Trabajo Fin de Máster
Ingeniería de Telecomunicación

Autoencoders variacionales para la detección de ataques en redes de telecomunicaciones

Autor: José Manuel Gata Romero

Tutor: Juan José Murillo Fuentes

Cotutor: José Carlos Aradillas Jaramillo

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

Sevilla, 2021

Departamento de Teoría de la Señal y Comunicaciones

Trabajo Fin de Máster
Ingeniería de Telecomunicación

Autoencoders variacionales para la detección de ataques en redes de telecomunicaciones

Autor:
José Manuel Gata Romero

Tutor:
Juan José Murillo Fuentes
Catedrático de Universidad

Cotutor:
José Carlos Aradillas Jaramillo
Predoctoral PIF FPU Ministerio

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

Sevilla, 2021

Trabajo Fin de Máster: Autoencoders variacionales para la detección de ataques en redes de telecomunicaciones

Autor: José Manuel Gata Romero
Tutor: Juan José Murillo Fuentes
Cotutor: José Carlos Aradillas Jaramillo

El tribunal nombrado para juzgar el trabajo arriba indicado, compuesto por los siguientes profesores:

Presidente:

Vocal/es:

Secretario:

acuerdan otorgarle la calificación de:

El Secretario del Tribunal

Fecha:

Resumen

El Internet de las Cosas (IoT) es ya una realidad y su importancia no dejará de crecer en los próximos años. Cualquier "cosa" es susceptible de ser conectada a través de Internet (sensores, cámaras, dispositivos personales, etc) y sus campos de aplicación son también muy diversos: desde la agricultura o la automatización del hogar, hasta la comunicación entre vehículos o el monitoreo remoto de pacientes.

A medida que la información intercambiada en estas redes IoT vaya adquiriendo un mayor valor y relevancia, serán fruto de ataques cada vez más diversos y complejos.

El objetivo de este trabajo es ver cómo la Inteligencia Artificial, más concretamente el Deep Learning, puede ser aplicado para resolver estos problemas. Se estudiará e implementará una arquitectura de red neuronal de gran importancia dentro de este campo, denominada Autoencoder Variacional Condicional (CVAE), con el fin de realizar una clasificación de diferentes tipos de ataques.

Además, se comparará el rendimiento de dicho modelo con otros ya clásicos del Machine Learning, mostrando las diferentes ventajas que ofrece.

Abstract

The Internet of Things (IoT) is already a reality and its importance will not stop growing in the next years. Any "thing" is susceptible to be connected through Internet (sensors, cameras, personal devices, etc) and its application fields are also very diverse: from the agriculture or home automation, to communication between vehicles or remote patient monitoring.

As the exchanged information in these IoT networks gain a greater value and relevance, they will be target of more diverse and complex attacks.

The aim of this work is to see how the Artificial Intelligence, more specifically the Deep Learning, can be applied to solve these troubles. A neural network architecture of great importance within of this field will be studied and implemented, called Conditional Variational Autoencoder (CVAE), with the purpose to realize a classification of different types of attacks.

Furthermore, the performance of this model will be compared with others Machine Learning classics, showing the different advantages that it offers.

Índice

1 Introducción

La dependencia del ser humano de la tecnología en la vida cotidiana no para de aumentar. A través de las redes de telecomunicaciones, se intercambia información de prácticamente cualquier naturaleza, por lo que garantizar que se realice con seguridad es uno de los aspectos claves a tener en cuenta.

1.1 Detección de Intrusiones

Una intrusión es un conjunto de acciones deliberadas y no autorizadas cuyo objetivo es comprometer la seguridad de un ordenador u otros componentes de red en términos de confidencialidad, integridad y disponibilidad. El agente encargado de llevar a cabo una intrusión es denominado intruso, y puede distinguirse entre intrusos internos o externos [1, 2].

Los intrusos internos son aquellos que cuentan con acceso al sistema, pero los privilegios que poseen no se corresponden con el acceso que realizan. Por otra parte, los intrusos externos ni siquiera cuentan con acceso al sistema en cuestión.

Para proteger a los equipos y mantener una red a salvo de posibles atacantes, surgen los Sistemas de Detección de Intrusiones (IDSs). Entre las funciones principales de estos sistemas, se encuentran la recopilación y análisis del tráfico presente en un equipo o una red, la configuración de subsistemas para generar informes sobre posibles vulnerabilidades existentes, o el reconocimiento de patrones de una serie de ataques típicos.

Evidentemente, la teoría de detección de intrusiones supone que las actividades intrusivas son notablemente diferentes del resto de actividades habituales del sistema y, por lo tanto, pueden ser detectadas.

1.1.1 Clasificación de los IDSs

Los Sistemas de Detección de Intrusiones pueden clasificarse en dos categorías fundamentales, en función de su implementación en la realidad: los basados en equipo y los basados en red [1, 2].

Un Sistema de Detección de Intrusiones en el Equipo (HIDS) monitoriza y analiza el funcionamiento interno de un sistema de cómputo, sin prestar atención a sus interfaces externas. Por ejemplo, un HIDS podría detectar qué programa propio de un determinado ordenador intentó acceder a una serie de recursos de memoria de forma ilegítima, sin tener en cuenta la política de seguridad establecida por el sistema operativo.

Por otra parte, un Sistema de Detección de Intrusiones en la Red (NIDS) es un sistema encargado de detectar actividades maliciosas e intrusivas, o violaciones de políticas, en las diferentes interfaces de comunicación de un equipo o una red de equipos. Generalmente, el principal ataque al que tiene que hacer frente un NIDS, es el de un atacante externo que quiere conseguir acceder a un determinado sistema para robar información o dejarlo inutilizado. Por ejemplo, en un caso típico de una red conectada al "resto del mundo" a través de Internet, un NIDS debe ser capaz de identificar como una posible intrusión la llegada masiva de peticiones de conexión TCP a un gran número de puertos diferentes en un período de tiempo minúsculo. En este caso concreto, se trata de un tipo de ataque denominado *port scan*.

Al final, el objetivo de los IDSs, que llevan siendo un campo de investigación activa muchos años, es tener sistemas rápidos y de gran precisión que sean capaces de analizar el tráfico intercambiado, ya sea interno o en la red, y poder predecir posibles amenazas.

Conforme el volumen, diversidad y valor de los datos en cuestión continúa creciendo, la importancia de estos sistemas también lo hace, en redes como las IoT (Internet of Things), que pueden trabajar con datos críticos procedentes de múltiples escenarios de uso.

1.1.2 Técnicas de detección

Profundizando en el campo de los NIDSs, que es en el que se situará este trabajo, existen diferentes técnicas para la detección de intrusiones [1, 2]. Según el mecanismo de detección empleado, es posible clasificarlas en métodos de detección basados en firma y métodos de detección basados en anomalía.

Los métodos de detección basados en firma emplean una base de datos que contiene patrones intrusivos identificados previamente y, a partir de la misma, identifican e informan cuando se está ante un ataque. Es decir, la detección está basada en un conjunto de reglas o firmas mediante las cuales es posible detectar todos los patrones de ataques ya conocidos. Uno de los desafíos más importantes de esta técnica de detección es el hecho de conseguir establecer reglas que abarquen el mayor número de variaciones posibles de un determinado ataque ya presente en los datos de referencia, puesto que por su propia naturaleza cuenta con el inconveniente de la incapacidad de detectar nuevos ataques no presentes en la base de datos.

Por otra parte, los métodos de detección basados en anomalía utilizan un modelo, generalmente sustentado en métodos de Machine Learning, mediante el cual clasifican el tráfico como bueno o malo. Como se explicó anteriormente, se trabaja con la suposición de que cualquier actividad intrusiva es necesariamente anómala. Por tanto, este modelo construye un perfil de comportamiento normal y, cuando el estado del sistema varía una cantidad estadísticamente significativa respecto a este perfil establecido, informa que se está ante una intrusión. El principal problema de este método es que puede darse el hecho de que comportamientos anómalos que no sean intrusiones sean clasificados como tal, a lo cual se denomina falsos positivos. Para tratar los dos aspectos anteriores, es clave establecer una serie de umbrales cuyo objetivo sea el maximizar el número de intrusiones detectadas minimizando el número de falsos positivos.

Una característica importante de los modelos de Machine Learning empleados para este método de detección es la necesidad de trabajar con datos no balanceados. En cualquier sistema, las intrusiones suelen ser una excepción, y presentan una gran dificultad para ser distinguidas del tráfico normal, mucho más abundante. Esto supone un importante reto tanto para los algoritmos de predicción como para las propias métricas empleadas para la evaluación de sus rendimientos, puesto que puede dar lugar a extraer conclusiones engañosas sino se analiza con la suficiente profundidad.

Actualmente, la mayor parte de la investigación en el campo de la detección de intrusiones está concentrada en los métodos de detección basados en anomalía para intrusiones en la red, ya que permiten detectar tanto ataques ya conocidos como nuevos ataques.

Cabe destacar que, a la hora de modelar un problema de detección de anomalías de este tipo, pueden adoptarse diferentes enfoques: métodos probabilísticos, métodos de clustering o métodos de desviación. En los métodos probabilísticos, se caracteriza la distribución de probabilidad de los datos normales y se define como una anomalía cualquier dato con una probabilidad dada menor que cierto umbral. En los métodos de clustering, se agrupan los datos y se clasifica como una anomalía cualquier dato demasiado alejado de la agrupación de datos normales en cuestión. Por último, en los métodos de desviación, se define un modelo que permite reconstruir los datos normales y, se considera una anomalía, a cualquier dato cuya reconstrucción difiera suficientemente respecto al dato original de partida. Al igual que en ambos métodos anteriores, es necesario fijar un determinado umbral que establezca la distinción entre lo que será categorizado como un dato normal o uno anómalo.

1.2 Objetivos y estructura del trabajo

El objetivo de este trabajo es el estudio e implementación del método presentado en [2], el cual es denominado ID-CVAE (Intrusion Detection CVAE), para poder llevar a cabo una tarea de clasificación de ataques. Se trata de un método basado en un Autoencoder Variacional Condicional (CVAE), donde las etiquetas de las intrusiones son incluidas dentro de las capas del decodificador CVAE.

Este método cuenta con numerosas ventajas frente a otros similares, como el Autoencoder Variacional (VAE) y, además, proporciona mejores resultados de clasificación que otros clasificadores ampliamente conocidos (Random Forest, Support Vector Machine, Regresión Softmax, Perceptrón Multicapa), como se demostrará durante el desarrollo del trabajo.

Enlazando con lo explicado anteriormente en este Capítulo 1, el ID-CVAE es un método de detección basado en anomalía y modelado desde un enfoque de desviación, aunque con ciertas modificaciones que serán explicadas en su momento. En la fecha de su publicación, fue un trabajo único dentro del campo de la detección de intrusiones en la red, ya que suponía la primera aplicación de un CVAE en el mismo y ofrecía el primer algoritmo capaz de realizar recuperación de características.

Para alcanzar el objetivo fijado, se organiza el trabajo de la siguiente forma: tras este Capítulo 1 de introducción, en el Capítulo 2 se comenzará dando una visión general sobre la Inteligencia Artificial (AI), Machine Learning y Deep Learning, explicando diferentes ideas claves, así como algunos algoritmos tradicionales de estos campos. En el Capítulo 3, se estudiarán en profundidad los autoencoders, haciendo énfasis en los variacionales y sus derivados, como el CVAE. Además, se expondrá de forma general el método en cuestión, el ID-CVAE, que no es más que un caso particular de este tipo de arquitecturas.

Ya en el Capítulo 4, se mostrará el trabajo práctico desarrollado en torno al mismo, presentando los resultados más importantes y su comparativa con otros métodos, hasta finalizar extrayendo una serie de conclusiones en el Capítulo 5.

2 Inteligencia Artificial

La Inteligencia Artificial es uno de los campos en pleno auge sobre el que se está realizando una mayor investigación en los últimos años. En estas próximas páginas, se explicarán una serie de conceptos fundamentales del mismo cuyo conocimiento es imprescindible para cualquier persona interesada en él: desde sus diferentes enfoques (especialmente Machine Learning y Deep Learning) hasta varios algoritmos típicos del subcampo del Machine Learning que serán utilizados posteriormente en el trabajo.

2.1 Visión general

Gracias a algunos escritos, se tiene constancia que al menos ya en la época de la Antigua Grecia, existían personas que soñaban con crear máquinas capaces de "pensar". Sin embargo, no es hasta la década de 1950 cuando la Inteligencia Artificial comienza a experimentar una gran evolución, convirtiéndose en un campo similar a lo que se conoce hoy en día [3, 4].

2.1.1 Inteligencia Artificial

Existen múltiples definiciones para explicar qué es la Inteligencia Artificial, pero se puede definir de forma clara como el esfuerzo por automatizar tareas intelectuales realizadas habitualmente por humanos. Se trata de un campo muy extenso, con diferentes enfoques (ver Figura 2.1) que se irán explicando a continuación, y que abarcan desde el Machine Learning o Deep Learning a otros que no requieren ningún tipo de aprendizaje [3, 4].

En sus inicios, rápidamente abordó con éxito problemas que son intelectualmente difíciles para una persona pero relativamente sencillos para un ordenador. Estos son problemas acotados, bien definidos, donde no se requiere que el ordenador tenga un gran conocimiento del "mundo", y que se pueden describir por una serie de reglas formales, codificadas a mano por programadores. A este enfoque de la AI, se le conoce como AI simbólica y fue empleada, por ejemplo, en los primeros programas de ajedrez. El ajedrez es un juego sujeto a una serie de restricciones, donde solo existen 64 posiciones en el tablero y 32 piezas que pueden moverse de la forma establecida. Aunque diseñar una estrategia ganadora sea un gran logro, hay que tener presente que la dificultad no está en cómo describir el conjunto de piezas y movimientos permitidos al ordenador, ya que esto puede ser hecho fácilmente mediante una lista de reglas formales proporcionadas por los programadores con anticipación.

Este paradigma dominó la AI hasta finales de la década de 1980, donde se comenzaron a ver sus limitaciones para solucionar problemas más complejos y difusos, que los seres humanos resuelven de manera intuitiva pero son muy difíciles de articular formalmente. Irónicamente, ya en la década de 1990, un ordenador podía batir al mejor jugador humano de ajedrez del mundo, sin embargo tenía dificultades para reconocer la voz de una persona, una cara en una imagen, clasificar objetos, etc. Para dar solución a este tipo de problemas, surgió un nuevo enfoque, el conocido Machine Learning.

2.1.2 Machine Learning

El Machine Learning [3, 4], dentro de la AI, tiene el objetivo de proporcionar a los ordenadores la capacidad de aprender a realizar una determinada tarea, de adquirir su propio conocimiento para poder llevar a cabo la misma a partir de unos datos de entrada sin procesar. Este enfoque supuso una revolución, ya que mostró un

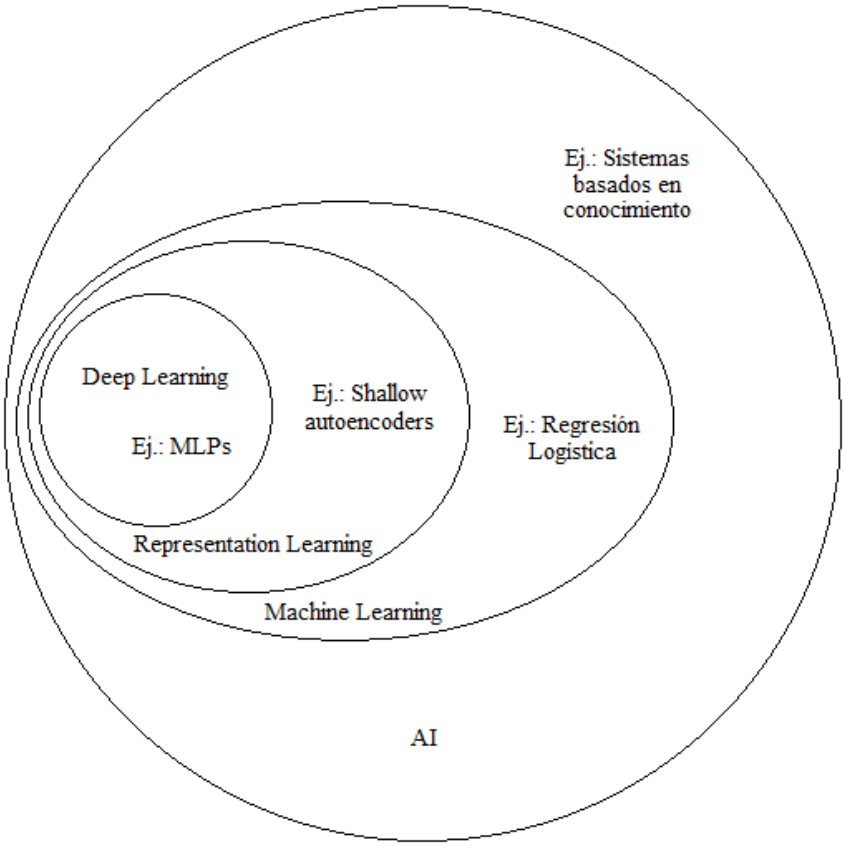

Figura 2.1 Diagrama de Venn que muestra las relaciones entre los diferentes enfoques de la AI, junto con un ejemplo de tecnología asociada a cada uno de ellos.

nuevo paradigma en el mundo de la programación. Hasta entonces, en la programación tradicional, empleada por ejemplo en la AI simbólica, los programadores introducían una serie de reglas (lo que sería realizar un programa) y un conjunto de datos que serían procesados conforme a las mismas, para así obtener respuestas. Con el Machine Learning, se pasa a introducir dicho conjunto de datos y las respuestas esperadas una vez procesados, y lo que se obtienen son las reglas.

Estos dos paradigmas se recogen en la Figura 2.2, donde se observan claramente las diferencias de forma gráfica, mediante diagramas de bloques.

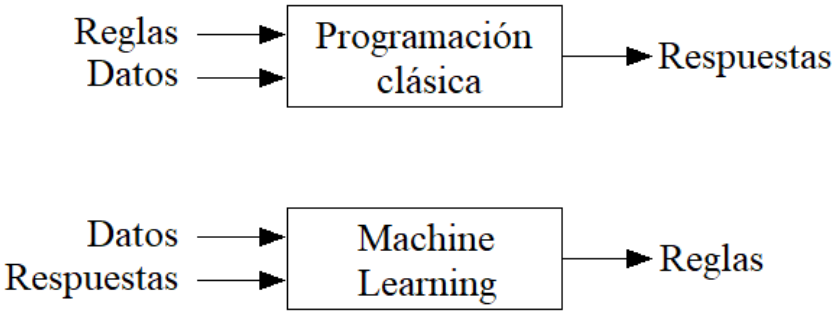

Figura 2.2 Machine Learning: un nuevo paradigma de programación.

Se dice que un algoritmo de Machine Learning es entrenado, no explícitamente programado. A partir de los datos de entrada, trata de encontrar una estructura estadística en los mismos que le permite generar las reglas para poder automatizar la tarea en cuestión. Para saber si el aprendizaje se está produciendo adecuadamente,

es necesario tener una forma de medir las diferencias entre la salida dada por el algoritmo para un cierto ejemplo y la salida que cabía esperar. Esta medida se usa como señal de realimentación para ir ajustando el algoritmo durante lo que se denomina la fase de entrenamiento.

Representaciones de los datos

En general, el rendimiento de los algoritmos de Machine Learning presenta una gran dependencia respecto a la representación de los datos de entrada [3, 4]. Dicha representación, que no es más que una forma concreta de ver los datos, queda determinada por las características que los componen. A partir de los datos en bruto, los algoritmos se encargan de transformarlos a nuevas representaciones que sean útiles para conseguir realizar cierta tarea objetivo. Dichos algoritmos no llevan a cabo las transformaciones de una forma arbitraria, sino que buscan la mejor opción entre un conjunto predefinido de posibilidades, lo cual es conocido como espacio de hipótesis.

En la Figura 2.3 puede observarse un ejemplo sencillo del impacto de la representación sobre el rendimiento de un determinado algoritmo. Se trata de dos representaciones diferentes de unos mismos datos, tan solo cambiando el sistema de coordenadas de referencia. Si se quisieran separar los dos tipos de muestras presentes en los datos simplemente utilizando una línea recta, para el gráfico de la izquierda (coordenadas cartesianas) la tarea sería imposible, mientras que para el de la derecha (coordenadas polares) sería inmediato con una recta vertical.

