 1.
- *ccp_alpha*: parámetro de complejidad que se utiliza en la poda de mínimo coste-complejidad. La poda de mínimo coste-complejidad es un algortimo que "poda" un árbol para evitar el sobreajuste. En este trabajo no realizaremos podas, por lo que no se completa.

Random Forest

Este modelo se puede crear con la clase *RandomForestClassifier* del módulo *ensemble* de Sklearn. Los parámetros permitidos son los siguientes:

- *n_estimators*: número de árboles que componen el bosque. Se van a emplear 100 árboles.
- *criterion*: función para medir la calidad de una división. Por defecto se escoge la función de impurezas Gini.
- *max_depth*: profundidad máxima del árbol. En este caso no se ha indicado profundidad máxima, por lo que los nodos se expanden hasta que todos los nodos sean nodos hoja, o no se alcance el número mínimo de muestras requeridas para dividir el nodo.
- *min_samples_split*: número mínimo de muestras requeridas para dividir un nodo interno. Por defecto se toman 2 muestras.
- *min_samples_leaf*: número mínimo de muestras requeridas en un nodo hoja. Se indica 1.
- *min_weight_fraction_leaf*: fracción mínima ponderada de la suma total de pesos requerida para un nodo hoja. Se indica 0,0.
- *max_features*: número de características a considerar cuando se busca la mejor división. Se indica "*sqrt*", por lo que el número de características será la raíz del número de características que se le pase al método.
- *max_leaf_nodes*: número máximo de los mejores nodos hoja. Este parámetro no se completa, por lo que no hay un límite máximo.
- *min_impurity_decrease*: valor que limita si un nodo se divide o no según si produce una disminución de la impureza mayor o igual a este valor. Por defecto se indica 0.

- *bootstrap*: toma valor booleano, indica si se van a utilizar muestras de arranque para construir los árboles o si se indica falso, se utilizará todo el conjunto de datos para construir cada árbol. Se indica "*True*".
- *n_jobs*: número de procesos o subprocesos que se ejecutan de forma simultánea. Este parámetro no se completa.
- *random_state*: Controla tanto la aleatoriedad de las muestras al construir los árboles, como el muestreo de las características a considerar al buscar la mejor división en los nodos.
- *verbose*: habilita la salida detallada. Por defecto se indica 0.
- *class_weight*: pesos asociados a las clases. Se indica pesos igual a 1.
- *ccp_alpha*: parámetro de complejidad que se utiliza en la poda de mínimo coste-complejidad. La poda de mínimo coste-complejidad es un algortimo que "poda" un árbol para evitar el sobreajuste. No realizaremos podas, por lo que no se completa.
- *max_samples*: número de muestras empleadas para entrenar cada estimador. Se tomarán el número de la primera entrada de los datos de entrenamiento.

Máquina de vectores de soporte

Para la clasificación con máquinas de vectores de soporte, utilizamos la clase SVC (*Support Vector Classification*). Como nuestra clsificación es múltiple, SVC puede utilizar un enfoque "uno contra uno" o "uno contra el resto" a la hora de enfrentar las distintas clases para clasificar. Algunos de sus parámetros más importantes son los siguientes:

- *C*: parámetro de regularización, debe pasarse un valor positivo. La fuerza de regularización será inversamente proporcional a este parámetro. Se indica 1.
- *kernel*: indica el tipo de kernel que se emplea. Se toma "*rbf*".
- *degree*: grado de la función kernel cuando se indica como polinomial ("*poly*"). Se ignora en este caso.
- *gamma*: coeficiente kernel cuando se indica "*rbf*", "*poly*" o "*sigmoid*". El valor de este parámetro será de 0,5.
- *coef0*: término independiente de la función kernel. Solo es relevante si se indica función "*poly*" o "*sigmoid*". No se aplica.
- *tol*: tolerancia para el criterio de parada. Tomará un valor de 0,001.
- *cache_size*: tamaño de la memoria caché del kernel. Se indica 200 (valor por defecto).
- *class_weight*: pesos de cada clase que multipican al parámetro C. No se indica valor, por lo que por defecto se tomará valores iguales a 1.

- *verbose*: habilita la salida detallada.
- *decision_function_shape*: las opciones son "*ovo*" para un enfoque uno contra uno, y "*ovr*" para uno contra el resto. Por defecto, se toma "*ovr*".

AdaBoost

Este modelo se crea con la clase *AdaBoostClassifier* del módulo *ensemble* de Sklearn. Los parámetros son los siguientes:

- *base_estimator*: estimador o modelo base a partir del que se construye el conjunto potenciado. Como no se completa este parámetro, se toma el valor por defecto "*None*", y el estimador base será un árbol de decisión de profundidad igual a 1.
- *n_estimators*: número de modelos que necesita el conjunto. Se indican 50.
- *learning_rate*: peso que se aplica a cada clasificador en cada iteración. A mayor tasa, más contribución de cada clasificador. El valor indicado por defecto y que se usa es de 1.
- *algorithm*: dos valores posibles, "*SAMME*" y "*SAMME.R*" según el tipo de algoritmo a usar. Estos algoritmos son adaptaciones de AdaBoost extendiendo la funcionalidad a problemas multiclase. La diferencia entre ambos es que SAMME genera valores discretos y SAMME.R, probabilidades de pertenencia a una clase. Se utiliza SAMME.R.
- *random_state*: controla la aleatoriedad del estimador.

Perceptrón Multicapa

En último lugar, se ha implementado un modelo sencillo de MLP como clasificador para comprobar sus prestaciones.