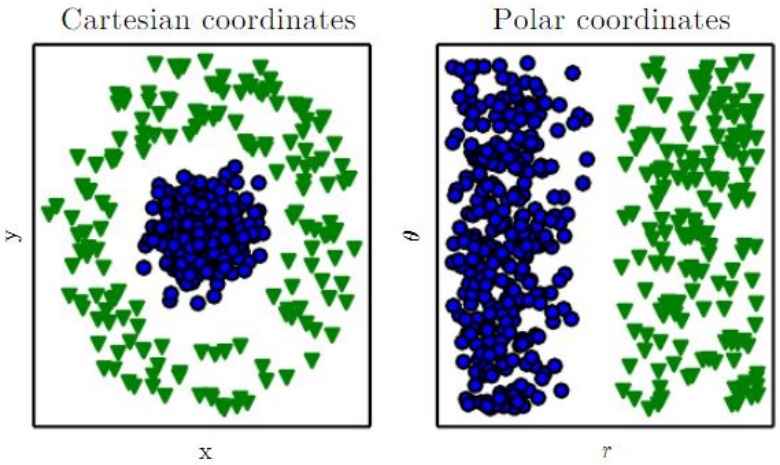

Figura 2.3 Diferentes representaciones de los mismos datos en función del sistema de coordenadas de referencia [4].

Como ha quedado reflejado con el ejemplo anterior, existen tareas que pueden ser muy difíciles de ejecutar, incluso imposibles, con una representación pero que pueden resultar extremadamente sencillas con otra. Por tanto, diseñar el conjunto correcto de características que deben tener los datos para llevar a cabo una cierta tarea es uno de los desafíos más importantes a los que hay que enfrentarse cuando se trabaja en el desarrollo de un sistema de Machine Learning. Este proceso de trabajo con las características es denominado feature engineering [3, 5], y comprende desde seleccionar aquellas más útiles de entre las disponibles o combinar algunas existentes para dar lugar a otras más significativas, hasta recopilar información con la que poder crear nuevas características.

El problema principal del feature engineering es que existen multitud de tareas para las que es excesivamente complicado saber qué características son relevantes, y pueden pasar años hasta que toda una comunidad de investigadores en el campo en cuestión descubran dichas claves. Para evitar esto, la solución es utilizar el Machine Learning para descubrir la representación en sí misma, es decir, aplicarlo para obtener la representación adecuada de los datos de entrada. Este enfoque dentro del Machine Learning es conocido como Representation Learning [4], y puede disminuir los tiempos de una manera abrumadora respecto al caso en el que el conjunto de características es diseñado manualmente. Además, generalmente las representaciones aprendidas suelen dar un mejor rendimiento cuando se emplean en el sistema de Machine Learning posterior de lo que resultaban las diseñadas a mano.

Aunque con este último enfoque pudiera parecer que ya se tienen las suficientes herramientas para afrontar satisfactoriamente cualquier problema dentro de la Inteligencia Artificial, esto no es así. En muchas ocasiones,

resulta tan difícil para un algoritmo de Machine Learning obtener una buena representación de los datos de entrada como resolver el problema original y, entonces, el Representation Learning es de poca utilidad. Aquí es donde toma presencia el conocido Deep Learning, que será explicado a continuación.

2.1.3 Deep Learning

El Deep Learning es un subcampo específico del Machine Learning [3]. Se trata de una nueva forma de aprender representaciones de los datos en capas sucesivas de representaciones, que van desde menos a más significativas. El número de capas presentes en un determinado modelo es lo que se conoce como profundidad del mismo. Actualmente, un modelo de Deep Learning puede contar con decenas o incluso centenas de capas de representaciones, todas ellas aprendidas de forma automática a partir de los datos de entrenamiento que se le proporcionan al modelo. Esto marca las diferencias con otros enfoques del Machine Learning, donde habitualmente solo existen una o dos capas de representaciones para los datos.

Las representaciones por capas en Deep Learning son normalmente aprendidas a través de las famosas redes neuronales artificiales (ANNs). Se trata de estructuras que cuentan con una serie de capas apiladas unas sobre otras, cuya inspiración viene del campo de la neurociencia y el estudio del cerebro. No obstante, cabe destacar que aunque algunos de los conceptos de dichos campos hayan tenido cierta influencia en el desarrollo del Deep Learning, no existen evidencias de que el cerebro lleve a cabo su aprendizaje como estos modelos desarrollados hoy en día.

En la Figura 2.4 se puede observar la apariencia típica de un modelo de Deep Learning. Se trata de una red neuronal con varias capas de profundidad, que transforma una imagen de un dígito con el objetivo de reconocer de qué dígito se trata.

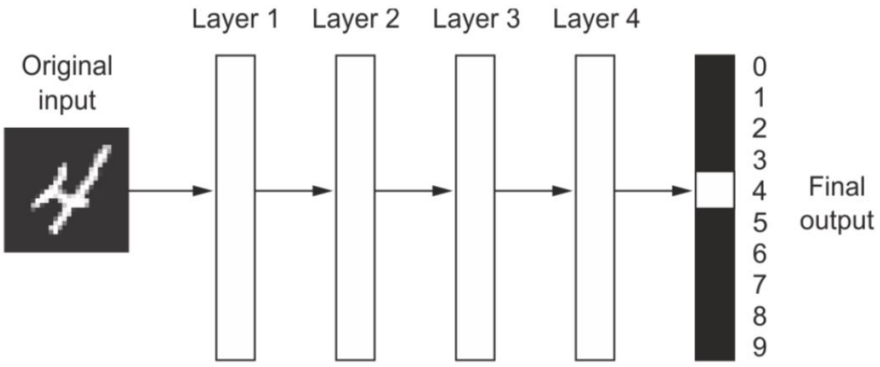

Figura 2.4 Red neuronal profunda para clasificación de dígitos [3].

Dicha red, como refleja la Figura 2.5, transforma la imagen del dígito en diferentes representaciones, cada cual más diferente de la original y cada vez más significativas para lograr llevar a cabo la tarea en cuestión. Una ANN puede ser vista como un proceso de múltiples etapas donde la información va pasando por diferentes filtros que la van haciendo cada vez más útil para conseguir el resultado esperado.

Cómo trabaja el Deep Learning

Una vez son conocidas las ideas básicas de este campo, hay que profundizar en cómo un modelo de Deep Learning aprende estas representaciones sucesivas de los datos (ver Figura 2.6) [3].

La transformación que lleva a cabo una capa sobre sus datos de entrada queda parametrizada por los pesos de la capa, que son básicamente un conjunto de números configurados en matrices o vectores. Por tanto, el proceso de aprendizaje consiste en encontrar el conjunto de valores para los pesos de las diferentes capas de la red, de tal forma que los datos de entrenamiento sean correctamente mapeados a la salida correspondiente. Como se dijo en la subsección anterior, para ello es necesario medir las diferencias entre la salida actual y la salida esperada, de lo cual se encarga la función de pérdida (también llamada función objetivo) de la red. Esta función compara las dos salidas y calcula un cierto score, que ofrece un índice de cómo de bien la red ha actuado ante una muestra específica.

El score obtenido es usado como señal de realimentación para ajustar los pesos adecuadamente, en una dirección en la cual se produzca un decremento del mismo para la muestra en cuestión (matemáticamente, esta dirección se corresponde con el gradiente de la función de pérdida con respecto a los parámetros de

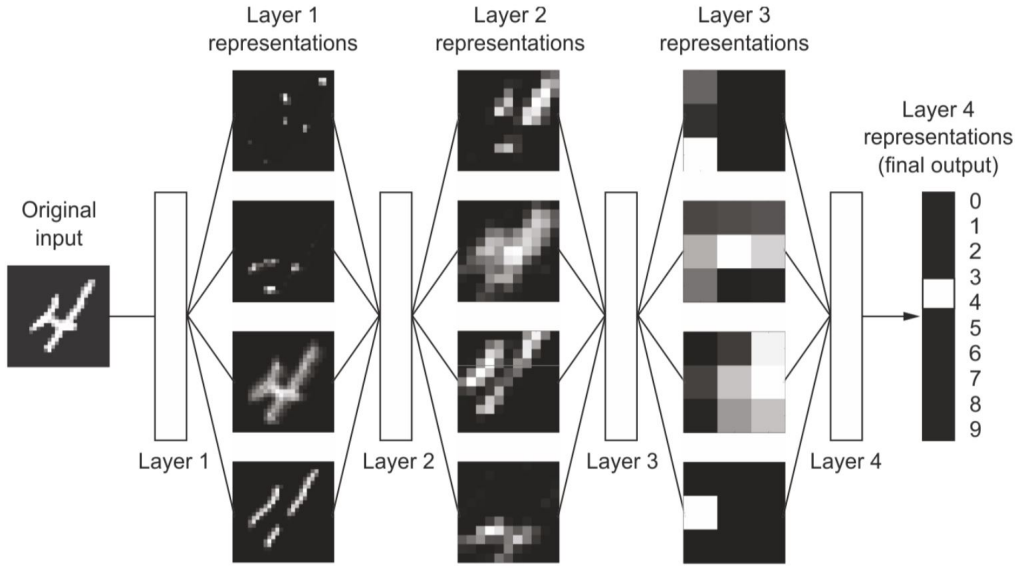

Figura 2.5 Representaciones profundas aprendidas por un modelo de clasificación de dígitos [3].

la red). Esta es la labor del optimizador, la cual realiza implementando el algoritmo de Backpropagation, algoritmo fundamental en Deep Learning.

Básicamente, el funcionamiento del algoritmo de Backpropagation es el siguiente. En primer lugar, los pesos de las diferentes capas de la red son inicializados aleatoriamente, por lo que las transformaciones que implementa cada una también lo son. Para una determinada muestra de entrenamiento, el algoritmo realiza la predicción (Forward Pass) y calcula el score mediante la función de pérdida. Evidentemente, para este primer caso dicho score es grande, ya que al haber sido los pesos inicializados aleatoriamente la salida actual está lejos de la que debería ser obtenida. Una vez obtenido esto, el algoritmo recorre las capas en orden inverso calculando la influencia que cada uno de los pesos ha tenido en el error final (Reverse Pass) y los ajusta proporcionalmente en la dirección adecuada (Gradient Descent Step). Este bucle es ejecutado para cada muestra de entrenamiento y, repetido un número suficiente de veces, acaba conduciendo a valores de pesos que minimizan el score. Cuando este proceso finaliza, se dice que la red está entrenada.

El Deep Learning, aún siendo un campo conocido desde hace bastantes años, no ha comenzado a adquirir una enorme relevancia hasta esta última década, donde ha logrado resultados importantes en problemas de percepción como el reconocimiento de voz o la clasificación de imágenes, que hasta ahora habían sido altamente complicados de resolver por las máquinas. Existen varios factores que han impulsado claramente el desarrollo reciente del campo, entre los que destacan los avances en el hardware, en los sets de datos y en los propios algoritmos [3, 4].

El hardware y los datos eran el principal cuello de botella que se encontraban los investigadores hace un par de décadas. Al ser un campo de gran carga práctica, orientado a ingeniería, a menudo las demostraciones son realizadas empíricamente más que teóricamente, por lo que solo es posible avanzar cuando se dispone de los recursos apropiados para poder probar las ideas.

En la actualidad, gracias al incremento en la potencia de los ordenadores y la existencia de una cantidad abismal de datos derivada del mundo de Internet, es posible entrenar ANNs mucho mayores en menor cantidad de tiempo y probar nuevas ideas o recuperar antiguas que no habían podido ser llevadas a la práctica. Paralelamente, también se han realizado grandes mejoras en los algoritmos de entrenamiento, haciendo todo ello al Deep Learning en particular, y al Machine Learning en general, los campos más populares y exitosos dentro de la Inteligencia Artificial.

2.2 Tipos de sistemas de Machine Learning

El Machine Learning es un campo de estudio muy amplio, con sistemas muy diversos, por lo que resulta útil establecer diferentes categorías que permitan estar situado en todo momento. Generalmente, los sistemas de Machine Learning se suelen clasificar en función de los siguientes aspectos [5]:

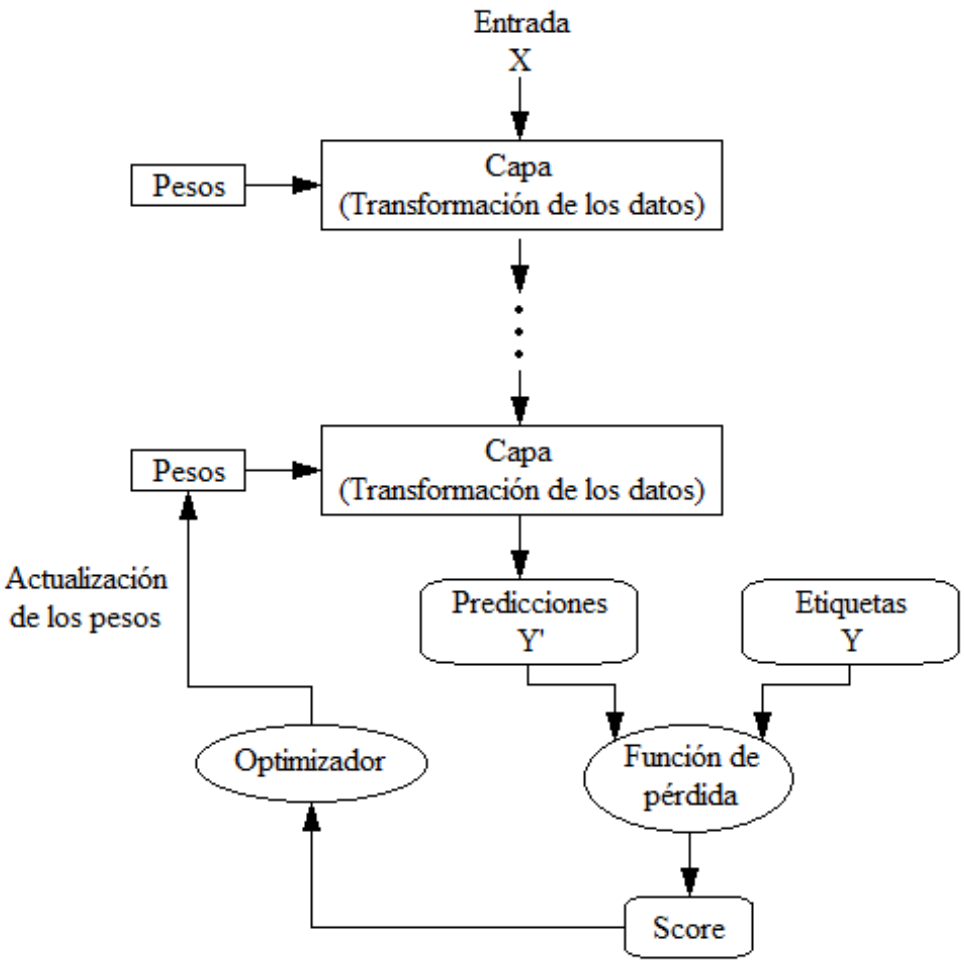

Figura 2.6 Modelo funcional del proceso de aprendizaje en Deep Learning.

- Según su supervisión durante la fase de entrenamiento (supervisados, no supervisados, semisupervisados, por refuerzo).
- Según puedan o no aprender gradualmente a partir de un flujo de datos de entrenamiento entrante (aprendizaje online vs batch).
- Según su forma de predecir ante nuevas muestras, es decir, cómo llevan a cabo la generalización (aprendizaje basado en instancia vs basado en modelo).

De cara a este trabajo, es importante tener claras las diferencias atendiendo al primer criterio, por lo que será explicado más en detalle a continuación [3, 5].

2.2.1 Aprendizaje supervisado

El aprendizaje supervisado es aquel en el que los datos de entrenamiento con los que trabaja el sistema están etiquetados, es decir, incluyen los objetivos. El algoritmo aprende a mapear los datos de entrenamiento a sus etiquetas correspondientes y, posteriormente, puede predecir una nueva etiqueta para un nuevo dato no presente durante la fase de entrenamiento.

Entre las tareas más típicas de este tipo de aprendizaje se encuentran la clasificación y la regresión, que pueden realizarse mediante algoritmos como las Support Vector Machines (SVMs), la Regresión Lineal, los Random Forests, etc.

Cabe destacar que, hoy en día, la mayoría de aplicaciones existentes en Deep Learning son aplicaciones de aprendizaje supervisado, como la que se verá posteriormente en este trabajo, una tarea de clasificación de ataques multiclase.

2.2.2 Aprendizaje no supervisado

A diferencia del caso supervisado, en aprendizaje no supervisado los datos de entrenamiento no están etiquetados. Los algoritmos no supervisados tratan de encontrar representaciones útiles de los datos de entrada sin tener un objetivo de referencia, para lo cual, intentan aprender la distribución de probabilidad generadora de los datos o algunas propiedades interesantes de la misma.

En cuanto a las tareas más típicas, están las de clustering, con algoritmos como el k-Means, o aquellas para la reducción de la dimensionalidad y la visualización de los datos, como el Principal Component Analysis (PCA).

Además, este tipo de aprendizaje es utilizado en múltiples ocasiones como paso previo a una tarea de aprendizaje supervisado con el fin de conseguir un mejor conocimiento de los datos, para lo cual se emplean técnicas como la PCA recientemente mencionada.

2.2.3 Aprendizaje semisupervisado

Existen sistemas intermedios entre los dos vistos hasta ahora, y son aquellos conocidos como sistemas de aprendizaje semisupervisado. Estos sistemas implementan algoritmos que pueden tratar con datos de entrenamiento parcialmente etiquetados (los datos no etiquetados son mucho más numerosos que los etiquetados, al ser la etiqueta un "añadido" a los mismos) y, habitualmente, dichos algoritmos están basados en combinaciones de los de los casos anteriores.

2.2.4 Aprendizaje por refuerzo

El aprendizaje por refuerzo es una rama del Machine Learning que aún no ha conseguido éxitos al nivel de las anteriormente explicadas, pero recientemente ha comenzado a adquirir bastante importancia a raíz de varias aplicaciones en el terreno de los juegos y se espera que tenga un gran futuro en campos como la robótica.

En este tipo de aprendizaje, un agente, que es como se conoce al sistema de aprendizaje en este contexto, se encuentra observando el entorno y debe seleccionar qué acciones realiza ante una determinada situación. Dependiendo de si la acción tomada es o no la que debía llevar a cabo, recibe una recompensa o una penalización, la cual le permite aprender por sí mismo cuál es la mejor estrategia ante una situación dada para maximizar las recompensas. Básicamente, el agente va aprendiendo mediante prueba y error.

2.3 Algoritmos típicos de Machine Learning

Existen una serie de algoritmos de gran popularidad dentro del Machine Learning que presentan un rendimiento notable en aplicaciones de muy diversa naturaleza. En esta sección, se explicarán algunos de los mismos, como son la Regresión Softmax, las SVMs y los Random Forests, así como se darán ciertos detalles acerca de su implementación en la librería Scikit-Learn [6, 7] (concretamente en su versión 0.22.2.post1), mediante la cual serán empleados en el Capítulo 4 para una tarea de clasificación de ataques.

2.3.1 Regresión Softmax

A pesar de que aparezca la palabra regresión en su nombre, la Regresión Softmax es un algoritmo utilizado fundamentalmente para clasificación. Este modelo es la generalización al caso multiclase de la conocida Regresión Logística, en la que solo se pueden resolver de forma directa tareas de clasificación binaria [5].

En una clasificación binaria, la Regresión Logística permite estimar la probabilidad de que una muestra pertenezca a una determinada clase. Suponiendo que A y B son las dos clases presentes en la tarea de clasificación, si dicha probabilidad estimada es superior a 0.5 para la clase A, el modelo predice esta clase para la muestra en cuestión. Por el contrario, si es inferior al umbral anterior, la clase predecida asociada a la muestra será la clase B.

Para conseguir estimar estas probabilidades, la Regresión Logística actúa de un modo similar a la Regresión Lineal, el algoritmo de regresión más simple existente. Concretamente, la probabilidad estimada por el modelo de Regresión Logística, $\hat{p}$, viene dada por la siguiente ecuación

$$\hat{p} = h_\theta(\mathbf{x}) = \sigma(\theta^T \cdot \mathbf{x}), \tag{2.1}$$

donde

- $\mathbf{x} = [1, x_1, \cdots, x_n]^T$ es el vector de características de la muestra en cuestión, siendo n el número de características de la muestra.
- θ^T es la traspuesta del vector de parámetros del modelo, $\theta = [\theta_0, \cdots, \theta_n]^T$.
- $\sigma(\cdot)$ es la función logística, un tipo de función sigmoidal dada por $\sigma(t) = \frac{1}{1+e^{-t}}$, cuya principal característica es que devuelve un valor entre 0 y 1.
- h_θ es la función hipótesis, que depende de los parámetros del modelo.

La principal diferencia respecto a la Regresión Lineal es que en este caso se pasa el valor de $\theta^T \cdot \mathbf{x}$ a través de la función logística, lo cual permite tener una probabilidad a la salida, en lugar de ser directamente dicho valor el resultado de la predicción.

Al final, desde el punto de vista del entrenamiento, el problema se reduce en aprender los valores del vector de parámetros del modelo que maximicen el rendimiento de la clasificación.

Por su parte, la Regresión Softmax, como se dijo anteriormente, es la extensión al caso multiclase de la Regresión Logística, por lo que está basada también en ideas parecidas. Para una determinada muestra, $\mathbf{x}$, se calcula un cierto valor de score para cada clase k, $s_k(\mathbf{x})$, mediante

$$s_k(\mathbf{x}) = \theta_k^T \cdot \mathbf{x}, \tag{2.2}$$

siendo θ_k^T la traspuesta del vector de parámetros de la clase en cuestión, θ_k. Por tanto, cabe destacar que se tiene un vector de parámetros asociado a cada clase, hecho que no ocurría en la Regresión Logística, donde existía uno único en el modelo.

A partir del cálculo de los diferentes scores para la muestra tratada, se estima la probabilidad de que la misma pertenezca a una determinada clase k, $\hat{p}_k$, haciendo uso de la función softmax, quedando así entonces

$$\hat{p}_k = \sigma(\mathbf{s}(\mathbf{x}))_k = \frac{e^{s_k(\mathbf{x})}}{\sum_{j=1}^{K} e^{s_j(\mathbf{x})}}, \tag{2.3}$$

donde K es el número de clases presentes en la clasificación y $\mathbf{s}(\mathbf{x})$ un vector que contiene todos los scores $s_k(\mathbf{x})$ de la muestra (por lo que tiene K elementos).

La fase de entrenamiento, no consiste más que en aprender la configuración de los vectores de parámetros que maximice la probabilidad estimada para aquella clase a la cual pertenece la muestra en cuestión, y minimice la del resto de clases, ya que a la hora de realizar una predicción el modelo determinará como clase para una nueva muestra aquella para la que su probabilidad obtenida sea más alta.

Implementación en Scikit-Learn

Scikit-Learn implementa la Regresión Logística y, en particular, la Regresión Softmax, mediante la clase **LogisticRegression** [6, 7]. Los parámetros más importantes de esta clase que aquí competen son:

- *multi_class*: su valor por defecto es "auto". Este parámetro permite establecer la forma en la se afronta una tarea de clasificación multiclase. Configurándolo a "ovr" se implementa la estrategia One-vs-Rest (también conocida como One-vs-All) sobre el modelo de Regresión Logística, mientras que si su valor es "multinomial" lo que se realiza es justamente la Regresión Softmax.
- *C*: su valor por defecto es 1.0. Se trata del parámetro que controla la regularización del modelo. Conforme se aumenta su valor, menor es la regularización a la que es sometido, y viceversa.

2.3.2 Support Vector Machines

Una SVM [5, 6, 7] es un algoritmo de Machine Learning de gran potencia y versatilidad que permite dar solución a tareas tanto de clasificación como de regresión. Existen diferentes versiones dependiendo de la transformación que realizan sobre los datos de entrada, la cual queda determinada por lo que se denomina el kernel. Entre los más comunes, se encuentran el lineal, el polinómico, el RBF gaussiano o el sigmoidal. A continuación, se explicará la SVM lineal en su versión orientada a clasificación, ya que es la que se utilizará más adelante en este trabajo.

Antes de entrar detalladamente en los conceptos fundamentales que sustentan las SVMs, cabe destacar que se trata de algoritmos que solo pueden afrontar directamente problemas de clasificación binaria. No obstante, existen diferentes estrategias (One-vs-Rest, One-vs-One) que permiten que puedan ser utilizadas para tareas multiclase.

De forma general, la idea principal de un clasificador SVM lineal es construir un hiperplano que separe las muestras de entrenamiento en dos, en función de la clase a la que pertenezcan, haciendo que el límite de decisión aprendido se encuentre lo más lejos posible de las muestras de entrenamiento más cercanas.

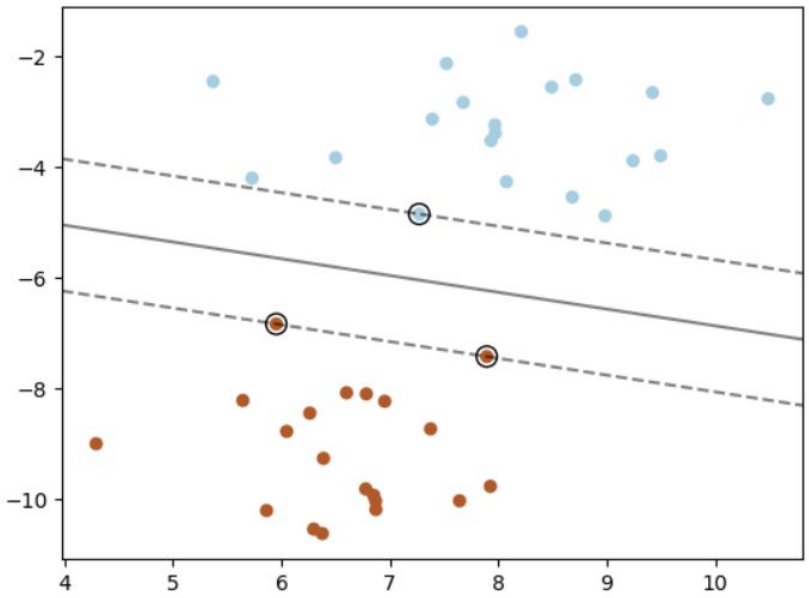

Figura 2.7 Clasificador SVM lineal para un caso bidimensional linealmente separable [6, 7].

La Figura 2.7 permite comprender claramente de forma gráfica la explicación anterior. Se trata de un clasificador SVM lineal donde, en este caso, las muestras presentes en los datos de entrenamiento son linealmente separables. En principio, existen infinitas rectas que separarían las muestras correspondientes a cada clase, obteniendo el clasificador un buen rendimiento en la fase de entrenamiento. Sin embargo, aquí está la particularidad del SVM. Este algoritmo maximiza la distancia entre el límite de decisión (línea continua negra) y las muestras de entrenamiento más cercanas (las que están rodeadas), lo cual consigue que, ante nuevas muestras, se tenga un rendimiento muy superior de lo que se obtendría con una línea recta cualquiera que también separase correctamente dichas muestras de entrenamiento. Es decir, el clasificador SVM tendrá un error de generalización mucho menor.

El espacio existente entre las dos líneas discontinuas es lo que se conoce como margen, el cual está determinado por el conjunto de muestras que están sobre dichas líneas, llamadas vectores de soporte. Estos vectores de soporte son los que condicionan completamente el tamaño de dicho margen que, como se podría pensar de forma intuitiva, interesa que sea lo más amplio posible.

A la hora de construir un clasificador SVM lineal y, en general, cualquiera con otro kernel, existen dos enfoques principales: hard margin y soft margin.

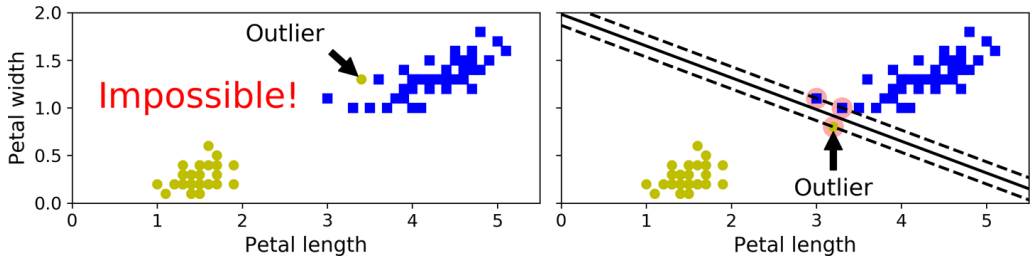

Figura 2.8 Sensibilidad de un clasificador SVM lineal basado en hard margin ante outliers [8].

En la clasificación hard margin, se exige que todas las muestras de entrenamiento se encuentren fuera del margen y clasificadas correctamente, como ocurría precisamente en la Figura 2.7. Esto tiene el principal problema de que si las clases no son separables linealmente, el algoritmo no puede encontrar una solución, hecho que se muestra en la gráfica izquierda de la Figura 2.8. Además, el límite de decisión puede verse

muy afectado si los datos son demasiado ruidosos o contienen outliers, conduciendo a un modelo que no generalizaría bien ante nuevas muestras (gráfica derecha de la Figura 2.8).

Para evitar estos problemas, está la clasificación soft margin, que otorga una mayor flexibilidad. En este caso, se permite que haya muestras de entrenamiento dentro del margen (o incluso clasificadas incorrectamente), y se trata de encontrar un balance adecuado entre que el margen siga siendo tan ancho como sea posible, y la cantidad de muestras que violan las reglas de lo que se proponía en hard margin sea lo menor posible. Este balance es controlado por un hiperparámetro de regularización, el cual debe ser fijado con el objetivo de conseguir el modelo que tenga una mayor capacidad de generalización.

Una vez han sido explicadas las nociones básicas de un clasificador SVM lineal, hay que profundizar en cómo lo anterior se materializa matemáticamente, cómo el algoritmo lleva a cabo una predicción.

A nivel de notación, existen algunas diferencias respecto a lo utilizado en la Subsección 2.3.1. En la Regresión Softmax, el vector de parámetros θ incluía tanto el término de sesgo, θ_0, como el resto de pesos asociados a las características, $\theta_1, \cdots, \theta_n$. Además, se añadía la característica correspondiente al sesgo a todas las muestras, $x_0 = 1$. A la hora de tratar con las SVMs, esto se modifica ligeramente, teniendo por un lado el término del sesgo, denominado b, y por otro el vector de pesos de las características, denotado $\mathbf{w} = [w_1, \cdots, w_n]^T$. Por tanto, no se añadirá ninguna nueva característica a las muestras.

Suponiendo que A y B son las dos clases presentes en la clasificación, para predecir la clase de una nueva muestra $\mathbf{x}$, el clasificador comienza calculando la función de decisión, $\mathbf{w}^T \cdot \mathbf{x} + b$. Si dicho valor es mayor o igual que cero la clase predecida será, por ejemplo, la clase A, mientras que si es inferior a 0 será la clase B. Es decir, el límite de decisión está determinado por el conjunto de puntos donde la función de decisión es 0. Para el caso de un dataset con dos características, la función de decisión es un plano bidimensional y el límite de decisión una recta. En general, para un dataset con n características, la función de decisión será un hiperplano n-dimensional, mientras que el límite de decisión otro de dimensión $n-1$.

Al final, la fase de entrenamiento de un clasificador SVM lineal se reduce a encontrar los valores de $\mathbf{w}$ y b que consigan un margen lo más amplio posible sujeto a las restricciones impuestas por el enfoque elegido, ya sea el hard margin o el soft margin.

Implementación en Scikit-Learn

La clase **LinearSVC** [6, 7] es la implementación en Scikit-Learn de una SVM con kernel lineal para tareas de clasificación. Como se indicó anteriormente, las SVMs son algoritmos de clasificación binaria, pero pueden ser extendidas al caso multiclase mediante el uso de diferentes técnicas. Para la selección de las mismas, la clase **LinearSVC** dispone del parámetro *multi_class*, configurado por defecto a "ovr" (One-vs-Rest), que es como se utilizará en este trabajo.

Además del parámetro anterior, también cuenta con el parámetro *C*, para el control de la regularización del modelo y, cuyo valor por defecto, vuelve a ser 1.0, como ocurría en la clase **LogisticRegression**. A valores mayores de dicho parámetro, se consigue un menor número de muestras incorrectamente clasificadas o dentro del margen, pero dicho margen es más estrecho. Por el contrario, fijar *C* a valores más pequeños permite tener un margen más amplio, evidentemente a cambio de que más muestras se sitúen dentro del mismo o sean clasificadas erróneamente.

2.3.3 Random Forests

Un Random Forest es un conjunto de Árboles de Decisión, normalmente entrenado mediante el método de Bagging, aunque en ocasiones también se emplea Pasting [5]. Estos dos últimos métodos son algoritmos de gran popularidad dentro de lo que se denomina Ensemble Learning, campo de estudio del Machine Learning que trabaja con la idea de que si realizas una pregunta a una gran cantidad de personas al azar y agregas sus respuestas, a menudo dicha respuesta final será mejor que la respuesta dada por un experto en la materia en cuestión. Traduciéndolo concretamente a lo que aquí compete, quiere decir que si tienes un grupo de predictores, por ejemplo clasificadores, y agregas sus predicciones, obtendrás un rendimiento superior que con el mejor clasificador empleado por separado.

Los métodos de Ensemble Learning aumentan sus prestaciones conforme los diferentes predictores que forman el conjunto sean más independientes entre sí. Esto se puede conseguir mediante dos enfoques: uno, que los diferentes predictores provengan de algoritmos de entrenamiento distintos, y otro, como es el caso de los Random Forests, que empleen el mismo algoritmo pero sean entrenados en subconjuntos diferentes del set de entrenamiento.

Precisamente, dicho segundo enfoque es el que implementan Bagging y Pasting. Son métodos que, dado un set de entrenamiento, crean tantos subconjuntos de este set de entrenamiento como predictores vayan a ser

entrenados. Estos subconjuntos de datos se obtienen mediante muestreo aleatorio del set de datos original y ambos métodos permiten que aparezcan muestras iguales en subconjuntos diferentes. La distinción entre Bagging y Pasting se produce en el hecho de que, para un determinado subconjunto de datos, en Bagging pueden aparecer muestras repetidas, mientras que en Pasting no.

A la hora de realizar una predicción, suponiendo que se está trabajando con clasificadores, se obtienen las respuestas de cada uno y se agregan correspondientemente. En función de la forma empleada para llevar a cabo dicha agregación, se distingue principalmente en hard voting y soft voting. El primero propone como respuesta final aquella clase que ha sido más votada por los diferentes clasificadores, mientras que el segundo toma aquella con la probabilidad de clase más alta, siendo estas probabilidades calculadas a partir de la probabilidad obtenida para cada clase por cada clasificador.

El Random Forest es uno de los algoritmos de Machine Learning más versátiles y potentes que existen actualmente, pudiendo ajustarse perfectamente a datos de alta complejidad. Esto, además de las ventajas evidentes que supone para poder ser empleado en múltiples aplicaciones, tiene el riesgo de que presenta gran tendencia al sobreajuste. Con el objetivo de evitarlo, es clave regularizar correctamente el modelo y, para ello, dispone de múltiples hiperparámetros que ajustar. A continuación, se explicarán algunos de los más importantes, mostrando además su correspondencia en la librería Scikit-Learn.

Implementación en Scikit-Learn

Los clasificadores Random Forests se encuentran implementados en la clase **RandomForestClassifier** en Scikit-Learn [5, 6, 7]. Por defecto, esta clase lleva a cabo el método de Bagging (*bootstrap*=True), pero también permite emplear Pasting (*bootstrap*=False).

Respecto a sus parámetros, primeramente, es clave decidir el número de estimadores presentes en el conjunto, es decir, "el número de árboles que conforman el bosque" (*n_estimators*). Al tratarse de un modelo basado en las técnicas de Bagging o Pasting, también cuenta con dos hiperparámetros típicos de las mismas, como son el número de muestras que contiene cada subconjunto de datos (*max_samples*) y el número de características presentes en dichas muestras (*max_features*). No obstante, para este caso de los Random Forests, el parámetro *max_features* no es exactamente el número de características presentes en las muestras, como suele ocurrir en las técnicas de Bagging o Pasting en general, sino que tiene que ver con el número de características evaluadas por el algoritmo de entrenamiento cuando va a separar un nodo. Es decir, las muestras cuentan con todas las características presentes originalmente, y es en cada nodo donde se toma un subconjunto aleatorio de tamaño *max_features* dentro del cual se selecciona aquella que obtiene la mejor separación.

Por último, dispone de los hiperparámetros propios de los Árboles de Decisión, algoritmo base de los clasificadores que conforman el Random Forest, los cuales permiten controlar el desarrollo y forma de los diferentes árboles durante el entrenamiento. Entre los más destacados, se encuentran:

- Profundidad máxima del árbol (*max_depth*).
- Mínimo número de muestras que un nodo debe tener antes de poder ser separado (*min_samples_split*).
- Mínimo número de muestras que debe tener un nodo final (*min_samples_leaf*). Un nodo final no es más que aquel que no tiene ningún nodo hijo.
- Número máximo de nodos finales que puede tener el modelo (*max_leaf_nodes*).

3 Autoencoders

En este capítulo se estudiarán los autoencoders, estructuras centrales dentro del campo del Deep Learning y, concretamente, esenciales en este trabajo. Se realizará un especial énfasis en los de tipo variacional y sus derivados, como es el caso del CVAE y, finalmente, se presentará un ejemplo concreto, el ID-CVAE, el cual será implementado en el Capítulo 4 para resolver una tarea de clasificación de ataques.

3.1 Fundamentos de los autoencoders

Un autoencoder es una arquitectura de red neuronal artificial que intenta aprender una representación eficiente de los datos de entrada de forma automática, sin supervisión. Para aprender dicha representación, denominada representación latente o código, el autoencoder trata de copiar su entrada a su salida, lo cual parece, a priori, una tarea trivial. Sin embargo, la realidad es que no lo es, ya que para evitar la solución trivial, donde simplemente copie la entrada a la salida de manera directa, sin aprender ninguna característica útil sobre los datos de entrada, el modelo es sujeto a una serie de restricciones que lo fuerzan a encontrar dichas características significativas [4, 8, 9].

Existen diferentes formas de restringir a un autoencoder con el objetivo de forzarlo a extraer patrones relevantes de los datos. Entre las más conocidas se encuentran, por ejemplo, el limitar el tamaño de la representación latente, es decir, que dicha representación tenga una dimensión menor que la dimensión de los datos de entrada, o el añadir ruido a las entradas y entrenar el autoencoder para recuperar las entradas originales libres de ruido.

Independientemente del tipo de restricción impuesta, todos los autoencoders presentan una arquitectura común, formada por dos partes fundamentales. Por un lado, se encuentra el encoder o codificador, encargado de transformar la entrada, $\mathbf{x}$, en la representación latente, representada por el vector $\mathbf{z} = f(\mathbf{x})$. Por otro lado, se encuentra el decoder o decodificador, cuyo trabajo consiste en la recuperación de los datos originales a partir de dicha representación latente, $\tilde{\mathbf{x}} = g(\mathbf{z})$ (ver Figura 3.1).

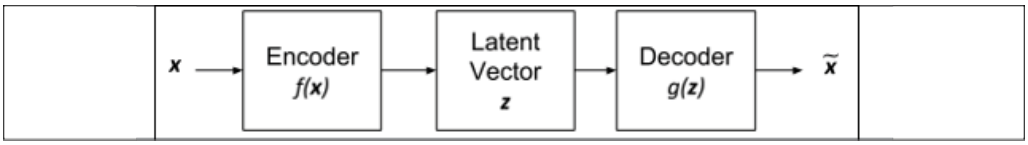

Figura 3.1 Diagrama de bloques de un autoencoder [9].