Esta arquitectura cuenta con cuatro capas *Dense* de dimensiones de salida 512, 256, 128 y 64 y función de activación ReLU. Finalmente, la capa de salida es una *Dense* de dimensión 4 y función de activación *Softmax*, que será la capa que devuelva los porcentajes de la predicción de cada clase. La función de optimización empleada es Adam, con un *learning rate* de 0,0001.

Capítulo 4

Resultados y Discusión

En este capítulo, se presentan y analizan los resultados obtenidos en las clasificaciones de las señales acústicas mediante cinco modelos distintos de ML: *Random Forest*, árbol de decisión, SVM, AdaBoost y redes MLP.

Antes de mostrar los resultados obtenidos con estos modelos, se ilustrará el funcionamiento de los AE en su tarea propia de autocodificar. En la Figura 4.1 se representan las pérdidas de entrenamiento y validación obtenidas durante el entrenamiento.

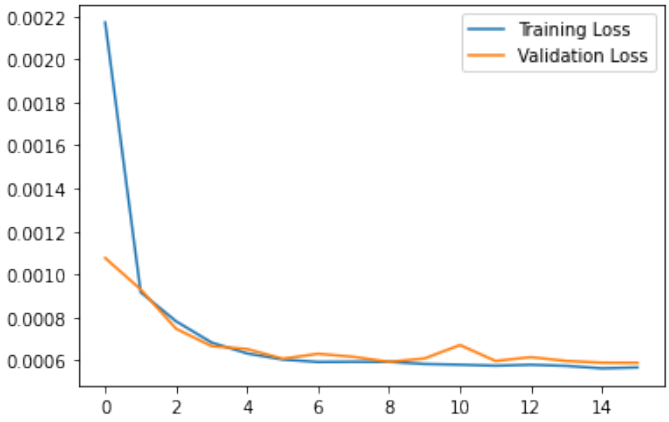
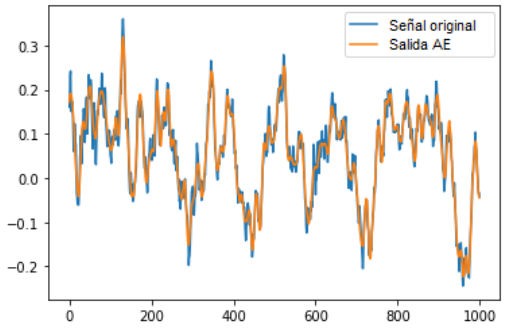
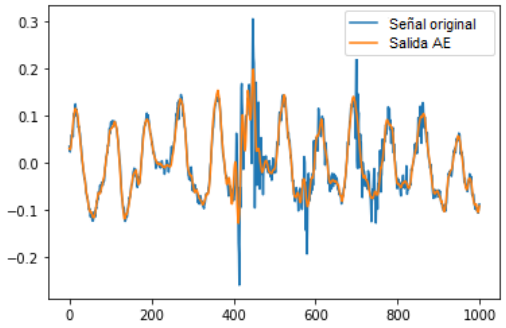


Figura 4.1: Pérdidas AE.

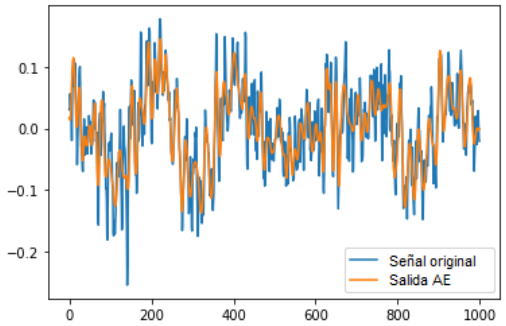
Además, podemos comprobar el efecto del AE en las señales representando gráficamente las entradas y las salidas del AE en una misma gráfica en la Figura 4.2. De esta Figura, se concluye el correcto funcionamiento de los AEs a la hora de producir salidas parecidas a las entradas.



(a)



(b)



(c)

Figura 4.2: Reconstrucción de señales por AE.

Así pues, el AE demuestra ser una buena opción de preprocesado en la eliminación de ruido de señales temporales. La Figura 4.1 muestra valores bajos de *loss* y, como se aprecia en la Figura 4.2, consigue reconstruir fielmente la señal de entrada a partir de las características extraídas en el espacio latente, eliminando el ruido y valores atípicos de las series.

4.1 Métricas empleadas

A continuación se detalla una comparativa entre las todas las métricas empleadas en la clasificación de audios submarinos, que son los cuatro tipos de *Shallow Methods* y un Perceptrón multicapa.

Las métricas que se van a recoger son las siguientes:

1. Precisión: es una medida del número de predicciones correctas realizadas. Se calcula como la suma de las entradas correctamente clasificados de cada clase c dividido por la suma de las entradas correctamente clasificados para c y los valores que se clasificaron como c sin ser de dicha clase (falso positivo). Como esta clasificación tiene un total de 4 clases, la expresión para calcular la precisión es la siguiente:

$$Precision = \frac{\sum_{c=1}^{4}(Valores\_correctos)_c}{\sum_{c=1}^{4}(Valores\_correctos)_c + (Falsos\_positivos)_c} \tag{4.1}$$

2. *Recall*: esta métrica indica el número de predicciones correctas conseguidas de todas las predicciones correctas que podrían haberse obtenido. Se calcula como la suma de las entradas correctamente clasificadas para la clase c dividido por la suma de los casos bien clasificados y los casos mal clasificados para la clase c (el total de etiquetas de la clase c). La expresión equivalente para 4 clases es la siguiente:

$$Recall = \frac{\sum_{c=1}^{4}(Valores\_correctos)_c}{\sum_{c=1}^{4}(Valores\_correctos)_c + (Falsos\_negativos)_c} \tag{4.2}$$

3. *F1-Score*: esta medida combina la precisión y el *Recall* y resume el rendimiento del modelo. Se calcula con la siguiente expresión:

$$F1Score = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{4.3}$$

4. *Accuracy*: esta medida indica la exactitud a la hora de clasificar. Se compara directamente la predicción obtenida con los resultados que deberían haberse obtenido y se calcula la precisión de la predicción.