En general, tanto el codificador como el decodificador son funciones no lineales, y pueden ser implementados mediante redes neuronales simples, como un Perceptrón Multicapa (MLP) o una Red Neuronal Convolucional (CNN), dependiendo de su aplicación.

Al final, el objetivo de un autoencoder es conseguir que $\tilde{\mathbf{x}}$, llamada habitualmente reconstrucción, sea lo más parecida posible a $\mathbf{x}$. Para ello, como ocurre en el resto de arquitecturas de redes neuronales artificiales, ya sean de mayor o menor complejidad, existe una función de pérdida cuyo valor se intenta minimizar durante la fase de entrenamiento. En este caso, dicha función de pérdida, denotada como $L(\mathbf{x},\tilde{\mathbf{x}})$, mide la diferencia entre la entrada original y la recuperada, es decir, entre la entrada y la salida. A una función de pérdida de

este tipo, en la que se evalúa el parecido entre la entrada y la salida, se le conoce como función de pérdida de reconstrucción.

Gracias a un correcto entrenamiento, un autoencoder permite obtener resultados como los que se muestran en la Figura 3.2. Se trata de un autoencoder que trabaja con la base de datos MNIST [10] (base de datos de gran relevancia y utilización dentro del Machine Learning y Deep Learning), la cual contiene imágenes de dígitos escritos a mano de $28 \times 28 \times 1 = 784$ píxeles. La entrada, por tanto, que no es más que una imagen de un determinado dígito, es de dimensión 784. El codificador comprime el espacio de entrada de dimensión 784 en un espacio de dimensión 16, tamaño fijado para la representación latente, y, a partir del mismo, el decodificador intenta recuperar la entrada original, consiguiéndolo con bastante calidad, como se puede observar.

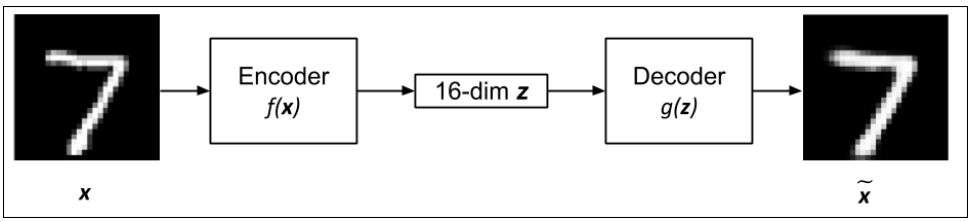

Figura 3.2 Ejemplo de un autoencoder con la base de datos MNIST [9].

3.1.1 Tipos de autoencoders

Como se ha mencionado anteriormente, existen múltiples alternativas para evitar que un autoencoder se comporte de forma trivial, y consiga aprender representaciones eficientes de los datos de entrada [4, 8, 9]. Esto da lugar a diferentes tipos de autoencoders, cada uno con propiedades particulares en su representación latente, las cuales dependen, evidentemente, de la restricción en cuestión impuesta. La más típica, es establecer que la dimensión de la representación latente sea menor que la de los datos de entrada, como ocurría en el ejemplo mostrado de la base de datos MNIST. En tal caso, se habla de un Undercomplete Autoencoder.

Otra forma de forzar a un autoencoder a encontrar características útiles de los datos es añadiendo ruido a sus entradas. Es decir, dado un determinado set de datos de entrenamiento, se añade ruido al mismo (por ejemplo, ruido gaussiano), y dichos datos ruidosos son los que se proporcionan como entradas al modelo. El objetivo del autoencoder es obtener a la salida los datos originales sin ruido, de ahí que a un autoencoder de este tipo se le conozca como DAE (Denoising Autoencoder) [11].

Los Sparse Autoencoders [12] son un ejemplo más de restricción a la que pueden ser sometidos estos modelos. En este caso, la técnica consiste en obligar al autoencoder a reducir el número de neuronas activas en el código para una determinada entrada, añadiendo un término apropiado en la función de pérdida. Es decir, para un cierto set de datos de entrenamiento, el autoencoder es forzado a utilizar, por ejemplo, una media del 5 % de las neuronas presentes en la capa de código, lo cual provoca que tenga que encontrar características de cada muestra de entrada que permitan conseguir un buen rendimiento a la hora de la reconstrucción manteniendo la media de neuronas activas en el valor fijado.

Las técnicas de regularización empleadas tanto en los Denoising como en los Sparse Autoencoders pueden ser aplicadas para trabajar con representaciones latentes cuya dimensión sea igual o superior que la dimensión de los datos de entrada. A los autoencoders que cuentan con una representación latente de estas características se les denomina Overcomplete Autoencoders.

Estos Overcomplete Autoencoders requieren de una atención y cuidado especial en su utilización ya que, conforme mayor es la dimensión del espacio latente en estas estructuras, también aumenta la capacidad de representación con la que cuentan y, por tanto, presentan una mayor tendencia a memorizar la entrada (copiar directamente la entrada a la salida, sin aprender características útiles de los datos). Es clave aplicar las técnicas de regularización anteriormente explicadas de forma precisa para poder tratar con autoencoders de este tipo.

A continuación, se explicará detalladamente un caso más de autoencoder de gran relevancia, que requiere una sección dedicada tanto por su potencial como por la forma en la que trabaja.

3.2 Autoencoders Variacionales

Los Autoencoders Variacionales [3, 8, 9] son uno de los tipos de autoencoders de mayor importancia en la actualidad. Como el resto de autoencoders, cuentan con la estructura básica formada por un codificador y un decodificador, y tienen como principal objetivo reconstruir los datos de entrada consiguiendo aprender la representación latente de los mismos. Sin embargo, conceptualmente son muy diferentes de los explicados hasta ahora.

Para empezar, son autoencoders probabilísticos, donde sus salidas obtenidas presentan una cierta aleatoriedad, incluso cuando ya han completado la fase de entrenamiento. Esto no ocurría para los anteriores, donde sus salidas eran deterministas para la fase de predicción, y solo el DAE introducía una cierta aleatoriedad durante la fase de entrenamiento (al añadir el correspondiente ruido a las muestras originales).

Por otra parte, y esta es su característica más importante, un VAE es un modelo generativo. Un modelo generativo es aquel capaz de generar nuevas muestras que parecen ser parte del set de datos original con el que ha sido entrenado. Para el caso del VAE, esto es conseguido gracias a que su espacio latente es continuo, a diferencia de lo que ocurría en los tratados previamente. Este aspecto permite que, una vez el modelo ha sido entrenado, basta tomar muestras de su espacio latente y pasarlas a través del decodificador para obtener a la salida nuevas muestras que nunca antes habían sido vistas. En este proceso no interviene para nada el codificador, de ahí que a lo que realmente se le denomina modelo generativo en un VAE es al decodificador en sí mismo. El codificador, por su parte, es conocido como modelo de reconocimiento o inferencia, encargado de generar el espacio latente a partir de la entrada.

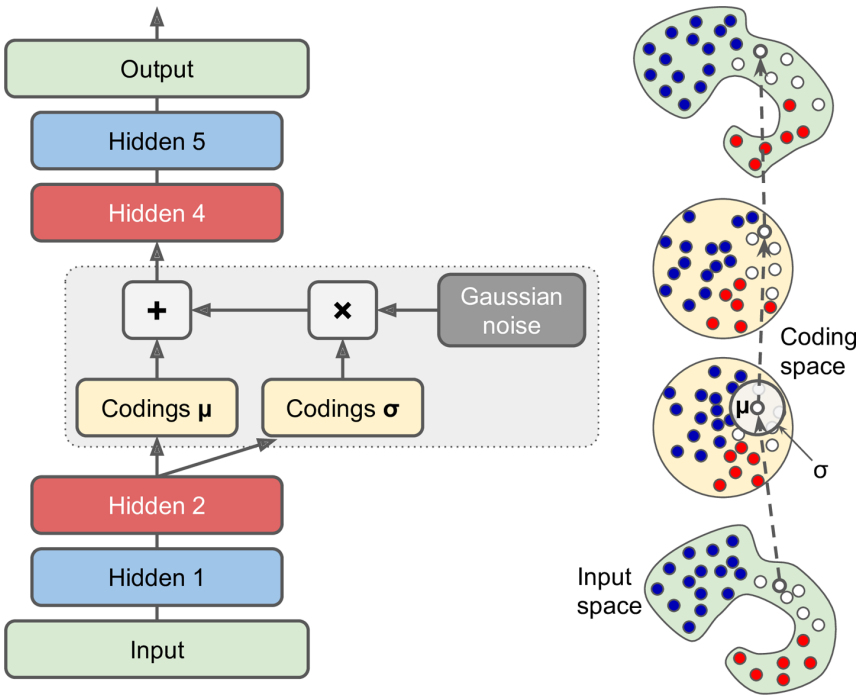

Figura 3.3 Autoencoder variacional [8].

En la parte izquierda de la Figura 3.3 se muestra un ejemplo de un Autoencoder Variacional. Como se ha dicho anteriormente, en él se pueden reconocer tanto el encoder como el decoder pero, en la salida del primero y la entrada del segundo es donde se encuentran las principales modificaciones respecto a otros tipos. En un autoencoder tradicional, el codificador produce un cierto código fijo, $\mathbf{z}$, para una determinada entrada dada. Dicho código es tomado posteriormente por el decodificador, para volver a recuperar la entrada original.

Sin embargo, para el caso de un VAE, el codificador no obtiene dicho código a su salida directamente, sino que produce dos códigos intermedios a partir de los cuales se generará el código final, $\mathbf{z}$, que empleará el decodificador. Básicamente, lo que obtiene el encoder es un código de medias, representado por el vector $\mu(\mathbf{x})$, y otro de desviación típica, representado por el vector $\sigma(\mathbf{x})$. Una vez calculados dichos códigos, $\mathbf{z}$ es obtenido mediante el muestreo aleatorio de una distribución gaussiana multidimensional con media $\mu(\mathbf{x})$ y

desviación típica $\sigma(\mathbf{x})$. El decodificador, entonces, simplemente realiza su tarea de forma normal intentando recuperar la entrada original, $\mathbf{x}$, a partir de este nuevo vector $\mathbf{z}$.

En la parte derecha de la Figura 3.3 puede observarse todo el proceso anterior, desde que se tiene una muestra de entrenamiento a la entrada del codificador hasta que se obtiene su reconstrucción a la salida. Como se puede ver, a partir de la muestra de entrenamiento se obtienen $\mu(\mathbf{x})$ y $\sigma(\mathbf{x})$, y la entrada al decodificador es una muestra aleatoria cercana a dicha media. Finalmente, el decodificador obtiene la salida, muy próxima a la entrada original.

Como se indicó en los primeros párrafos de esta sección, la característica más destacada de un VAE es, indudablemente, su capacidad para generar nuevas muestras similares a aquellas con las que ha sido entrenado, es decir, ser un modelo generativo. Para realizar esta generación, una vez se tiene el VAE ya entrenado, basta alimentar al decodificador con muestras aleatorias tomadas de una distribución $N(0,I)$ y, a su salida, se obtendrán nuevas muestras significativas (cabe recordar que el codificador no interviene para nada en este proceso).

Este hecho tiene una explicación y es que, en general, se busca que la distribución de $\mathbf{z}$ sea lo más parecida posible a una $N(0,I)$. Durante la fase de entrenamiento, por tanto, el decodificador aprende a recuperar las entradas originales teniendo en su propia entrada muestras con características muy similares a aquellas tomadas de una $N(0,I)$ y, cuando durante la fase de inferencia es alimentado directamente con muestras procedentes de dicha distribución, apenas nota diferencias, obteniendo así a su salida nuevas muestras totalmente coherentes con las empleadas en su entrenamiento.

3.2.1 Formulación matemática

En general, los modelos generativos intentan aprender la distribución de probabilidad de los datos de entrada haciendo uso de redes neuronales. En el caso del VAE, esto es conseguido gracias a su estructura compuesta por los dos bloques referidos ya en múltiples ocasiones, el codificador y el decodificador [2, 3, 8, 9].

El codificador, llamado también en este contexto, como se dijo anteriormente, modelo de reconocimiento o inferencia, es modelado por la distribución de probabilidad condicional $Q_\theta(\mathbf{z} \mid \mathbf{x})$ (θ, empleada aquí, y ϕ, empleada posteriormente en el decodificador, representan el conjunto de parámetros de las redes neuronales en las que se basan ambos bloques). Esto indica que, a partir de la entrada, $\mathbf{x}$, es capaz de generar el vector latente, $\mathbf{z}$. Generalmente, se toma como dicha distribución una gaussiana multidimensional, por lo que

$$Q_\theta(\mathbf{z} \mid \mathbf{x}) = N\left(\mathbf{z}; \mu(\mathbf{x}), \sum(\mathbf{x})\right), \tag{3.1}$$

donde $\mu(\mathbf{x})$ es el código de medias y $\sum(\mathbf{x}) = diag\left(\sigma^2(\mathbf{x})\right)$ la matriz de covarianza, calculada a partir del código de desviación típica. Como se ha explicado, ambos códigos son los obtenidos por el encoder a su salida. Cabe destacar que los elementos de $\mathbf{z}$ son independientes, de ahí que la matriz de covarianza sea simplemente la diagonal con las varianzas y cero en el resto.

Por otra parte, el decodificador, que es el modelo generativo, es modelado por la distribución de probabilidad condicional $P_\phi(\mathbf{x} \mid \mathbf{z})$. Es decir, reconstruye la entrada original, $\mathbf{x}$, a partir del vector latente, $\mathbf{z}$.

Como ocurre para cualquier red neuronal, el objetivo durante la fase de entrenamiento es aprender los valores de los pesos (en este caso, θ y ϕ) que minimicen una determinada función de pérdida. Para el VAE, la función de pérdida en cuestión, L_{VAE}, está formada por dos términos, teniéndose que

$$L_{VAE} = L_R + L_{KL} \tag{3.2}$$

El primero, $L_R = -\log P_\phi(\mathbf{x} \mid \mathbf{z})$, es la conocida función de pérdida de reconstrucción. Básicamente, dicho término mide la capacidad del modelo de obtener a su salida la entrada original, es decir, de minimizar las diferencias entre $\tilde{\mathbf{x}}$ y $\mathbf{x}$. Desde otro punto de vista, indica que se quiere maximizar la probabilidad de recuperar la distribución de entrada dada la distribución del vector latente. Entre las funciones más comunes para medir dicha pérdida de reconstrucción se encuentran el MSE u otras basadas en la entropía cruzada.

Por otra parte, el segundo término, L_{KL}, denominado función de pérdida latente o de regularización, es una distancia entre la distribución del codificador, $Q_\theta(\mathbf{z} \mid \mathbf{x})$, y la distribución a priori de $\mathbf{z}$, $P(\mathbf{z})$. Para su cálculo, se emplea la divergencia KL (Kullback-Leibler), que permite medir las diferencias entre dos distribuciones en términos de distancia. En general, la divergencia KL viene dada por

$$D_{KL}(p_{data} \parallel p_g) = \mathbb{E}_{x \sim p_{data}} \log \frac{p_{data}(x)}{p_g(x)} \tag{3.3}$$

Por tanto, el término L_{KL} es simplemente $L_{KL} = D_{KL}(Q_\theta(\mathbf{z} \mid \mathbf{x}) \parallel P(\mathbf{z}))$, tomándose habitualmente la distribución a priori de $\mathbf{z}$ como $P(\mathbf{z}) = N(0,I)$. Intuitivamente, esta pérdida latente fuerza al autoencoder a que la distribución de su espacio latente sea lo más parecida posible a la distribución fijada como referencia, actuando como una especie de regularización contra el posible sobreajuste a los datos de entrenamiento.

3.2.2 Autoencoders Variacionales Condicionales

Dentro de los Autoencoders Variacionales, existen diferentes versiones que permiten tener un mayor control sobre el modelo. La más importante es el Autoencoder Variacional Condicional, un tipo de VAE donde se emplean las etiquetas de clase [2, 9]. En su forma más común, dichas etiquetas son incluidas en formato one-hot tanto en la entrada del encoder como del decoder. No obstante, no tiene por qué ser así, pueden ser incluidas solo en uno de ellos y no necesariamente en la capa de entrada.

Para el ejemplo de la base de datos MNIST, el introducir la etiqueta de un cierto número permite controlar qué número va a ser generado a partir de un determinado muestreo de la representación latente. Esto no era posible con el VAE, donde si se realizase un muestreo aleatorio de dicha representación sería imposible controlar qué dígito va a generar. Además, gracias al uso de las etiquetas, normalmente el dígito generado tiene un aspecto más realista de lo que ocurría en el caso del VAE.

Desde el punto de vista matemático, la diferencia entre el VAE y el CVAE se traduce en una nueva condicionalidad de las diferentes distribuciones de probabilidad presentes en el modelo respecto a $\mathbf{c}$, el vector que representa las etiquetas en formato one-hot.

El objetivo ahora es aprender la distribución de probabilidad de los datos de entrada condicionados, para lo cual el codificador es modelado por la distribución de probabilidad condicional $Q_\theta(\mathbf{z} \mid \mathbf{x}, \mathbf{c})$ y el decodificador por $P_\phi(\mathbf{x} \mid \mathbf{z}, \mathbf{c})$, en el caso más general.

Sin embargo, como se dijo anteriormente, dicha condicionalidad puede establecerse únicamente en el codificador o en el decodificador, no necesariamente en ambos. En el primer caso, lógicamente, el codificador seguiría manteniendo la misma distribución de probabilidad condicional, $Q_\theta(\mathbf{z} \mid \mathbf{x}, \mathbf{c})$, mientras que el decodificador en cuestión perdería dicha condicionalidad, quedando su distribución asociada como $P_\phi(\mathbf{x} \mid \mathbf{z})$.

Por otra parte, si la condicionalidad fuese impuesta solamente en el decodificador, las correspondientes distribuciones de probabilidad asociadas quedarían como $Q_\theta(\mathbf{z} \mid \mathbf{x})$, para el caso del codificador, y $P_\phi(\mathbf{x} \mid \mathbf{z}, \mathbf{c})$, para el caso del decodificador.

Para finalizar, es importante indicar que para el CVAE, en su comparativa respecto al VAE, es necesario añadir, de igual manera, dicha condicionalidad en los diferentes términos de la función de pérdida, en la que cabe destacar que se sigue empleando, generalmente, una distribución $N(0,I)$ para, en este caso, la distribución a priori de $\mathbf{z}$ dado $\mathbf{c}$, $P(\mathbf{z} \mid \mathbf{c})$.

3.2.3 ID-CVAE

Una vez conocidas las bases tanto de los autoencoders en general, como de los VAEs y los CVAEs, se procede a presentar un ejemplo concreto, el propuesto en [2], el cual es conocido como ID-CVAE. Se trata de un método de detección de intrusiones en la red, aplicado concretamente a redes IoT, que puede llevar a cabo tanto tareas de clasificación de ataques como de recuperación de características.

Esencialmente, es un método basado en un Autoencoder Variacional Condicional, donde las etiquetas de las diferentes intrusiones son incluidas exclusivamente en una de las capas del decodificador, a diferencia del modelo de CVAE más típico donde se incluían en ambos bloques. Desde el punto de vista matemático, por tanto, la distribución de probabilidad asociada al encoder es simplemente $Q_\theta(\mathbf{z} \mid \mathbf{x})$ (no depende de $\mathbf{c}$, el vector de etiquetas), mientras que la del decoder es $P_\phi(\mathbf{x} \mid \mathbf{z}, \mathbf{c})$.

En la Figura 3.4 se muestra una comparativa entre las arquitecturas del VAE y del ID-CVAE donde, como se acaba de explicar, la distinción principal entre las mismas no es más que la inclusión del vector de etiquetas en formato one-hot (llamado en esta figura $\mathbf{L}$, en lugar de $\mathbf{c}$) en el decodificador.

El ID-CVAE, en particular, y los CVAEs, en general, son métodos de Machine Learning no supervisados (como el resto de autoencoders), pero entrenados de una manera supervisada, debido al uso de estas etiquetas de clase durante el entrenamiento. Este cambio permite que el ID-CVAE cuente con múltiples ventajas respecto al VAE, tanto en términos de flexibilidad como de rendimiento y complejidad de la solución. Por ejemplo, como se verá en el Capítulo 4, puede abordar una tarea de clasificación multiclase en un solo paso de entrenamiento, hecho que no es posible con un VAE.

Además de las ventajas que ostenta frente a modelos similares, también ofrece mejores resultados de clasificación que clasificadores tradicionales del Machine Learning, como los Random Forests, las SVMs, la Regresión Softmax y los MLPs.

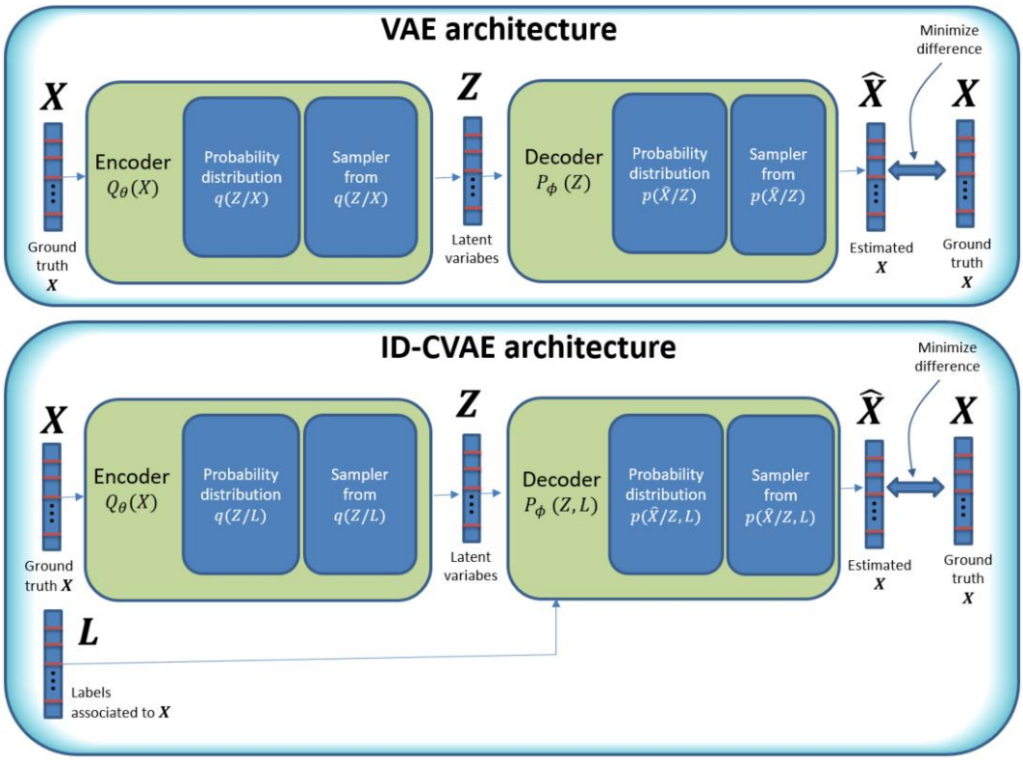

Figura 3.4 Comparación del ID-CVAE con una arquitectura de VAE típica [2].

En su momento, el ID-CVAE fue un método único en el campo NIDS, suponiendo la primera aplicación de un CVAE en el mismo y proporcionando el primer algoritmo que permitía recuperar características de sets de datos incompletos. En el próximo capítulo, se explicará detalladamente su arquitectura, así como el proceso necesario para poder utilizarlo en una tarea de clasificación.

4 Descripción y resultados del trabajo práctico

A lo largo de este capítulo, se llevará a cabo la exposición del trabajo práctico desarrollado, desde la base de datos seleccionada hasta las diferentes métricas de rendimiento que permiten analizar la calidad de la clasificación. Para la realización del mismo, se ha empleado el entorno Anaconda 3-2020.02 [13], destacando el uso de la versión 2.0.0 de TensorFlow [14] y la versión 0.22.2.post1 de Scikit-Learn [6, 7]. Los correspondientes códigos implementados, mediante los cuales pueden obtenerse los resultados que se mostrarán posteriormente, pueden encontrarse en el Apéndice A.

4.1 Introducción

El objetivo del trabajo práctico realizado es replicar la tarea de clasificación mostrada en [2], una tarea de clasificación de ataques multiclase aplicada a la detección de intrusiones en una red IoT. Concretamente, esta tarea es abordada mediante el método ID-CVAE introducido en el capítulo anterior, el cual ha sido diseñado expresamente para la misma. Además, se emplearán otra serie de algoritmos de Machine Learning familiares para su resolución, con el fin de poder establecer una comparativa con el rendimiento ofrecido por dicho ID-CVAE y así extraer las conclusiones oportunas.

4.2 Set de datos seleccionado

El set de datos empleado es el NSL-KDD, un dataset representativo para la detección de intrusiones basada en anomalía (puede encontrarse en [15]). Se trata de una versión mejorada del dataset KDD 99 (siendo este el más utilizado hasta el momento en dicho campo), en la que se resuelven algunos de los problemas presentes en el mismo, como el número de muestras redundantes, haciéndolo más útil y realista [2, 15, 16, 17].

Al ser un dataset de gran popularidad dentro del campo NIDS y contar con propiedades como un número de muestras razonable tanto en los sets de entrenamiento como de test, permite realizar experimentos empleando todas las muestras disponibles y comparar los resultados de manera consistente con trabajos desarrollados por otros autores. Concretamente, cuenta con un total de 125.973 muestras de entrenamiento y 22.544 muestras de test, cada una con 41 características, de las cuales 38 son continuas y 3 categóricas.

En cuanto a los tipos de intrusiones, que no son más que las etiquetas de dichos sets de datos, el dataset de entrenamiento cuenta con un total de 23 posibles (la clase normal, que indica la inexistencia de ataque, más 22 asociadas con diferentes tipos de intrusiones), mientras que el dataset de test aumenta su número hasta contar con 38 posibilidades. Es decir, existen muestras de test que contienen ataques no presentes a la hora del entrenamiento, lo cual proporciona una gran variabilidad entre ambos sets de datos y supondrá un enorme desafío en la predicción a los diferentes algoritmos empleados.

Además de lo anterior, la distribución entre las diferentes clases está fuertemente desbalanceada, característica típica de las bases de datos utilizadas en este campo, ya que, evidentemente, los ataques en una red son anomalías dentro del tráfico normal de la misma, mucho más abundante, y no todos los tipos de ataques son igual de habituales.

4.2.1 Tipos de intrusiones

De las 23 clases del set de entrenamiento y las 38 del set de test, hay un total de 21 clases comunes (letra estándar en la Tabla 4.1), 2 que solo se encuentran en el set de entrenamiento (letra cursiva en la Tabla 4.1) y 17 que solo aparecen en el set de test (letra negrita en la Tabla 4.1) [2, 16, 17].

Tabla 4.1 Categorización de ataques de los datasets de entrenamiento y test.

DoS	Probe	R2L	U2R
apache2	ipsweep	ftp_write	buffer_overflow
back	**mscan**	guess_passwd	loadmodule
land	nmap	**httptunnel**	perl
mailbomb	portsweep	imap	**ps**
neptune	**saint**	multihop	rootkit
pod	satan	**named**	**snmpguess**
processtable		phf	**sqlattack**
smurf		**sendmail**	**worm**
teardrop		**snmpgetattack**	**xterm**
udpstorm		*spy*	
		warezclient	
		warezmaster	
		xlock	
		xsnoop	

Como se puede observar en la tabla anterior, las etiquetas originales correspondientes a las distintas intrusiones pueden agruparse en 4 grandes categorías: DoS, Probe, R2L y U2R, en función de las características del ataque en cuestión. Realizando esta categorización, se obtienen unos sets de datos de entrenamiento y test con 5 clases diferentes, entre las que se encuentran la Normal, que indica la ausencia de intrusión, y las 4 correspondientes a estos tipos de ataques. Es justamente con dicha clasificación con la que trabajarán los métodos desarrollados y, por tanto, la que se empleará a la hora de mostrar los resultados.

De forma general, los 4 posibles tipos de ataques presentes consisten en:

- Denial of Service (DoS): Se trata de un ataque en el que el ejecutor del mismo bloquea o niega el acceso a un recurso o servicio, ya sea de una máquina o una red, a usuarios legítimos, haciendo parecer que están demasiado ocupados para atender la petición requerida por el usuario en cuestión.
- Probe: Este ataque se encarga de recoger información sobre las vulnerabilidades potenciales del sistema objetivo, con la intención de que esta información pueda ser utilizada para llevar a cabo ataques posteriores en dicho sistema.
- Remote to Local (R2L): Se produce cuando un atacante capaz de enviar paquetes a través de la red a una máquina de la que no es usuario explota alguna vulnerabilidad de la misma para conseguir acceso como si de un usuario normal se tratara.
- User to Root (U2R): En este tipo de ataque, tras ganar acceso al sistema como un usuario más, es decir, como si se hubiese realizado previamente un ataque R2L, el atacante consigue privilegios de administrador explotando ciertas debilidades encontradas.

La agrupación de las clases originales en las nuevas que serán empleadas de la forma mostrada en la Tabla 4.1 permite, finalmente, obtener unos datasets de entrenamiento y test con la distribución de las muestras indicada en la Tabla 4.2, donde, tal y como era esperado, la clase Normal es claramente la clase predominante.

4.2.2 Características de las muestras

Respecto a las características de las muestras, como se dijo anteriormente, cada muestra cuenta con 41 características, de las cuales 38 son continuas y 3 categóricas. En función de sus propiedades, dichas características pueden ser clasificadas en uno de los siguientes grupos: básicas, de tráfico y de contenido [2, 16, 17].

Tabla 4.2 Número de muestras por clases en los datasets de entrenamiento y test.

Clases	Muestras entrenamiento	Muestras test
Normal	67343	9711
DoS	45927	7458
Probe	11656	2421
R2L	995	2554
U2R	52	400
Total	**125973**	**22544**

Las características básicas son aquellas que contienen información procedente de la conexión en sí, tales como el protocolo de transporte empleado o el servicio que va sobre el mismo, además de otros atributos que pueden ser extraídos de dicha conexión.

Las características de tráfico, por su parte, son características calculadas respecto a una ventana o intervalo de tiempo y, a su vez, pueden dividirse en 2 subgrupos: "same host" y "same service". Las primeras, analizan las conexiones producidas en los últimos 2 segundos que tienen el mismo destino que la conexión en cuestión a examinar, mientras que las segundas analizan aquellas producidas en ese mismo intervalo de tiempo pero que utilizan el mismo servicio que la conexión actual, en vez del mismo destino.

Ambos subgrupos, como se acaba de explicar, examinan el tráfico dentro de una determinada ventana de tiempo, de ahí que realmente se las conozca como características de tráfico basadas en tiempo. A pesar de la información tan valiosa que proporcionan este tipo de características, existen determinados ataques, especialmente de la categoría Probe, cuyos patrones de comportamiento no pueden detectarse en 2 segundos, sino que emplean, por ejemplo, un tiempo de 1 minuto entre un intento de intrusión y el siguiente. Para poder controlarlos, están las características de tráfico basadas en conexión donde, en lugar de analizar los 2 segundos previos a la conexión actual (ventana de tiempo), se tienen en cuenta las 100 últimas conexiones producidas (ventana de conexión).

Las características de tráfico, en general, son verdaderamente útiles para detectar ataques de tipo DoS y Probe, ya que estos involucran una gran cantidad de conexiones al mismo equipo en un período corto de tiempo, teniendo así patrones de intrusión fuertemente repetitivos. Sin embargo, para casos como el R2L y el U2R, esto no es así. Debido a la propia naturaleza de estas intrusiones, donde el ataque se ejecuta introduciéndolo en diferentes campos de las tramas de datos, a menudo solo necesitan realizar una única conexión para conseguir sus frutos. Por tanto, estos ataques escapan de ser detectados por las características de tráfico, haciendo necesario tener características que observen comportamientos sospechosos en dichas tramas (por ejemplo el número de intentos fallidos de login), las denominadas características de contenido.

4.3 Preprocesado de los datos

El preprocesado de los datos es realizado mediante la función *prepare_data*, que puede encontrarse en el Código A.2 del Apéndice A. Básicamente, hay que distinguir dos partes en este preprocesado, la correspondiente a las características y la correspondiente a las etiquetas [2].

Como se ha indicado en la sección previa, cada muestra está formada por un total de 41 características (38 continuas y 3 categóricas). De las características continuas, 6 son descartadas, ya que contienen principalmente ceros y, en aquellas muestras en las que dichas características toman valores no nulos, estos no aportan ninguna información relevante de cara a la tarea de clasificación.

Una vez hecho el descarte anterior, las 32 características continuas restantes son escaladas al rango $[0, 1]$ y, además, se aplica una codificación one-hot a todas las características categóricas. Las 3 características categóricas, *protocol*, *flag* y *service* tienen, respectivamente, 3, 11 y 70 valores distintos, dando lugar así a que, tras el preprocesado realizado, cada muestra de entrenamiento y test cuente con 116 características (32 continuas y 84 binarias, siendo estas últimas las asociadas a las características categóricas indicadas).

En cuanto a las clases, el preprocesado de las mismas consiste, simplemente, en implementar lo ya mostrado en la Tabla 4.1. Se trata de transformar las etiquetas originales en una de las 5 posibles categorías finales: DoS, Normal, Probe, R2L y U2R.

4.4 ID-CVAE

La arquitectura detallada del ID-CVAE, método presentado de forma genérica en el capítulo anterior, es la mostrada en la Figura 4.1.

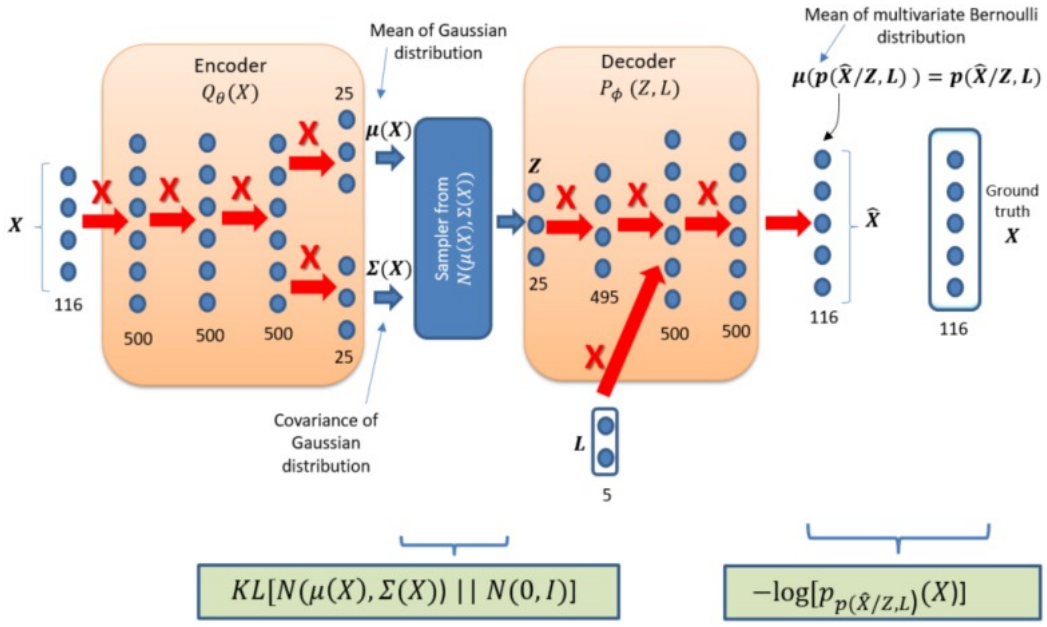

Figura 4.1 Detalles del modelo ID-CVAE [2].

Al igual que se explicó en dicho capítulo para los VAEs y los CVAEs teóricos, este método también utiliza una distribución gaussiana multidimensional como la distribución para $Q_\theta(\mathbf{z} \mid \mathbf{x})$ (con media $\mu(\mathbf{x})$ y matriz de covarianza $\sum(\mathbf{x}) = diag\left(\sigma^2(\mathbf{x})\right)$ y una distribución $N(0,I)$ como la distribución a priori de $\mathbf{z}$, $P(\mathbf{z})$ [2].

En cuanto a la función de pérdida seleccionada, denominada $L_{ID-CVAE}$ en este modelo, que, como se conoce, está compuesta por dos términos ($L_{ID-CVAE} = L_R + L_{KL}$), las elecciones de los mismos también son coherentes con lo estudiado anteriormente [2, 9].

Concretamente, el término L_R es la entropía cruzada binaria entre la entrada, $\mathbf{x}$, y la salida, $\tilde{\mathbf{x}}$, gracias al empleo de una distribución de Bernoulli como distribución para $P_\phi(\mathbf{x} \mid \mathbf{z}, \mathbf{c})$ (donde $\mathbf{c}$ es $\mathbf{L}$ en la Figura 4.1).

Por otra parte, el término L_{KL}, debido también a las características de las distribuciones tomadas para, en este caso, $Q_\theta(\mathbf{z} \mid \mathbf{x})$ y $P(\mathbf{z})$, queda simplificado como

$$L_{KL} = D_{KL}\left(N(\mu(\mathbf{x}), \sum(\mathbf{x})) \parallel N(0,I)\right) = -\frac{1}{2}\sum_{j=1}^{J}\left(1 + \log(\sigma_j)^2 - (\mu_j)^2 - (\sigma_j)^2\right), \tag{4.1}$$

donde J es la dimensión del vector latente $\mathbf{z}$.

Ya desde el punto de vista de implementación, el que compete principalmente a esta sección, cabe destacar varios aspectos relevantes. El primero, clave en un modelo de este tipo, es el decidir cómo y dónde será incluido el vector de etiquetas en el decodificador. En este caso, lo que se realiza es una concatenación de dicho vector de etiquetas en formato one-hot con los valores de la segunda capa del decodificador (la elección de la capa, como indican los propios autores en [2], se ha determinado experimentalmente).

Otra cuestión importante es la configuración de las redes neuronales que forman el codificador y el decodificador. En la Figura 4.1 se muestran tanto las capas como el número de neuronas presentes en cada una que tienen ambos bloques, siendo además dichas capas completamente conectadas (hecho indicado por las flechas rojas representadas en la figura). En todas ellas, se utiliza como función de activación la función ReLU, excepto en la última capa del codificador (función Lineal) y en la última del decodificador (función Sigmoidal).

En el Código A.3 se realiza la implementación del ID-CVAE, así como el entrenamiento del mismo. Dicho modelo entrenado es guardado, lo cual permite su uso posterior sin necesidad de un nuevo entrenamiento. Desde un punto de vista funcional, es una práctica recomendable cuando se trabaja con modelos complejos

de este tipo separar la fase de entrenamiento y la de predicción en dos ficheros diferentes, guardar el modelo una vez entrenado en el primero de ellos y, posteriormente, simplemente cargarlo en el segundo. La mayor parte del tiempo empleado en ejecutar un cierto algoritmo desde cero la ocupa la fase de entrenamiento, por lo que esta metodología proporcionará grandes beneficios.

4.4.1 Clasificación

Una de las posibles aplicaciones del ID-CVAE es dar solución a problemas de clasificación así que, tras haber visto ya su estructura y ciertos detalles de su implementación, se está en disposición de explicar cómo puede ser empleado para llevar a cabo la tarea de clasificación de ataques propuesta en el trabajo [2]. La utilización del ID-CVAE como clasificador no es directa, sino que es necesario realizar un determinado proceso, el cual se ilustra en la Figura 4.2.