5. Matriz de confusión: esta matriz mide la calidad de las predicciones indicando qué predicciones son verdaderas o correctas y cuántas son falsas. Si tenemos un total de cuatro clases, la dimensión de la matriz será de 4x4, correspondiendo cada fila y cada columna a una clase. La Figura 4.3 ilustra el formato de una matriz de confusión para este problema. En cada fila se indica el número de predicciones correctas en aquella posición en la que la clase coincida en fila y columna (resaltadas en azul en la Figura 4.3), siendo el resto falsas predicciones. Por columnas se visualiza la configuración total de predicciones realizadas para cada clase, mientras que las filas indican el número real de entradas por clases.

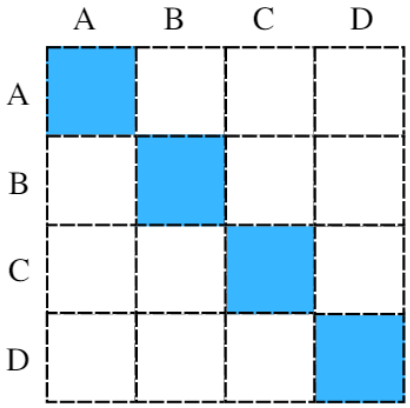


Figura 4.3: Formato de una matriz de confusión de cuatro clases.

4.2 Casos de estudio

Con el fin de analizar la importancia del AE en el problema de clasificación, así como de evaluar el efecto del FPB, se han estudiado un total de cuatros escenarios, detallados a continuación:

- Caso 1. Modelos de ML con datos en crudo.
- Caso 2. Modelos de ML aplicando en la fase de preprocesado un FPB a los datos en crudo.
- Caso 3. Modelos de ML aplicando en la fase de preprocesado en primer lugar un FPB, seguido del AE.
- Caso 4. Modelos de ML aplicando en la fase de preprocesado un AE.

Por otra parte, se ha calculado el tiempo empleado para entrenar y ejecutar cada modelo en cada uno de los cuatro casos. Al final de esta sección se presentan y discuten dichos tiempos.

A continuación se detallan los resultados numéricos de todos estos. Debido al carácter estocástico de algunos clasificadores, se ha realizado un total de cinco predicciones distintas para dichos clasificadores, mostrando el valor medio y desviación típica de los resultados obtenidos tras las cinco ejecuciones.

4.2.1 Caso 1: ML con datos en crudo

En este caso se analizan los resultados de ejecutar los métodos de ML estudiados con los datos en crudo (utilizando la parte central de los datos y cortando en 13180 *timesteps*), recogidos en las Tablas 4.1 a 4.10.

- Clasificador *Random Forest:*

Random Forest								
Clase	Precisión		*Recall*		*F1-Score*		*Accuracy*	
	Media	Desviación típica	Media	Desviación típica	Media	Desviación típica	Media	Desviación típica
A	0.915	0.018	0.593	0.002	0.720	0.006		
B	0.995	0.004	0.589	0.003	0.740	0.001		
C	0.710	0.004	0.940	0.005	0.809	0.003		
D	0.783	0.011	0.762	0.011	0.772	0.008		
							0.778	0.001

Tabla 4.1: Resultados para el clasificador *Random Forest* en Caso 1.

	A	B	C	D
A	455	0	225	90
B	5	368	223	26
C	27	3	1607	69
D	18	0	213	741

Tabla 4.2: Matriz de confusión *Random Forest* para Caso 1.

- Clasificador Árbol de decisión:

Árbol de decisión								
Clase	Precisión		*Recall*		*F1-Score*		*Accuracy*	
	Media	Desviación típica	Media	Desviación típica	Media	Desviación típica	Media	Desviación típica
A	0.655	0.016	0.680	0.010	0.667	0.012		
B	0.626	0.007	0.684	0.010	0.654	0.006		
C	0.786	0.004	0.748	0.003	0.767	0.003		
D	0.707	0.008	0.705	0.006	0.706	0.006		
							0.715	0.003

Tabla 4.3: Resultados para el clasificador Árbol de decisión en Caso 1.

	A	B	C	D
A	523	45	103	99
B	29	433	113	47
C	128	139	1285	154
D	99	63	126	684

Tabla 4.4: Matriz de confusión Árbol de decisión para Caso 1.

- Clasificador SVM:

SVM				
Clase	Precisión	*Recall*	*F1-Score*	*Accuracy*
A	1	0.474	0.643	
B	1	0.453	0.624	
C	0.588	1	0.741	
D	1	0.539	0.701	
				0.707

Tabla 4.5: Resultados para el clasificador SVM en Caso 1.

	A	B	C	D
A	365	0	405	0
B	0	282	340	0
C	0	0	1706	0
D	0	0	448	524

Tabla 4.6: Matriz de confusión SVM para Caso 1.