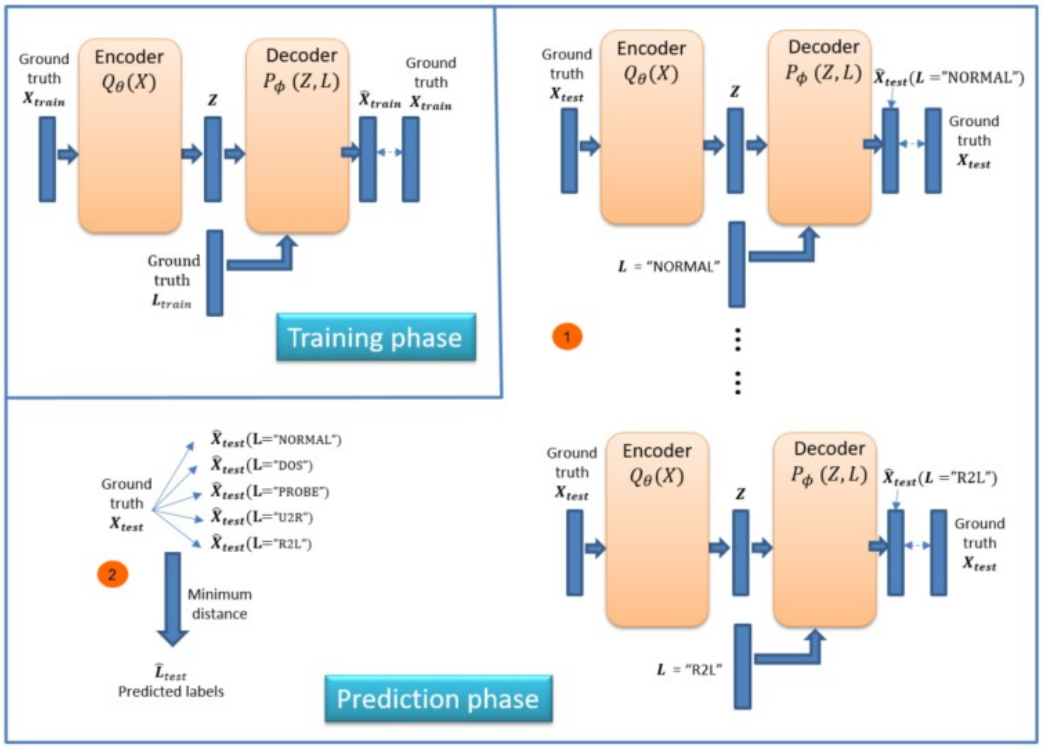

Figura 4.2 Proceso de clasificación en el ID-CVAE [2].

Como se puede observar, este proceso cuenta con dos fases, la de entrenamiento y la de predicción. La fase de entrenamiento, implementada en el Código A.3 al que se acaba de hacer referencia en la sección previa, consiste, como en la mayoría de modelos de Deep Learning, en aprender los valores de los pesos del modelo que minimicen la función de pérdida en cuestión, consiguiendo así que las muestras de entrenamiento recuperadas, $\tilde{\mathbf{x}}_{\mathbf{train}}$, sean lo más parecidas posibles a las originales, $\mathbf{x}_{\mathbf{train}}$.

En la fase de predicción, por su parte, el objetivo es clasificar una determinada muestra de test, $\mathbf{x}_{\mathbf{test}}$, en una de las cinco categorías presentes en los datasets procesados. Dentro de esta fase, se distinguen a su vez dos pasos.

En el primero, se llevan a cabo tantas predicciones, $\tilde{\mathbf{x}}_{\mathbf{test}}(\mathbf{L} = i)$ (donde i corresponde a DoS, Normal, Probe, R2L o U2R), para una determinada muestra de test, $\mathbf{x}_{\mathbf{test}}$, como clases hay en la tarea de clasificación. Para ello, la clave es el uso de las dos entradas presentes en el ID-CVAE, es decir, las características de la muestra de test en cuestión y el vector de etiquetas en formato one-hot. En cada una de las predicciones a realizar para dicha muestra, que en el caso de este trabajo serán un total de 5, se alimenta al modelo con un vector de etiquetas diferente (el correspondiente a la clase $\mathbf{L} = i$), teniendo así una predicción asociada a cada uno de los valores de etiqueta.

El segundo paso de esta fase de predicción consiste, simplemente, en calcular la distancia entre la muestra original, $\mathbf{x}_{\mathbf{test}}$, y cada una de las reconstrucciones obtenidas en el paso anterior, $\tilde{\mathbf{x}}_{\mathbf{test}}(\mathbf{L} = i)$, para elegir finalmente como clase predecida aquella cuya reconstrucción tenga un mayor parecido con la original, es

decir, aquella clase para la que su salida se encuentre a una menor distancia de la entrada. En este modelo, la medida de distancia utilizada es la distancia euclídea.

De acuerdo a todo este proceso explicado, por tanto, puede concluirse que el ID-CVAE requiere una única etapa de entrenamiento, y tantas etapas de test como valores distintos de etiquetas existan en la tarea de clasificación. Como es sabido, la fase de entrenamiento es la que demanda más tiempo y recursos, mientras que la de test es de gran rapidez. Este es uno de los principales motivos de que dicho método ofrezca un gran salto de rendimiento en la comparativa respecto a un VAE tradicional, ya que en el caso del VAE es necesario también realizar tantas etapas de entrenamiento como número de clases existan.

Básicamente, el ID-CVAE necesita crear un único modelo, el cual entrena utilizando todas las muestras de entrenamiento disponibles, mientras que el VAE necesita crear tantos modelos como clases existan, siendo cada uno de ellos entrenado únicamente con las muestras de entrenamiento asociadas a una determinada clase.

La implementación de la fase de predicción del ID-CVAE se encuentra en el Código A.4 donde, además, se calculan los diferentes resultados y métricas de rendimiento que serán mostrados a continuación.

4.4.2 Resultados

Como se acaba de indicar, se procede ahora a presentar los resultados de clasificación conseguidos al aplicar el método ID-CVAE en cuestión a la base de datos seleccionada, la NSL-KDD. Para ello, con el objetivo de realizar un análisis detallado, se proporcionan las siguientes métricas de rendimiento, tanto en forma agregada como por clase: Accuracy, F1, Precision y Recall. Sus correspondientes definiciones, para un caso de clasificación binaria donde las dos clases presentes son denominadas clase positiva y clase negativa, vienen dadas por las siguientes expresiones [18]:

$$\text{Accuracy} = \frac{t_p + t_n}{t_p + t_n + f_p + f_n}, \tag{4.2}$$

$$\text{Precision} = \frac{t_p}{t_p + f_p}, \tag{4.3}$$

$$\text{Recall} = \frac{t_p}{t_p + f_n}, \tag{4.4}$$

$$\text{F1} = \frac{2 \cdot \text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} = \frac{2t_p}{2t_p + f_p + f_n}, \tag{4.5}$$

donde

- t_p (true positive) es el número de muestras de la clase positiva correctamente clasificadas.
- t_n (true negative) es el número de muestras de la clase negativa correctamente clasificadas.
- f_p (false positive) es el número de muestras de la clase negativa incorrectamente clasificadas, es decir, clasificadas como clase positiva.
- f_n (false negative) es el número de muestras de la clase positiva incorrectamente clasificadas, es decir, clasificadas como clase negativa.

Su extensión al caso de clasificación multiclase puede realizarse de manera sencilla, como se explicará seguidamente, dando lugar a las dos posibles formas anteriormente mencionadas de presentar dichas métricas para el caso en cuestión, agregadas o por clase.

El cálculo de su versión por clase se fundamenta en la conocida estrategia One-vs-Rest. Básicamente, se trata de simplificar la tarea multiclase a tantas tareas binarias como número de clases existan, considerando en cada una de ellas una determinada clase frente al resto. Para la base de datos empleada en este trabajo, por tanto, se plantean 5 tareas de clasificación binaria, entre las que se encuentran DoS-vs-Rest, Normal-vs-Rest, etc. Por ejemplo, dentro de la tarea Normal-vs-Rest, una muestra de la clase DoS clasificada incorrectamente como R2L, realmente es considerada un "true negative" ya que, aunque en esencia esta predicción sea errónea, desde el punto de vista de la tarea de clasificación binaria considerada ambas clases (DoS y R2L) pertenecen al grupo Rest, donde las diferentes clases originales que lo conforman son tratadas como una sola.

Una vez calculadas las diferentes métricas por clase de acuerdo a la estrategia anterior, el caso agregado consiste simplemente en calcular una métrica única que resuma los resultados por clase obtenidos. Para realizar dicha agregación, existen diferentes alternativas [6, 7], en función del proceso utilizado para el

promediado, entre las que se encuentran: “micro”, “macro”, “samples” y “weighted”, siendo esta última la empleada en este trabajo.

Cabe destacar que, de entre todas las métricas que se acaban de explicar, debido a la distribución de clases tan fuertemente desbalanceada con la que cuenta la base de datos empleada en este trabajo, la F1 puede ser considerada el mejor indicativo de evaluación de la calidad del modelo.

Antes de comenzar a mostrar los resultados conseguidos, es necesario realizar una consideración de cara a la obtención de los mismos. Durante la fase de entrenamiento del ID-CVAE, se emplea el 90 % de las muestras del dataset de entrenamiento correspondiente para el entrenamiento en sí del modelo, y el 10 % restante para su validación. Sin embargo, en [2], artículo de referencia con el cual serán comparados los resultados obtenidos en este TFM, los autores indican que han utilizado todas las muestras de entrenamiento disponibles para el propio entrenamiento, por lo que se desconoce cómo han validado dicho modelo.

Respecto a la fase de predicción, sí que se emplean el 100 % de las muestras del dataset de test en ambos trabajos.

En la Figura 4.3 y la Tabla 4.3 se muestra la comparativa entre los resultados de clasificación conseguidos en este TFM y los obtenidos en el artículo de referencia en cuestión, [2].

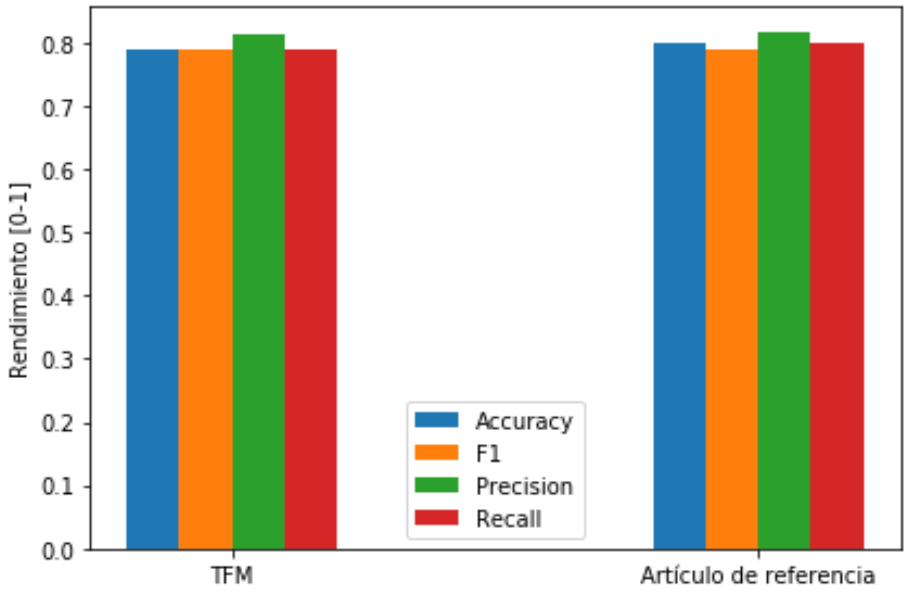

Figura 4.3 Comparativa de rendimiento entre el ID-CVAE del TFM y el del artículo de referencia.

Tabla 4.3 Comparativa de rendimiento entre el ID-CVAE del TFM y el del artículo de referencia.

	TFM	**Artículo de referencia**
Accuracy	0.7909	0.8010
F1	0.7902	0.7908
Precision	0.8133	0.8159
Recall	0.7909	0.8010

Como se puede observar, los resultados son extremadamente parecidos, hecho incluso sorprendente, ya que no es habitual que esto ocurra en Machine Learning y Deep Learning. En general, cuando se intentan reproducir resultados de otros autores empleando, teóricamente, el mismo modelo, suele haber ciertas diferencias (en múltiples ocasiones bastante notables), siendo realmente extraño el conseguir una similitud como la dada aquí.

Para empezar, a la hora de replicar los resultados de un determinado artículo, el modelo desarrollado no suele ser exactamente el mismo que el propuesto en dicho artículo ya que, normalmente, no se indica cómo han sido configurados los diferentes parámetros de los que dispone el modelo en cuestión.

Por otra parte, otra fuente de error relevante es la propia programación e implementación del modelo, en cuanto a librerías y herramientas utilizadas, e incluso a la versión empleada de una misma librería o herramienta, como quedará reflejado en la sección posterior.

Cualquier mínimo cambio respecto al modelo original hace que los resultados logrados puedan ser bastante diferentes, de ahí que resulte tan extraña la similitud de las soluciones mostradas aquí. En este caso, aunque no se indica en [2], seguramente las versiones de las librerías utilizadas y, en general, la forma de programar el modelo en sí misma, presente múltiples diferencias con lo realizado en este TFM, debido a la constante evolución de las herramientas usadas en el campo desde la fecha de publicación del artículo (Septiembre de 2017). A esto hay que añadirle el posible error fruto del desconocimiento del valor de múltiples parámetros del modelo no proporcionados por los autores, resultando así el enorme parecido entre las soluciones en cierta parte una casualidad.

Una vez dadas las explicaciones anteriores, se procede a profundizar en los resultados de clasificación del ID-CVAE desarrollado en este TFM. Para ello, en la Figura 4.4 y la Tabla 4.4 se muestran las métricas de rendimiento por clase obtenidas, las cuales, apoyadas por la matriz de confusión mostrada en la Figura 4.5 (y su correspondiente versión normalizada, en la Figura 4.6), permiten tener un conocimiento mucho mayor de cómo se está comportando este modelo.

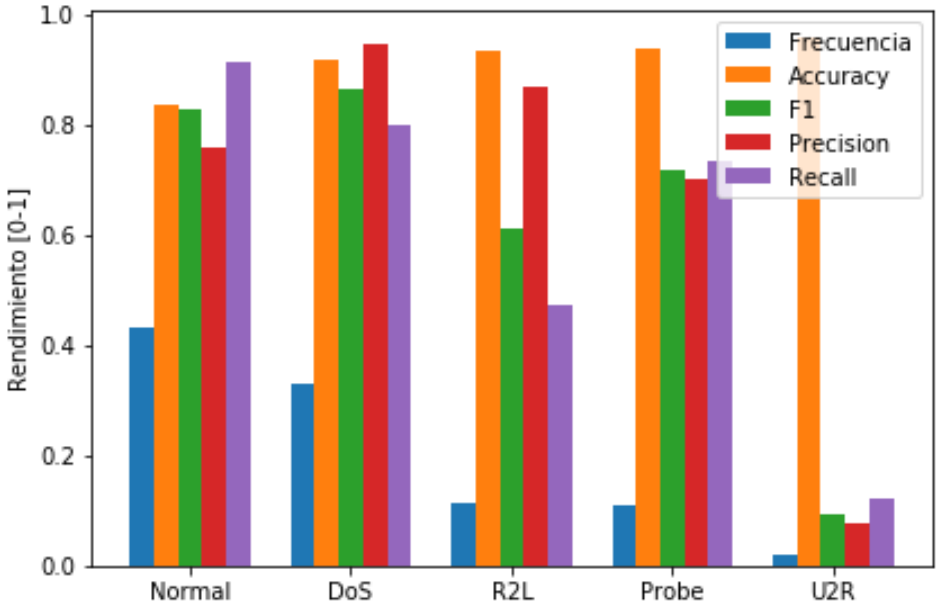

Figura 4.4 Métricas de rendimiento por clase del ID-CVAE.

Tabla 4.4 Métricas de rendimiento por clase del ID-CVAE.

	Normal	**DoS**	**R2L**	**Probe**	**U2R**
Frecuencia	0.4308	0.3308	0.1133	0.1074	0.0177
Accuracy	0.8361	0.9173	0.9319	0.9378	0.9587
F1	0.8274	0.8645	0.6103	0.7167	0.0936
Precision	0.7571	0.9438	0.8679	0.7010	0.0767
Recall	0.9121	0.7975	0.4706	0.7332	0.1200

Básicamente, como cabía esperar, los resultados dependen en gran medida de la cantidad de muestras de cada clase presentes en los datasets de entrenamiento y test (Frecuencia, en la Figura 4.4 y la Tabla 4.4, indica el tanto por uno de muestras de cada clase presentes en el dataset de test). Para las dos clases con un mayor número de muestras, Normal y DoS, se obtiene una F1 superior a la del caso agregado, siendo dicha segunda clase la que consigue un mejor valor de F1 de entre todas las existentes.

Por otra parte, para R2L y Probe, las dos clases que cuentan con una cantidad de muestras inmediatamente inferior a las anteriores, dicha F1 ya sufre una importante disminución, aún sin llegar, ni mucho menos, al

nivel de los resultados obtenidos para la clase U2R, la cual el método no es capaz de tratar correctamente debido a la presencia ínfima de dicha clase en ambos datasets, especialmente en el de entrenamiento.

Un aspecto a tener en cuenta, que se produce tanto en la clase R2L como en la U2R, es el hecho de que la cantidad de muestras de estas clases presentes en el dataset de test es superior al número de muestras de las mismas disponibles en el dataset de entrenamiento. Este podría ser el motivo que explicase la notable diferencia a favor en el resultado de la F1 de la clase Probe respecto a la R2L ya que, aunque sus números sean similares en el dataset de test (2421 frente a 2554, respectivamente), la primera dispone de una cantidad mucho mayor de muestras de entrenamiento (11656 frente a 995), que le permiten realizar un mejor aprendizaje de sus características.

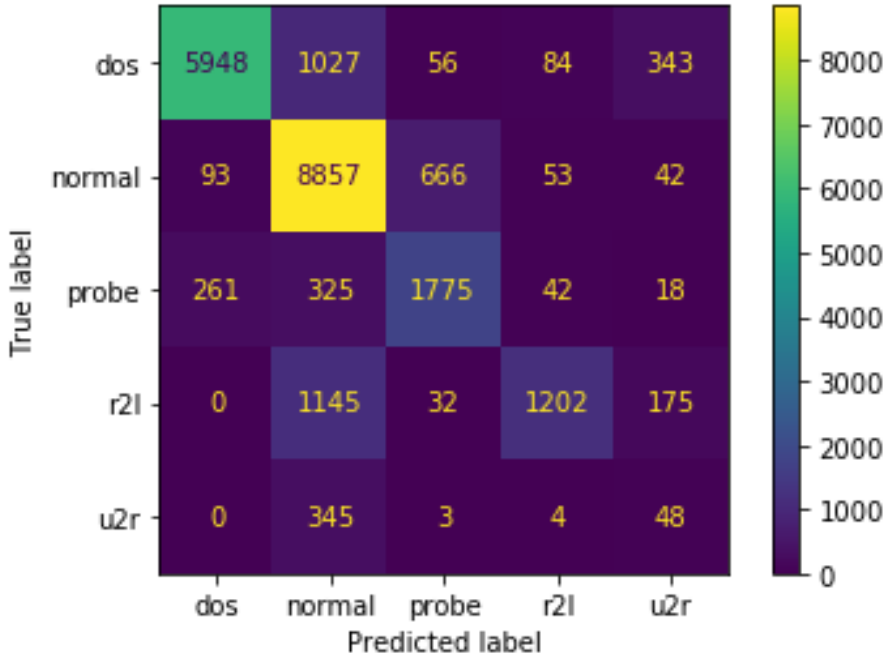

Figura 4.5 Matriz de confusión del ID-CVAE.

Para todas las clases diferentes de la Normal, la clase con la que sufren el mayor número de equivocaciones es, precisamente, dicha clase Normal. Esto hace que la Normal sea una clase con un gran número de falsos positivos, que provocan que su Precision decremente significativamente su valor respecto a su Recall (para el Recall, el motivo que ocasiona una disminución de su valor son los falsos negativos que, como se puede ver, son mucho menos numerosos).

En la clase DoS, por su parte, el comportamiento de la Precision y el Recall es justo el contrario, siendo mayor la Precision, debido al bajo número de falsos positivos (de hecho, por ejemplo, no hay ninguna muestra de las clases R2L y U2R clasificada erróneamente como DoS), y menor el Recall, fruto del importante número de falsos negativos obtenidos para la clase Normal y, en menor medida, para la U2R.

La clase R2L vuelve a presentar un comportamiento similar al de la clase DoS (mayor Precision y menor Recall) pero, en este caso, mucho más acentuado, siendo la Precision $= 0.8679$ y el Recall $= 0.4706$. Como se puede consultar en la Figura 4.5, el número de falsos negativos obtenidos en esta clase es superior al de muestras correctamente clasificadas, de ahí su valor de Recall inferior al 50 %.