- Clasificador AdaBoost:

AdaBoost				
Clase	Precisión	*Recall*	*F1-Score*	*Accuracy*
A	0.288	0.022	0.041	
B	0.244	0.080	0.121	
C	0.449	0.746	0.561	
D	0.340	0.342	0.341	
				0.411

Tabla 4.7: Resultados para el clasificador AdaBoost en Caso 1.

	A	B	C	D
A	17	47	487	219
B	6	50	486	80
C	19	70	1272	345
D	17	38	585	332

Tabla 4.8: Matriz de confusión AdaBoost para Caso 1.

- Clasificador Red MLP:

Red MLP								
Clase	Precisión		*Recall*		*F1-Score*		*Accuracy*	
	Media	Desviación típica	Media	Desviación típica	Media	Desviación típica	Media	Desviación típica
A	0.691	0.002	0.699	0.001	0.695	0.001		
B	0.714	0.010	0.671	0.001	0.692	0.004		
C	0.783	0.002	0.809	0.002	0.796	0.002		
D	0.758	0.002	0.736	0.007	0.747	0.003		
							0.750	0.002

Tabla 4.9: Resultados para el clasificador MLP en Caso 1.

	A	B	C	D
A	538	30	115	87
B	41	416	131	34
C	105	101	1384	116
D	91	23	130	728

Tabla 4.10: Matriz de confusión Red MLP para Caso 1.

Para la clasificación con los datos "en crudo", los mejores resultados se han obtenido para el clasificador *Random Forest*, que presenta una media de 77,8 % de *accuracy*, las precisiones obtenidas varían entre 71 % (clase C) hasta un 99,5 % (clase B), y el *recall* es de 94 % para la clase C, mientras que en la clase B obtenemos el valor más bajo, de 58,9 %. Este valor indica la relación de entre el total de predicciones correctas que se pueden realizar para dicha clase y las que se han detectado. Esto significa que casi todas las predicciones hechas, por ejemplo, de la clase B, eran correctas (alta precisión), pero otras series de la clase B se han clasificado como otra clase por error (bajo *recall*). El *F1-Score*, que representa el rendimiento del algoritmo, ronda entre el 72 % y el 80,9 %.

El clasificador MLP presenta un rendimiento ligeramente inferior pero también favorable, entre el 69,2 % y un 79,6 %, y con una *accuracy* del 75 %.

En el resto de clasificadores se obtienen resultados de *accuracy* bastante similares. Para el Árbol de decisión y SVM, es de un 71,5 % y 70,7 % respectivamente, mientras que AdaBoost es el algoritmo con una media de *accuracy* más baja, de 41,1 %. Este clasificador parece más sensible frente a sobreajustes, ya que tiende a clasificar como clase C, que es la clase que más entradas tiene, debido a que la base de datos está desbalanceada.

4.2.2 Caso 2: ML aplicando un FPB

En este caso, se pasan los datos en crudo por un filtro paso bajo de Butterworth y la salida se usa como entrada para los clasificadores. Los resultados se recogen en las Tablas 4.11 a 4.20.

- Clasificador *Random Forest*:

Random Forest								
Clase	Precisión		*Recall*		*F1-Score*		*Accuracy*	
	Media	Desviación típica	Media	Desviación típica	Media	Desviación típica	Media	Desviación típica
A	0.878	0.017	0.620	0.009	0.727	0.011		
B	0.995	0.004	0.556	0.002	0.713	0.002		
C	0.707	0.004	0.928	0.005	0.803	0.001		
D	0.781	0.009	0.748	0.010	0.764	0.006		
							0.771	0.003

Tabla 4.11: Resultados para el clasificador *Random Forest* en Caso 2.

	A	B	C	D
A	421	0	173	74
B	8	382	273	28
C	25	1	1611	98
D	24	0	213	739

Tabla 4.12: Matriz de confusión *Random Forest* para Caso 2.

- Clasificador Árbol de decisión:

Árbol de decisión								
Clase	Precisión		*Recall*		*F1-Score*		*Accuracy*	
	Media	Desviación típica	Media	Desviación típica	Media	Desviación típica	Media	Desviación típica
A	0.626	0.007	0.704	0.003	0.663	0.004		
B	0.649	0.015	0.641	0.008	0.645	0.011		
C	0.780	0.005	0.756	0.007	0.768	0.006		
D	0.707	0.006	0.693	0.005	0.700	0.004		
							0.713	0.003

Tabla 4.13: Resultados para el clasificador Árbol de decisión en Caso 2.

	A	B	C	D
A	471	49	85	63
B	51	442	134	64
C	127	141	1318	149
D	95	49	150	682

Tabla 4.14: Matriz de confusión Árbol de decisión para Caso 2.

- Clasificador SVM:

SVM				
Clase	Precisión	*Recall*	*F1-Score*	*Accuracy*
A	0.963	0.388	0.553	
B	0.981	0.143	0.250	
C	0.565	0.943	0.706	
D	0.737	0.606	0.665	
				0.635

Tabla 4.15: Resultados para el clasificador SVM en Caso 2.

	A	B	C	D
A	288	1	403	78
B	2	98	512	10
C	0	0	1663	43
D	6	2	427	537

Tabla 4.16: Matriz de confusión SVM para Caso 2.

- Clasificador AdaBoost:

AdaBoost				
Clase	Precisión	*Recall*	*F1-Score*	*Accuracy*
A	0.268	0.029	0.052	
B	0.253	0.076	0.117	
C	0.456	0.741	0.564	
D	0.362	0.382	0.372	
				0.419

Tabla 4.17: Resultados para el clasificador AdaBoost en Caso 2.

	A	B	C	D
A	22	29	485	226
B	8	48	506	71
C	22	67	1266	353
D	30	46	522	369

Tabla 4.18: Matriz de confusión AdaBoost para Caso 2.

- Clasificador Red MLP:

Red MLP								
Clase	Precisión		*Recall*		*F1-Score*		*Accuracy*	
	Media	Desviación típica	Media	Desviación típica	Media	Desviación típica	Media	Desviación típica
A	0.669	0.004	0.664	0.004	0.667	0.001		
B	0.707	0.006	0.623	0.001	0.662	0.003		
C	0.733	0.002	0.803	0.002	0.766	0.001		
D	0.766	0.002	0.706	0.004	0.706	0.004		
							0.725	0.002

Tabla 4.19: Resultados para el clasificador MLP en Caso 2.

	A	B	C	D
A	500	35	130	88
B	41	391	169	28
C	115	105	1336	106
D	89	21	187	729

Tabla 4.20: Matriz de confusión Red MLP para Caso 2.

En este caso, los mejores resultados se obtienen para el clasificador *Random Forest*, que presenta una media de 77,1 % de *accuracy*. Este clasificador proporciona precisiones de entre 70,7 % (clase C) hasta un 99,5 % (clase B), al igual que en el caso anterior. El *recall* máximo es de 92,8 % para la clase C. El *F1-Score*, que representa el rendimiento del algoritmo, ronda entre el 71,3 % y el 80,3 %.

El clasificador MLP y el Árbol de decisión proporcionan valores similares, 72,5 % y 71,3 % respectivamente, mientras que en este caso con el AdaBoost se obtiene un 41,9 %, lo que cual indica que este clasificador, con los parámetros indicados, no es un buen clasificador para este problema.

4.2.3 Caso 3: ML aplicando un FPB y AE

Para este caso, se aplica un filtro paso bajo a los datos en crudo, y se utilizan los datos filtrados como entrada del autocodificador para entrenarlo. Las entradas de las técnicas de ML serán las características extraídas de AE. Los resultados se presentan en las Tablas 4.21 a 4.30.

- Clasificador *Random Forest*:

Random Forest								
Clase	Precisión		*Recall*		*F1-Score*		*Accuracy*	
	Media	Desviación típica	Media	Desviación típica	Media	Desviación típica	Media	Desviación típica
A	0.922	0.009	0.570	0.008	0.704	0.008		
B	0.993	0.003	0.541	0.007	0.701	0.006		
C	0.683	0.003	0.928	0.006	0.787	0.003		
D	0.762	0.013	0.745	0.008	0.753	0.009		
							0.756	0.003

Tabla 4.21: Resultados para el clasificador *Random Forest* en Caso 3.

	A	B	C	D
A	512	36	102	120
B	32	436	120	34
C	164	174	1250	118
D	113	24	58	777

Tabla 4.22: Matriz de confusión *Random Forest* para Caso 3.

- Clasificador Árbol de decisión:

Árbol de decisión								
Clase	Precisión		*Recall*		*F1-Score*		*Accuracy*	
	Media	Desviación típica	Media	Desviación típica	Media	Desviación típica	Media	Desviación típica
A	0.645	0.014	0.642	0.004	0.643	0.008		
B	0.629	0.014	0.632	0.004	0.630	0.008		
C	0.751	0.006	0.772	0.003	0.762	0.003		
D	0.719	0.007	0.687	0.005	0.702	0.004		
							0.705	0.002

Tabla 4.23: Resultados para el clasificador Árbol de decisión en Caso 3.

	A	B	C	D
A	486	41	131	95
B	55	398	131	45
C	99	124	1286	153
D	95	52	170	709

Tabla 4.24: Matriz de confusión Árbol de decisión para Caso 3.

- Clasificador SVM:

SVM				
Clase	Precisión	*Recall*	*F1-Score*	*Accuracy*
A	1	0.526	0.689	
B	1	0.523	0.687	
C	0.592	1	0.744	
D	1	0.526	0.690	
				0.719

Tabla 4.25: Resultados para el clasificador SVM en Caso 3.

	A	B	C	D
A	396	0	357	0
B	0	329	300	0
C	0	0	1662	0
D	0	0	486	540

Tabla 4.26: Matriz de confusión SVM para Caso 3.

- Clasificador AdaBoost:

AdaBoost				
Clase	Precisión	*Recall*	*F1-Score*	*Accuracy*
A	0.283	0.023	0.042	
B	0.558	0.046	0.085	
C	0.466	0.816	0.593	
D	0.464	0.472	0.468	
				0.464

Tabla 4.27: Resultados para el clasificador AdaBoost en Caso 3.

	A	B	C	D
A	17	9	504	223
B	4	29	541	55
C	15	8	1357	282
D	24	6	512	484

Tabla 4.28: Matriz de confusión AdaBoost para Caso 3.

- Clasificador Red MLP:

Red MLP								
Clase	Precisión		*Recall*		*F1-Score*		*Accuracy*	
	Media	Desviación típica	Media	Desviación típica	Media	Desviación típica	Media	Desviación típica
A	0.679	0.077	0.499	0.044	0.573	0.047		
B	0.737	0.058	0.309	0.082	0.430	0.087		
C	0.616	0.042	0.882	0.039	0.724	0.025		
D	0.753	0.052	0.625	0.073	0.680	0.042		
							0.657	0.036

Tabla 4.29: Resultados para el clasificador MLP en Caso 3.