Respecto a la clase Probe, no hay ningún aspecto novedoso a destacar, teniendo una Precision y Recall similares y, en general, un rendimiento intermedio entre las clases clasificadas con mayor calidad (DoS y Normal), y aquellas en las que se han conseguido unas peores prestaciones (R2L y U2R).

Por último, es necesario fijarse detalladamente en la clase U2R. Como se puede observar, los resultados de las diferentes métricas son cercanos a 0, excepto la Accuracy, cuyo valor no es significativo en sets de datos con una distribución de clases tan desbalanceada como el utilizado. Esto indica la incapacidad del ID-CVAE de aprender las propiedades relevantes de esta clase para su clasificación, hecho por otra parte ya esperado, debido a las pocas muestras disponibles de la misma en ambos sets de datos, aún más grave en el caso del dataset de entrenamiento, donde solo se cuenta con 52 muestras de este tipo de un total de 125973.

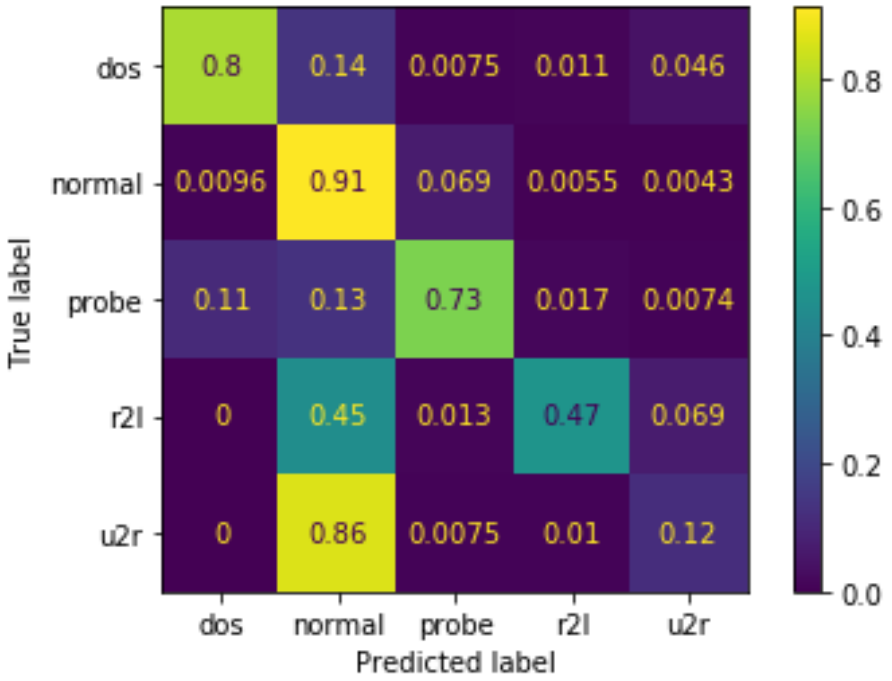

Figura 4.6 Matriz de confusión normalizada del ID-CVAE.

En general, prácticamente la totalidad de modelos de Machine Learning y Deep Learning que puedan aplicarse a esta base de datos, tendrán problemas para extraer características significativas de esta clase y, por tanto, grandes dificultades para clasificarla aceptablemente.

4.5 Modelos de Machine Learning

Por sí solos, los resultados de clasificación obtenidos por el ID-CVAE que se acaban de presentar no dicen mucho más acerca de si este modelo puede ser considerado de cierta calidad para resolver la tarea de clasificación de ataques en cuestión. Para su puesta en contexto, es necesario abordar dicha tarea con otra serie de métodos, y comparar así los resultados que se obtengan con los conseguidos por el ID-CVAE.

Concretamente, se han utilizado 4 modelos de amplia popularidad dentro del Machine Learning: Random Forest, SVM, Regresión Softmax y MLP, los cuales también son empleados en el artículo de referencia, [2]. Tal y como se ha hecho previamente para el ID-CVAE, en esta sección se mostrará la comparativa entre los resultados de clasificación de estos métodos logrados en [2] y los de este TFM.

Primeramente, como es práctica habitual en Machine Learning, se han evaluado estos modelos con sus diferentes parámetros por defecto, con el objetivo de comprobar si realmente son candidatos para afrontar con cierto éxito esta tarea de clasificación (Código A.5). Una vez visto que, efectivamente, es así, en el Código A.6 se ha procedido a la optimización de los mismos, buscando aquellas configuraciones de sus parámetros que maximicen sus rendimientos.

En la Figura 4.7/Tabla 4.5 y la Figura 4.8/Tabla 4.6 pueden observarse las correspondientes métricas de rendimiento de los 4 algoritmos de Machine Learning empleados en este TFM y en el artículo de referencia, respectivamente.

Como se muestra, los resultados de ambos trabajos presentan bastantes similitudes aunque, evidentemente, no llegan al grado de parecido producido anteriormente entre los dos ID-CVAEs.

Excepto para la Regresión Softmax, donde el rendimiento obtenido en este TFM es claramente superior al del artículo de referencia, el comportamiento del resto de casos es muy parejo, siendo el Random Forest de [2] ligeramente peor que el obtenido aquí, la SVM de dicho artículo un poco mejor que la de este TFM, y el MLP que puede considerarse prácticamente de la misma calidad, puesto que su F1 es levemente superior en este TFM (métrica más importante), pero el resto de métricas inferiores a las de [2].

En cuanto a las causas de estas diferencias obtenidas, se acusan principalmente a la versión de la librería de Scikit-Learn utilizada. En ambos trabajos se ha empleado dicha librería para realizar la correspondiente

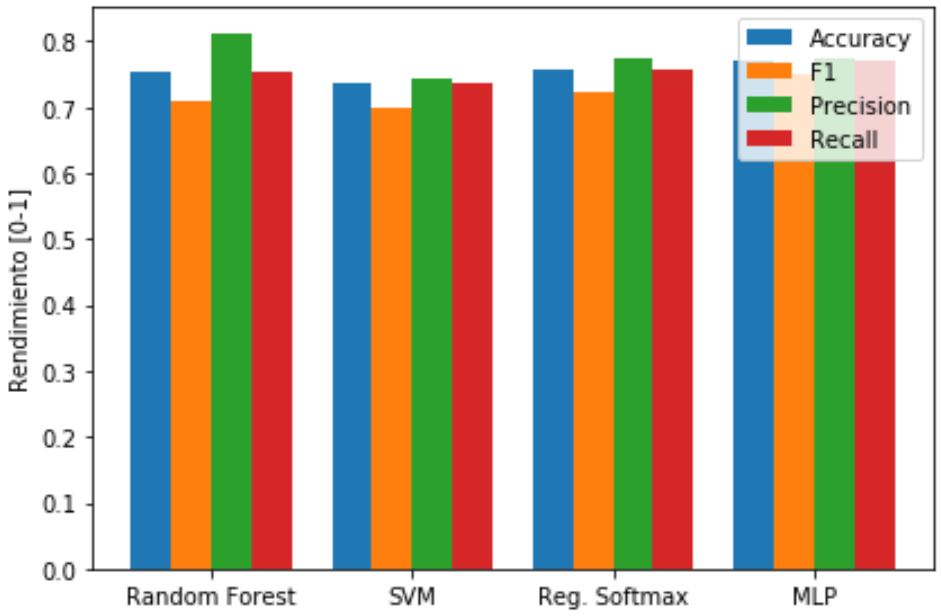

Figura 4.7 Métricas de rendimiento de los diferentes modelos de Machine Learning del TFM.

Tabla 4.5 Métricas de rendimiento de los diferentes modelos de Machine Learning del TFM.

	Random Forest	**SVM**	**Reg. Softmax**	**MLP**
Accuracy	0.7528	0.7381	0.7559	0.7690
F1	0.7096	0.6981	0.7219	0.7500
Precision	0.8107	0.7423	0.7730	0.7729
Recall	0.7528	0.7381	0.7559	0.7690

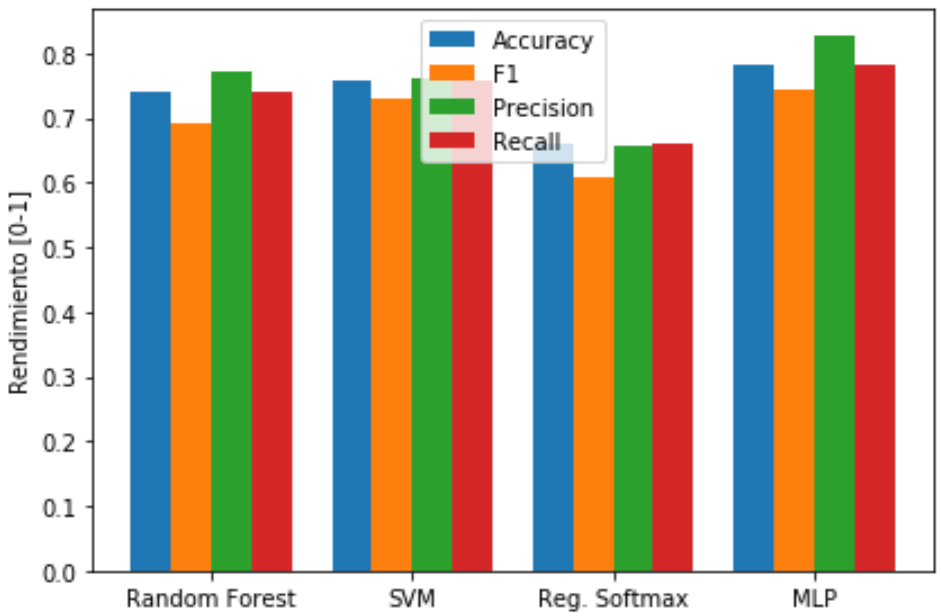

Figura 4.8 Métricas de rendimiento de los diferentes modelos de Machine Learning del artículo de referencia.

Tabla 4.6 Métricas de rendimiento de los diferentes modelos de Machine Learning del artículo de referencia.

	Random Forest	SVM	Reg. Softmax	MLP
Accuracy	0.7391	0.7560	0.6602	0.7831
F1	0.6909	0.7295	0.6066	0.7429
Precision	0.7708	0.7622	0.6570	0.8262
Recall	0.7391	0.7560	0.6602	0.7831

implementación de los modelos y, a nivel de los parámetros considerados a optimizar, no debe haber demasiadas diferencias ya que, principalmente, los parámetros a ajustar en este tipo de modelos son aquellos relacionados con la regularización de los mismos, con el objetivo de evitar el sobreajuste.

Para el caso de la Regresión Softmax, por ejemplo, el modelo inicial implementado en este TFM con todos los parámetros por defecto ya supera sobradamente el rendimiento del mostrado en el citado artículo, hecho que refuerza aún más el argumento dado acerca del motivo de estas diferencias (cabe recordar que el artículo de referencia data de Septiembre de 2017, donde aún no existía la versión de Scikit-Learn utilizada en este TFM).

4.6 ID-CVAE vs Modelos de Machine Learning

Finalmente, una vez expuesto todo el trabajo práctico desarrollado, se procede a recapitular los resultados de clasificación conseguidos por los distintos modelos utilizados en este TFM, con el objetivo de mostrar su comparativa de forma directa. Esto es lo que se recoge en la Figura 4.9 y la Tabla 4.7.

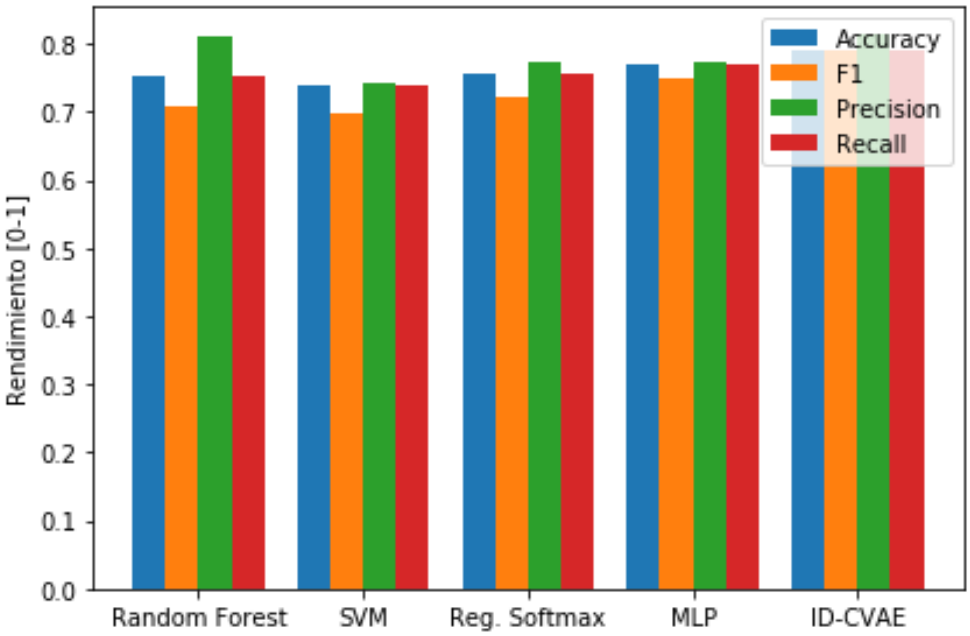

Figura 4.9 Comparativa de rendimiento entre el ID-CVAE y el resto de modelos.

Como se puede observar, el ID-CVAE es claramente el modelo que posee un mejor rendimiento, siendo todas sus métricas las más altas obtenidas, tan solo acercándose la Precision del Random Forest a la de dicho ID-CVAE.

De entre el resto de modelos empleados, el MLP es el que presenta unas mejores prestaciones, lo cual tiene cierta lógica ya que, al final, es un tipo de arquitectura de red neuronal, como lo es el ID-CVAE, aunque, evidentemente, este segundo de una mucho mayor complejidad. Esto parece indicar que los modelos de Deep Learning, basados en redes neuronales, pueden afrontar de una mejor forma esta tarea de clasificación de

Tabla 4.7 Comparativa de rendimiento entre el ID-CVAE y el resto de modelos.

	Random Forest	**SVM**	**Reg. Softmax**	**MLP**	**ID-CVAE**
Accuracy	0.7528	0.7381	0.7559	0.7690	0.7909
F1	0.7096	0.6981	0.7219	0.7500	0.7902
Precision	0.8107	0.7423	0.7730	0.7729	0.8133
Recall	0.7528	0.7381	0.7559	0.7690	0.7909

ataques que otros modelos de Machine Learning clásicos, como es el caso del Random Forest, la SVM y la Regresión Softmax.

5 Conclusiones y líneas futuras

La variedad de escenarios de aplicación del Internet de las Cosas es tan extensa como la de los numerosos ataques que sufrirán estas redes IoT y los dispositivos presentes en ellas. Para garantizar el desarrollo ilimitado de esta tecnología, uno de los aspectos fundamentales a considerar será el de la seguridad y, en ese sentido, la Inteligencia Artificial tiene mucho que aportar.

En la actualidad, dentro de los Sistemas de Detección de Intrusiones, principales encargados de la protección de los equipos y el mantenimiento de la red a salvo de amenazas, la línea de investigación dominante es aquella referida a los NIDSs y, más concretamente, a los que emplean como técnica para la detección de estas intrusiones un mecanismo basado en anomalía.

Aquí, justamente, es donde se sitúa el modelo estudiado e implementado en este trabajo, el ID-CVAE. A pesar de los múltiples modelos existentes en el campo, este ID-CVAE no es un caso más, sino que cuenta con una relevancia bastante especial. A fecha de Septiembre de 2017, fecha de su publicación, supuso la primera aplicación de un CVAE en el mismo y, además, ofrecía el primer algoritmo capaz de realizar recuperación de características. Este segundo aspecto, fuera del alcance fijado para este TFM, otorga al ID-CVAE un reconocimiento aún mayor que el conseguido por sus buenos resultados de clasificación.

En cuanto a su utilización como clasificador, se ha visto como el ID-CVAE es capaz de conseguir un rendimiento superior al de modelos típicos del Machine Learning, como son el Random Forest, las SVMs, la Regresión Softmax y los MLPs y, además, en su comparativa respecto a modelos similares, como el VAE tradicional, también resulta victorioso, debido a su mayor flexibilidad y menor complejidad de la solución.

Como se explicó en su momento, el ID-CVAE necesita realizar únicamente una etapa durante su fase de entrenamiento, mientras que el VAE tradicional requiere tantas como número de clases existan en la tarea de clasificación, siendo, por tanto, mucho más costoso computacionalmente, debido a la enorme cantidad de recursos que requiere el entrenar un modelo de Deep Learning.

Dados los buenos resultados obtenidos por este ID-CVAE sobre la base de datos empleada y, apoyándose también en el hecho de que el MLP sea el modelo con un mejor rendimiento de entre los otros utilizados, parece claro concluir que los modelos basados en redes neuronales tienen mayor capacidad de resolver exitosamente esta tarea de clasificación que otros modelos de Machine Learning típicos del estilo de los aquí empleados (Random Forest, SVM, Regresión Softmax). Por ello, como continuación de este trabajo, sería interesante abordar la tarea de clasificación propuesta con variantes del VAE, como el Ladder VAE o el Structured VAE, así como probar la otra gran familia de modelos generativos conocida, las GANs (Generative Adversarial Networks), con el objetivo de conseguir mejorar aún más los resultados de clasificación mostrados.

Por otra parte, también sería de gran interés explotar la otra aplicación posible del ID-CVAE mencionada anteriormente, su capacidad para recuperar características de muestras incompletas. Los datos recibidos por un dispositivo presente en una red IoT pueden poseer errores, ocasionados por su propia transmisión (mayormente inalámbrica) a través de dicha red o por otra serie de causas. Gracias a este método, sería posible recuperar aquella parte corrupta de los datos de un determinado paquete, permitiendo así la utilización del mismo sin la necesidad de un nuevo reenvío por parte del transmisor o, directamente, evitando perder esa información sino hay reenvío posible.

Para conseguir realizar esta recuperación de características, basta partir del modelo del ID-CVAE ya visto, cuya implementación se encuentra en el correspondiente Código A.3 del Apéndice A, y someterlo a un determinado proceso, de forma similar a lo hecho para su utilización como clasificador, pero de una mayor complejidad desde el punto de vista práctico (para mayor detalle, consultar [2]).

Apéndice A
Códigos

A continuación, se incluyen los códigos desarrollados en este trabajo, los cuales permiten obtener los diferentes resultados presentados en el Capítulo 4. Como se indicó en dicho capítulo, la implementación de los mismos se ha realizado en la plataforma Anaconda 3-2020.02 [13] y, más concretamente, empleando el entorno mostrado en el Código A.1 [18].