	A	B	C	D
A	404	16	207	126
B	29	253	310	37
C	62	55	1400	145
D	100	15	167	744

Tabla 4.30: Matriz de confusión Red MLP para Caso 3.

Para este caso, el mejor clasificador es también el *Random Forest*. Presenta un valor de *accuracy* del 75,6 %, un valor máximo de precisión de 99,3 % para la clase B, que presenta un *recall* de 54,1 % y un rendimiento del 70 %. El valor máximo de *recall* lo presenta la clase C, de un 92,8 %, mientras que su precisión es del 68,3 % y su rendimiento de un 78,7 %.

El clasificador SVM, a pesar de que las entradas que clasifica como A, B y C son todas correctas, (precisión igual a 1), no predice correctamente la totalidad de las entradas que tienen dichas clases, y por ello el valor de *recall* oscila en el 52,6 %.

En cuanto al resto de clasificadores, el árbol de decisión y la red MLP obtienen un valor de *accuracy* del 70,5 % y 65,7 % respectivamente, mientras que el AdaBoost realiza predicciones mejorables, obteniendo en cambio un 46,4 %.

4.2.4 Caso 4: ML con salidas de AE

En este caso, se estudian los resultados en las Tablas de 4.31 a 4.40 tras realizar la clasificación con las técnicas de ML siendo las entradas el resultado de aplicar extracción de características con un AE entrenado con los datos en crudo.

- Clasificador *Random Forest*:

Random Forest								
Clase	Precisión		*Recall*		*F1-Score*		*Accuracy*	
	Media	Desviación típica	Media	Desviación típica	Media	Desviación típica	Media	Desviación típica
A	0.918	0.003	0.585	0.005	0.714	0.005		
B	0.991	0.007	0.590	0.003	0.740	0.004		
C	0.707	0.005	0.945	0.006	0.809	0.003		
D	0.819	0.016	0.781	0.006	0.800	0.006		
							0.784	0.003

Tabla 4.31: Resultados para el clasificador *Random Forest* en Caso 4.

En la Tabla 4.32 se muestra la matriz de confusión para este caso.

	A	B	C	D
A	454	0	236	80
B	3	364	240	15
C	19	3	1624	60
D	17	1	192	762

Tabla 4.32: Matriz de confusión *Random Forest* para Caso 4.

- Clasificador Árbol de decisión:

Árbol de decisión								
Clase	Precisión		*Recall*		*F1-Score*		*Accuracy*	
	Media	Desviación típica	Media	Desviación típica	Media	Desviación típica	Media	Desviación típica
A	0.643	0.010	0.657	0.008	0.650	0.007		
B	0.642	0.012	0.655	0.009	0.649	0.006		
C	0.765	0.004	0.755	0.008	0.760	0.005		
D	0.716	0.004	0.709	0.008	0.713	0.004		
							0.710	0.002

Tabla 4.33: Resultados para el clasificador Árbol de decisión en Caso 4.

	A	B	C	D
A	520	36	131	83
B	31	427	120	44
C	137	114	1297	158
D	90	47	135	700

Tabla 4.34: Matriz de confusión Árbol de decisión para Caso 4.

- Clasificador SVM:

SVM				
Clase	Precisión	*Recall*	*F1-Score*	*Accuracy*
A	1	0.451	0.621	
B	0.940	0.352	0.512	
C	0.600	0.982	0.742	
D	0.910	0.631	0.743	
				0.701

Tabla 4.35: Resultados para el clasificador SVM en Caso 4.

	A	B	C	D
A	347	3	382	38
B	0	219	398	5
C	0	9	1676	21
D	0	2	357	613

Tabla 4.36: Matriz de confusión SVM para Caso 4.

- Clasificador AdaBoost:

AdaBoost				
Clase	Precisión	*Recall*	*F1-Score*	*Accuracy*
A	0.340	0.108	0.164	
B	0.453	0.291	0.354	
C	0.501	0.701	0.589	
D	0.482	0.500	0.490	
				0.482

Tabla 4.37: Resultados para el clasificador AdaBoost en Caso 4.

	A	B	C	D
A	83	22	422	243
B	8	181	407	26
C	75	166	1215	250
D	78	31	379	484

Tabla 4.38: Matriz de confusión AdaBoost para Caso 4.

- Clasificador Red MLP:

Red MLP								
Clase	Precisión		*Recall*		*F1-Score*		*Accuracy*	
	Media	Desviación típica	Media	Desviación típica	Media	Desviación típica	Media	Desviación típica
A	0.601	0.095	0.668	0.066	0.624	0.036		
B	0.648	0.077	0.692	0.041	0.665	0.026		
C	0.801	0.032	0.706	0.118	0.745	0.067		
D	0.763	0.041	0.780	0.042	0.770	0.004		
							0.714	0.043

Tabla 4.39: Resultados para el clasificador MLP en Caso 4.

	A	B	C	D
A	446	22	170	132
B	22	387	193	20
C	111	91	1407	97
D	63	14	121	774

Tabla 4.40: Matriz de confusión Red MLP para Caso 4.

En este caso de estudio, los mejores resultados se han obtenido de igual forma para el clasificador *Random Forest*, con una media de 78,4 % de *accuracy*, precisiones de entre 71 % (clase C) hasta un 99,1 % (clase B), y un *recall* de 94,5 % para la clase C, mientras que en la clase B obtenemos el valor más bajo, de 59 %. El rendimiento del algoritmo ronda entre el 71,4 % y el 80,9 %.

En cuanto al resto de clasificadores, se obtienen medias de *accuracy* bastante similares en el Árbol de decisión, SVM y la red MLP, en torno al 71 %, mientras que el AdaBoost es de nuevo el algoritmo que arroja peores resultados, con una media de *accuracy* de 48,2 %.

4.2.5 Tiempo de ejecución

Una parte importante en la realización de este trabajo ha sido la limitación de recursos *hardware* disponibles. Para el entrenamiento y ejecución de las redes, se ha optado por utilizar la herramienta de *Google Colaboratory*, junto con la suscripción a su servicio Pro para contar con más recursos y hacer menores los tiempos que tardan las redes en ser entrenadas y realizar las predicciones. Sin embargo, durante las pruebas en los diseños de las redes se ha superado la capacidad de la memoria RAM proporcionada y los tiempos de ejecución se volvían considerables mientras se intentaba mejorar las prestaciones de las redes. Es por ello que se ha realizado también un seguimiento de los tiempos de ejecución de cada resultado para cada clasificador, con el fin de analizar rendimiento de las redes en función del tiempo de espera necesario.

En la Tabla 4.41 se recoge el tiempo en segundos empleado por cada método para definir el clasificador, entrenarlo y devolver una predicción, introduciendo 12208 muestras en el entrenamiento.

	Caso 1	Caso 2	Caso 3	Caso 4
Random Forest	207	206	52	88
Árbol de decisión	363	337	24	25
SVM	6238	5874	4750	4800
AdaBoost	1742	1648	121	119
Red MLP	332	297	185	240

Tabla 4.41: Tiempos de ejecución por método (en segundos).

Para los Casos 1 y 2, se ha tardado un tiempo considerablemente mayor en obtener los resultados que no distan tanto de los obtenidos en los Casos 3 y 4. El motivo es que en los casos 3 y 4, las entradas de los métodos de ML son las obtenidas mediante las técnicas de extracción de características. Cada una de las entradas tendrán un número de muestras de 3295, mientras que en los casos 1 y 2 se procesan muestras de tamaño 13180, lo que genera modelos más complejos.

Además, si comparamos la exactitud obtenida por el mejor clasificador, el *Random Forest*, para cada caso junto a los tiempos de ejecución, como se visualiza en la Tabla 4.42, vemos que se consiguen prestaciones mejores con AEs en tiempos considerablemente menores.

	Accuracy	Tiempo de ejecución (s)
Caso 1	77,8 %	207
Caso 2	77,1 %	206
Caso 3	75,6 %	58
Caso 4	78,4 %	88

Tabla 4.42: *Accuracy* y tiempos de ejecución por caso para *Random Forest*

Capítulo 5

Conclusiones

La clasificación de series acústicas puede abordarse mediante técnicas de *Machine Learning* (ML). Dichas técnicas permiten, además, realizar una reducción de dimensionalidad de los datos de entrada, haciendo uso de los *Autoencoders* (AE). Esto nos permite desarrollar modelos más simples, haciendo más extensa su utilidad y aplicación.

En este Trabajo Fin de Grado se ha analizado el uso de AE para la tarea de clasificación de series temporales, combinando su uso con las técnicas de ML, y se ha comparado su funcionamiento respecto al uso de técnicas de ML sin reducción de dimensionalidad. Además, se han aplicado filtros a las señales originales para comparar los resultados.

Tras los resultados obtenidos, podemos ver que para todos los casos estudiados tenemos un mejor desempeño con el clasificador *Random Forest*. Además, aunque los resultados están bastante ajustados, se consigue una mayor media de acierto, de un 78,4 %, cuando introducimos como entradas de los clasificadores las señales introducidas por el codificador del AE, además de haber reducido la dimensionalidad de las entradas y conseguir mejores tiempos de ejecución en los clasificadores. Sin embargo, también se obtienen buenos resultados con clasificadores Perceptrón Multicapa, obteniendo un 75 % para el primer caso estudiado, donde las entradas son las series temporales sin filtrar y sin codificar.

Podemos concluir, por tanto, que no hay un único procedimiento para obtener buenos resultados en un problema de clasificación de señales acústicas de embarcaciones. Analizando la base de datos con la que se cuente y su comportamiento en distintos modelos de redes, así como implementando redes profundas, se podría llegar a obtener resultados muy fiables en un problema de clasificación de series temporales como este.

También apreciamos que el AE es un buen modelo para preprocesar datos, tanto para eliminar ruido como por su capacidad de extracción de la esencia de las entradas, y por ello mismo podría adaptarse a señales más complejas en las que sería más costoso aplicar métodos tradicionales de eliminación de ruido como los filtros.

Sin embargo, quedan varios aspectos a mejorar, así como trabajos futuros que se podrían abordar. El rendimiento de los modelos de clasificación se podría mejorar desarrollando modelos más complejos que permitan mejorar la tarea de clasificación. Además, sería interesante aplicar técnicas de *Deep Learning* (DL), que permitan abordar el tarea como un problema de clasificación de series temporales mono-variables. El entrenamiento simultáneo de AE y modelos de ML es otra opción interesante a considerar.

Bibliografía

[1] Gordon M. Wenz. Review of Underwater Acoustics Research: Noise. *The Journal of the Acoustical Society of America*, 51(3B):1010–1024, March 1972. Publisher: Acoustical Society of America.

[2] David Santos-Domínguez, Soledad Torres-Guijarro, Antonio Cardenal-López, and Antonio Pena-Gimenez. ShipsEar: An underwater vessel noise database. *Applied Acoustics*, 113:64–69, December 2016.