Código A.1 environment.yml.

```
name: tf2
channels:
  - conda-forge
  - defaults
dependencies:
  - graphviz
  - imageio=2.6.1
  - ipython=7.10.1
  - ipywidgets=7.5.1
  - joblib=0.14.0
  - jupyter=1.0.0
  - matplotlib=3.1.2
  - nbdime=1.1.0
  - nltk=3.4.5
  - numexpr=2.7.0
  - numpy=1.17.3
  - pandas=0.25.3
  - pillow=6.2.1
  - pip
  - py-xgboost=0.90
  - pydot=1.4.1
  - pyopengl=3.1.3b2
  - python=3.7.3
  - python-graphviz
  - requests=2.22.0
  - scikit-image=0.16.2
  - scikit-learn=0.22
  - scipy=1.3.1
  - tqdm=4.40.0
  - tensorboard=2.0.0
  - tensorflow=2.0.0 # or tensorflow-gpu if gpu
  - tensorflow-datasets=1.2.0
  - tensorflow-estimator=2.0.0
  - tensorflow-hub=0.7.0
```

```
  - tensorflow-probability=0.7
  - wheel
  - widgetsnbextension=3.5.1
  - pip:
    - atari-py==0.2.6
    - gym==0.15.4
    - opencv-python==4.1.2.30
    - psutil==5.6.7
    - pyglet==1.3.2
    - tensorflow-addons==0.6.0
    - tensorflow-data-validation==0.15.0
    - tensorflow-metadata==0.15.1
    - tensorflow-model-analysis==0.15.4
    - tensorflow-serving-api==2.0.0 # or tensorflow-serving-api-gpu if gpu
    - tensorflow-transform==0.15.0
    - tf-agents==0.3.0rc0
    - tfx==0.15.0
    - urlextract==0.13.0
    #- pyvirtualdisplay # add if on headless server
```

Código A.2 functions.py.

```
def acc_score_class(conf_mx, num_labels):
    import numpy as np
    accuracy_score_class = np.zeros(num_labels)
    for i in range(num_labels):
        tp = conf_mx[i,i]
        tn = sum(sum(conf_mx))-sum(conf_mx[i,:])-sum(conf_mx[:,i])+tp;
        accuracy_score_class[i] = (tp+tn)/(sum(sum(conf_mx)))
    return accuracy_score_class

def display_scores(scores):
    print("Scores =", scores)
    print("Mean =", scores.mean())
    print("Standard Deviation =", scores.std())

def prepare_data(train, test, dl):
    from sklearn.preprocessing import OneHotEncoder, MinMaxScaler
    from sklearn.compose import ColumnTransformer

    # Datasets without the labels and the last column, that isn't employed in
    # this system
    train2 = train.drop([41,42], axis=1)
    test2 = test.drop([41,42], axis=1)

    # Labels of the datasets
    train2_labels = train[41].copy()
    test2_labels = test[41].copy()

    # Categorization of the different attacks in the final labels used in the
    # system
    train2_labels = train2_labels.replace(["back","land","neptune","pod", \
                                        "smurf","teardrop"], "dos")
    train2_labels = train2_labels.replace(["ipsweep","nmap","portsweep", \
                                        "satan"], "probe")
```

```
train2_labels = train2_labels.replace(["ftp_write","guess_passwd","imap", \
                                       "multihop","phf","spy", \
                                          "warezclient","warezmaster"], \
                                      "r2l")
train2_labels = train2_labels.replace(["buffer_overflow","loadmodule", \
                                       "perl","rootkit"], "u2r")

if dl==2:
    test2_labels = test2_labels.replace(["back","land","neptune","pod", \
                                         "smurf","teardrop"], "dos")
    test2_labels = test2_labels.replace(["ipsweep","nmap","portsweep", \
                                         "satan"], "probe")
    test2_labels = test2_labels.replace(["ftp_write","guess_passwd", \
                                         "imap","multihop","phf","spy", \
                                            "warezclient","warezmaster"], \
                                        "r2l")
    test2_labels = test2_labels.replace(["buffer_overflow","loadmodule", \
                                         "perl","rootkit"], "u2r")
else:
    test2_labels = test2_labels.replace(["apache2","back","land", \
                                         "mailbomb","neptune","pod", \
                                            "processtable","smurf", \
                                               "teardrop","udpstorm"], \
                                        "dos")
    test2_labels = test2_labels.replace(["ipsweep","mscan","nmap", \
                                         "portsweep","saint","satan"], \
                                        "probe")
    test2_labels = test2_labels.replace(["ftp_write","guess_passwd", \
                                         "httptunnel","imap","multihop", \
                                            "named","phf","sendmail", \
                                               "snmpgetattack", \
                                                  "warezmaster","xlock", \
                                                     "xsnoop"], "r2l")
    test2_labels = test2_labels.replace(["buffer_overflow","loadmodule", \
                                         "perl","ps","rootkit","snmpguess", \
                                            "sqlattack","worm","xterm"], \
                                        "u2r")

# Numerical features, without the 6 not useful
train2_num = train2.drop([1,2,3,6,8,14,17,19,20], axis=1)

# Categorical features
train2_cat = train2[[1,2,3]]

# Numerical features scaling to the [0-1] range and one-hot encoding of the
# categorical features
num_attribs = list(train2_num)
cat_attribs = list(train2_cat)

full_pipeline = ColumnTransformer([
        ("num", MinMaxScaler(), num_attribs),
        ("cat", OneHotEncoder(), cat_attribs),
    ])

train2 = full_pipeline.fit_transform(train2)
test2 = full_pipeline.transform(test2)
```

```
    # Transformation of the original labels to numerical labels for Deep
    # Learning
    if dl==1 or dl==2:
        from sklearn.preprocessing import LabelEncoder
        le = LabelEncoder()
        train2_labels = le.fit_transform(train2_labels)
        test2_labels = le.transform(test2_labels)

    if dl==1:
        return train2, test2, train2_labels, test2_labels, le
    else:
        return train2, test2, train2_labels, test2_labels

def print_perf_metric_class(metric_name, metric_score_class, le):
    print("\t %s:" % metric_name)
    for i in range(len(le.classes_)):
        print("\t \t %s = %.4f " %(le.classes_[i], metric_score_class[i]))

def rmse(true,pred):
    import numpy as np
    return (np.sqrt(np.mean(np.square(true-pred),1)))
```

Código A.3 ID-CVAE_train.py.

```
from pandas import read_csv
from pandas import DataFrame
from sklearn.model_selection import train_test_split
from functions import prepare_data

from tensorflow.keras.layers import Lambda, Input, Dense
from tensorflow.keras.layers import concatenate
from tensorflow.keras.models import Model

from tensorflow.keras.losses import binary_crossentropy
from tensorflow.keras.utils import plot_model
from tensorflow.keras import backend as K
from tensorflow.keras.utils import to_categorical

import numpy as np
import matplotlib.pyplot as plt

# Reparameterization trick
# Instead of sampling from Q(z|X), sample eps = N(0,I)
# z = z_mean + sqrt(var)*eps
def sampling(args):
    """Reparameterization trick by sampling
        from an isotropic unit Gaussian.
    # Arguments:
        args (tensor): mean and log of variance of Q(z|X)
    # Returns:
        z (tensor): sampled latent vector
    """

    z_mean, z_log_var = args
    batch = K.shape(z_mean)[0]
```

```
    dim = K.int_shape(z_mean)[1]
    # By default, random_normal has mean=0 and std=1.0
    epsilon = K.random_normal(shape=(batch, dim))
    return z_mean + K.exp(0.5 * z_log_var) * epsilon

# Load training data
train = read_csv('C:/Users/josem/Desktop/2º/Trabajo Fin de Máster/NSL-KDD/
    KDDTrain+.txt', header=None)

# Prepare the data
train, valid = train_test_split(train, test_size=0.1, random_state=42)
train_prepared, valid_prepared, train_labels, valid_labels = prepare_data \
    (train, valid, 2)

# Number of features of the samples
original_dim=train_prepared.shape[1]

# Number of labels
num_labels = len(np.unique(train_labels))

# Network parameters
input_shape = (original_dim, )
label_shape = (num_labels, )
intermediate_dim = 500
intermediate_dim_L = 495
batch_size = 256
latent_dim = 25
epochs = 250

# CVAE model = encoder + decoder
# Encoder model
inputs = Input(shape=input_shape, name='encoder_input')
x = Dense(intermediate_dim, activation='relu')(inputs)
x = Dense(intermediate_dim, activation='relu')(x)
x = Dense(intermediate_dim, activation='relu')(x)
z_mean = Dense(latent_dim, name='z_mean')(x)
z_log_var = Dense(latent_dim, name='z_log_var')(x)

# Use reparameterization trick to push the sampling out as input
z = Lambda(sampling,
          output_shape=(latent_dim,),
          name='z')([z_mean, z_log_var])

# Instantiate encoder model
encoder = Model(inputs, [z_mean, z_log_var, z], name='encoder')
# encoder.summary()
# plot_model(encoder, to_file='id-cvae_encoder.png', show_shapes=True)

# Decoder model
latent_inputs = Input(shape=(latent_dim,), name='z_sampling')
y_labels = Input(shape=label_shape, name='class_labels')
x = Dense(intermediate_dim_L, activation='relu')(latent_inputs)
x = concatenate([x, y_labels])
x = Dense(intermediate_dim, activation='relu')(x)
x = Dense(intermediate_dim, activation='relu')(x)
outputs = Dense(original_dim, activation='sigmoid')(x)
```

```
# Instantiate decoder model
decoder = Model([latent_inputs, y_labels],
                outputs,
                name='decoder')
# decoder.summary()
# plot_model(decoder, to_file='id-cvae_decoder.png', show_shapes=True)

# Instantiate CVAE model
outputs = decoder([encoder(inputs)[2], y_labels])
cvae = Model([inputs, y_labels], outputs, name='cvae')
# cvae.summary()
# plot_model(cvae, to_file='id-cvae.png', show_shapes=True)

# CVAE loss = xent_loss + kl_loss
reconstruction_loss = binary_crossentropy(inputs, outputs)
reconstruction_loss *= original_dim
kl_loss = 1 + z_log_var - K.square(z_mean) - K.exp(z_log_var)
kl_loss = K.sum(kl_loss, axis=-1)
kl_loss *= -0.5
cvae_loss = K.mean(reconstruction_loss + kl_loss)
cvae.add_loss(cvae_loss)
cvae.compile(optimizer='adam')

# Train the autoencoder
history = cvae.fit([train_prepared, to_categorical(train_labels)],
        epochs=epochs,
        batch_size=batch_size,
        validation_data=([valid_prepared, to_categorical(valid_labels)], None))


DataFrame(history.history).plot(figsize=(8, 5))
plt.grid(True)
plt.gca().set_ylim(5, 10)
plt.savefig("curves_id-cvae_250_256.png")
plt.show()

# Save the trained CVAE
cvae.save("id-cvae_250_256")
```

Código A.4 ID-CVAE_test.py.

```
from pandas import read_csv
from functions import acc_score_class, prepare_data, print_perf_metric_class, \
    rmse
from tensorflow.keras.models import load_model
from tensorflow.keras.utils import to_categorical
import numpy as np
from sklearn.metrics import accuracy_score, f1_score, precision_score, \
    recall_score, confusion_matrix, ConfusionMatrixDisplay

# Load training and test data
train = read_csv('C:/Users/josem/Desktop/2º/Trabajo Fin de Máster/NSL-KDD/
    KDDTrain+.txt', header=None)
test = read_csv('C:/Users/josem/Desktop/2º/Trabajo Fin de Máster/NSL-KDD/
    KDDTest+.txt', header=None)
```

```
# Prepare the data
train_prepared, test_prepared, train_labels, test_labels, le = prepare_data \
    (train, test, 1)

# Number of samples of the test data
nsamples_test = test_prepared.shape[0]

# Number of labels
num_labels = len(np.unique(train_labels))

# Load of the model already trained and configured
cvae = load_model("id-cvae_250_256")

# Prediction phase
for i in range(num_labels):
    labels_1class = i*np.ones(nsamples_test)
    test_pred = cvae.predict([test_prepared, to_categorical(labels_1class, \
                                                    num_labels)])
    if i==0:
        dist = (rmse(test_prepared, test_pred)).reshape(nsamples_test,1)
    else:
        dist = np.concatenate((dist,(rmse(test_prepared, test_pred)).reshape \
                           (nsamples_test,1)),axis=1)

final_predictions = np.argmin(dist,axis=1)

# Recovery of the non-numerical labels from numerical labels
test_labels_text = le.inverse_transform(test_labels)
final_predictions_text = le.inverse_transform(final_predictions)

# Evaluation of test data
print("Classification performance metrics (aggregated):")
print("\t Accuracy = %.4f" % accuracy_score(test_labels_text, \
                                        final_predictions_text))
print("\t F1 = %.4f" % f1_score(test_labels_text, final_predictions_text, \
                              average='weighted'))
print("\t Precision = %.4f" % precision_score(test_labels_text, \
                                          final_predictions_text, \
                                              average='weighted'))
print("\t Recall = %.4f" % recall_score(test_labels_text, \
                                    final_predictions_text, \
                                        average='weighted'))

conf_mx = confusion_matrix(test_labels_text, final_predictions_text)
print("Confusion Matrix: ")
print(conf_mx)
conf_mx_d = ConfusionMatrixDisplay(conf_mx, display_labels=le.classes_)
conf_mx_d.plot(values_format='g')

norm_conf_mx = confusion_matrix(test_labels_text, final_predictions_text, \
                              normalize='true')
print("Normalized Confusion Matrix: ")
print(norm_conf_mx)
norm_conf_mx_d = ConfusionMatrixDisplay(norm_conf_mx, display_labels= \
                                    le.classes_)
norm_conf_mx_d.plot()
```

```
accuracy_score_class = acc_score_class(conf_mx, num_labels)
f1_score_class = f1_score(test_labels_text, final_predictions_text, \
                        average=None)
precision_score_class = precision_score(test_labels_text, \
                                    final_predictions_text, average=None)
recall_score_class = recall_score(test_labels_text, final_predictions_text, \
                                average=None)

print("Classification performance metrics (by class):")
print_perf_metric_class("Accuracy", accuracy_score_class, le)
print_perf_metric_class("F1", f1_score_class, le)
print_perf_metric_class("Precision", precision_score_class, le)
print_perf_metric_class("Recall", recall_score_class, le)
```

Código A.5 models.py.

```
from pandas import read_csv
from functions import display_scores, prepare_data
from sklearn.model_selection import cross_val_score

# Load training and test data
train = read_csv('C:/Users/josem/Desktop/2º/Trabajo Fin de Máster/NSL-KDD/
    KDDTrain+.txt', header=None)
test = read_csv('C:/Users/josem/Desktop/2º/Trabajo Fin de Máster/NSL-KDD/
    KDDTest+.txt', header=None)

# Prepare the data
train_prepared, _, train_labels, _ = prepare_data(train, test, 0)

# Model selection
model = "rf"

if model == "rf":
    from sklearn.ensemble import RandomForestClassifier
    clf = RandomForestClassifier(n_jobs=-1)
elif  model == "svm":
    from sklearn.svm import LinearSVC
    clf = LinearSVC(dual=False)
elif model == "softmax":
    from sklearn.linear_model import LogisticRegression
    clf = LogisticRegression(multi_class="multinomial", solver="sag")
elif model == "mlp":
    from sklearn.neural_network import MLPClassifier
    clf = MLPClassifier()

# Evaluation using cross-validation
scores = cross_val_score(clf, train_prepared, train_labels, scoring= \
                       "f1_weighted", cv=10)
display_scores(scores)
```

Código A.6 models_optimization.py.

```
from pandas import read_csv
from functions import prepare_data
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import accuracy_score, precision_score, recall_score

# Load training and test data
train = read_csv('C:/Users/josem/Desktop/2º/Trabajo Fin de Máster/NSL-KDD/
    KDDTrain+.txt', header=None)
test = read_csv('C:/Users/josem/Desktop/2º/Trabajo Fin de Máster/NSL-KDD/
    KDDTest+.txt', header=None)

# Prepare the data
train_prepared, test_prepared, train_labels, test_labels = prepare_data \
    (train, test, 0)

# Model selection
model = "rf"

if model == "rf":
    param_grid = [
      {'n_estimators': [50,75], 'max_features': ['auto'], 'max_depth': [50], \
       'min_samples_split': [5,10]},
    ]

    from sklearn.ensemble import RandomForestClassifier
    clf = RandomForestClassifier(n_jobs=-1)
elif  model == "svm":
    param_grid = [
    {'C': [10,20,30], 'max_iter': [1000]},
    ]

    from sklearn.svm import LinearSVC
    clf = LinearSVC(dual=False)
elif model == "softmax":
    param_grid = [
    {'C': [20,30], 'max_iter': [400]},
    ]

    from sklearn.linear_model import LogisticRegression
    clf = LogisticRegression(multi_class="multinomial", solver="sag")
elif model == "mlp":
    param_grid = [
    {'hidden_layer_sizes':[(200,50)], 'activation': ['relu'], 'alpha': \
     [1.e-03,5.e-04,1.e-04]},
    ]

    from sklearn.neural_network import MLPClassifier
    clf = MLPClassifier()

# Evaluation using GridSearchCV
grid_search = GridSearchCV(clf, param_grid, cv=10, scoring="f1_weighted", \
                           return_train_score=True)
grid_search.fit(train_prepared, train_labels)

print("Best hyperparameters:", grid_search.best_params_)
```

```
cvres = grid_search.cv_results_
for mean_score, params in zip(cvres["mean_test_score"], cvres["params"]):
    print("Score of", params, "= %.4f" % mean_score)

final_model = grid_search.best_estimator_

# Evaluation of test data
final_predictions = final_model.predict(test_prepared)

print("Classification performance metrics (aggregated) with the best
    hyperparameters:")
print("\t Accuracy = %.4f" % accuracy_score(test_labels, final_predictions))
print("\t F1 = %.4f" % grid_search.score(test_prepared, test_labels))
print("\t Precision = %.4f" % precision_score(test_labels, final_predictions, \
                                        average='weighted'))
print("\t Recall = %.4f" % recall_score(test_labels, final_predictions, \
                                    average='weighted'))
```

Índice de Figuras

Índice de Tablas

Índice de Códigos

Bibliografía

[1] M. H. Bhuyan, D. K. Bhattacharyya, and J. K. Kalita, "Network Anomaly Detection: Methods, Systems and Tools," *IEEE*, Junio 2013.

[2] M. López-Martín, B. Carro, A. Sánchez-Esguevillas, and J. Lloret, "Conditional Variational Autoencoder for Prediction and Feature Recovery Applied to Intrusion Detection in IoT," *Sensors (Basel)*, Septiembre 2017.

[3] F. Chollet, *Deep Learning with Python*. Manning, 2018.

[4] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[5] A. Géron, *Hands-On Machine Learning with Scikit-Learn and TensorFlow*, 1st ed. O'Reilly, 2017.

[6] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine Learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[7] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, R. Layton, J. VanderPlas, A. Joly, B. Holt, and G. Varoquaux, "API design for machine learning software: experiences from the scikit-learn project," in *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, 2013, pp. 108–122.

[8] A. Géron, *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*, 2nd ed. O'Reilly, 2019.

[9] R. Atienza, *Advanced Deep Learning with TensorFlow 2 and Keras*, 2nd ed. Packt, 2020.

[10] Y. LeCun, C. Cortes, and C. J. Burges. MNIST handwritten digit database. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[11] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol, "Extracting and Composing Robust Features with Denoising Autoencoders," in *ICML 2008: Proceedings of the 25th International Conference on Machine Learning*, 2008, pp. 1096–1103. [Online]. Available: https://icml.cc/2008/papers/592.pdf

[12] A. Coates and A. Y. Ng, "The Importance of Encoding Versus Training with Sparse Coding and Vector Quantization," in *ICML 2011: Proceedings of the 28th International Conference on International Conference on Machine Learning*, 2011, pp. 921–928. [Online]. Available: https://icml.cc/2011/papers/485_icmlpaper.pdf

[13] Anaconda, "Anaconda Software Distribution," Marzo 2020, Computer software: Vers. 3-2020.02. [Online]. Available: https://anaconda.com

[14] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster,

J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems," 2015, Software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[15] C. I. for Cybersecurity. NSL-KDD. [Online]. Available: https://www.unb.ca/cic/datasets/nsl.html

[16] M. Tavallaee, E. Bagheri, W. Lu, and A. A. Ghorbani, "A Detailed Analysis of the KDD CUP 99 data Set," *IEEE*, Diciembre 2009.

[17] B. Ingre and A. Yadav, "Performance Analysis of NSL-KDD dataset using ANN," *IEEE*, Marzo 2015.

[18] J. J. Murillo Fuentes, *Asignatura de Análisis de Datos y Procesado de la Información, del Máster en Ingeniería de Telecomunicación*. Dpto. Teoría de la Señal y Comunicaciones, Escuela Técnica Superior de Ingeniería, Universidad de Sevilla, Segundo cuatrimestre 2019/20.

[19] F. Chollet *et al.*, "Keras," 2015. [Online]. Available: https://keras.io

[20] T. pandas development team, "pandas-dev/pandas: Pandas," Febrero 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3509134

[21] Wes McKinney, "Data Structures for Statistical Computing in Python," in *Proceedings of the 9th Python in Science Conference*, Stéfan van der Walt and Jarrod Millman, Eds., 2010, pp. 56 – 61. [Online]. Available: https://doi.org/10.25080/Majora-92bf1922-00a

[22] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Septiembre 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2

[23] J. D. Hunter, "Matplotlib: A 2D Graphics Environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007. [Online]. Available: https://doi.org/10.5281/zenodo.3563226