[3] Laurent Fillinger, Pascal de Theije, Mario Zampolli, Alexander Sutin, Hady Salloum, Nikolay Sedunov, and Alexander Sedunov. Towards a passive acoustic underwater system for protecting harbours against intruders. In *2010 International WaterSide Security Conference*, pages 1–7, November 2010. ISSN: 2166-1804.

[4] Alexander Sutin, Barry Bunin, Alexander Sedunov, Nikolay Sedunov, Laurent Fillinger, Mikhail Tsionskiy, and Michael Bruno. Stevens Passive Acoustic System for underwater surveillance. In *2010 International WaterSide Security Conference*, pages 1–6, November 2010. ISSN: 2166-1804.

[5] Rajkumar Palaniappan, Kenneth Sundaraj, and Sebastian Sundaraj. A comparative study of the svm and k-nn machine learning algorithms for the diagnosis of respiratory pathologies using pulmonary acoustic signals. *BMC Bioinformatics*, 15(1):223, June 2014.

[6] Richard L Bankert, Michael Hadjimichael, Arunas P Kuciauskas, KA Richardson, F Joseph Turk, and Jeffrey D Hawkins. Automating the estimation of various meteorological parameters using satellite data and machine learning techniques. In *Frontiers Of Remote Sensing Information Processing*, pages 227–252. World Scientific, 2003.

[7] Erik Brynjolfsson, Tom Mitchell, and Daniel Rock. What can machines learn, and what does it mean for occupations and the economy? In *AEA papers and proceedings*, volume 108, pages 43–47, 2018.

[8] Pierre Geurts. Pattern Extraction for Time Series Classification. In Luc De Raedt and Arno Siebes, editors, *Principles of Data Mining and Knowledge Discovery*, Lecture Notes in Computer Science, pages 115–127, Berlin, Heidelberg, 2001. Springer.

[9] Jakob Abeßer, Stylianos Ioannis Mimilakis, Robert Gräfe, Hanna M Lukashevich, and IDMT Fraunhofer. Acoustic scene classification by combining autoencoder-based dimensionality reduction and convolutional neural networks. In *DCASE*, pages 7–11, 2017.

[10] Dengyu Xiao, Chengjin Qin, Honggan Yu, Yixiang Huang, Chengliang Liu, and Jianwei Zhang. Unsupervised machine fault diagnosis for noisy domain adaptation using marginal denoising autoencoder based on acoustic signals. *Measurement*, 176:109186, 2021.

[11] David Santos-Domínguez, Soledad Torres-Guijarro, Antonio Cardenal-López, and Antonio Pena-Gimenez. Shipsear: An underwater vessel noise database. *Applied Acoustics*, 113:64–69, 2016.

[12] Stuart J. Russell, Peter Norvig, and Ernest Davis. *Artificial intelligence: a modern approach*. Prentice Hall series in artificial intelligence. Prentice Hall, Upper Saddle River, 3rd ed edition, 2010.

[13] Jafar Alzubi, Anand Nayyar, and Akshi Kumar. Machine Learning from Theory to Algorithms: An Overview. *Journal of Physics: Conference Series*, 1142:012012, November 2018.

[14] T. Soni Madhulatha. AN OVERVIEW ON CLUSTERING METHODS. *IOSR Journal of Engineering*, 02(04):719–725, April 2012.

[15] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project Adam: Building an Efficient and Scalable Deep Learning Training System. page 13.

[16] Underfitting vs. Overfitting — scikit-learn 0.15-git documentation.

[17] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[18] Alaa Sagheer and Mostafa Kotb. Unsupervised Pre-training of a Deep LSTM-based Stacked Autoencoder for Multivariate Time Series Forecasting Problems. *Scientific Reports*, 9(1):19038, December 2019.

[19] Tung Kieu, Bin Yang, Chenjuan Guo, and Christian S. Jensen. Outlier Detection for Time Series with Recurrent Autoencoder Ensembles. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, pages 2725–2732, Macao, China, August 2019. International Joint Conferences on Artificial Intelligence Organization.

[20] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. Introduction to Information Retrieval. page 320, 2009.

[21] Yanli Liu, Yourong Wang, and Jian Zhang. New Machine Learning Algorithm: Random Forest. In David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Baoxiang Liu, Maode Ma, and Jincai Chang, editors, *Information Computing and Applications*, volume 7473, pages 246–252. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. Series Title: Lecture Notes in Computer Science.

[22] Boosting Algorithms in Python.

[23] Hassan Ismail Fawaz, Germain Forestier, Jonathan Weber, Lhassane Idoumghar, and Pierre-Alain Muller. Deep learning for time series classification: a review. *Data Mining and Knowledge Discovery*, 33(4):917–963, July 2019. arXiv:1809.04356 [cs, stat].

[24] Yuzhe Yang, Kaiwen Zha, Ying-Cong Chen, Hao Wang, and Dina Katabi. Delving into Deep Imbalanced Regression. *arXiv:2102.09554 [cs]*, May 2021. arXiv: 2102.09554.

[25] IBM. ¿Qué son las redes neuronales?, September 2021.

[26] Ihab S Mohamed. Detection and Tracking of Pallets using a Laser Rangefinder and Machine Learning Techniques. page 77.

[27] Saleh Albelwi and Ausif Mahmood. A Framework for Designing the Architectures of Deep Convolutional Neural Networks. page 20, 2017.

[28] Javier Naranjo-Alcazar, Sergi Perez-Castanos, Pedro Zuccarello, Fabio Antonacci, and Maximo Cobos. Open Set Audio Classification Using Autoencoders Trained on Few Data. *Sensors*, 20(13):3741, July 2020.