TRABAJO FIN DE GRADO

ATAQUES ADVERSARIOS EN EL APRENDIZAJE AUTOMÁTICO: CAUSAS Y UNA POSIBLE SOLUCIÓN

TRABAJO FIN DE GRADO PARA LA OBTENCIÓN DEL TÍTULO DE **GRADUADO EN INGENIERÍA EN TECNOLOGÍAS INDUSTRIALES**

FEBRERO **2022**

Laura Salguero Hinojosa

DIRECTOR DEL TRABAJO FIN DE GRADO:

Miguel Ruiz García

INDUSTRIALES
ETSII | UPM

POLITÉCNICA

Universidad Politécnica de Madrid

Escuela Técnica Superior de Ingenieros Industriales

Ataques adversarios en el aprendizaje automático: causas y una posible solución

Trabajo Fin de Grado

Grado de Ingeniería en Tecnologías Industriales
Departamento de Matemáticas Aplicadas a la Ingeniería Industrial

Autor: **Laura Salguero Hinojosa**
Tutor: **Miguel Ruiz García**

Febrero 2022

"Todos tenemos una reserva de fuerza insospechada
que emerge cuando la vida nos pone a prueba."

- Isabel Allende

Agradecimientos

No miento si digo que siento nervios y emoción al mismo tiempo que escribo estas líneas, porque sólo quiere decir una cosa: lo he conseguido.

En primer lugar, quiero dar las gracias a mi familia: papá, mamá y Ana. Por haber tenido paciencia conmigo cada día, haberme enseñado tanto y haberme convertido en la persona que soy hoy: una hija y una hermana orgullosa de vosotros. Gracias.

En segundo lugar, quiero dar las gracias a Miguel, por haberme dado la oportunidad de hacer este trabajo con él. Tu apoyo desde el primer momento ha sido fundamental para haber llegado hasta aquí, ojala hubiera más profesores como tú.

Las siguientes líneas van dedicadas a la persona que más me ha enseñado a conocerme y a enfrentarme a mis inseguridades estos años de la carrera. Este trabajo te lo dedico a ti, Oss. Gracias por ser una gran referencia (literalmente) en mi vida.

Gracias a Mike, por ser mi sombra desde que me levanto, a Ceci, por el cariño que siempre regalas desinteresadamente y a Adei, por la transparencia y la alegría que desprendes. Qué suerte haber coincidido con vosotros, Filipinos.

Gracias a dos personas muy especiales que no han faltado nunca cuando lo he necesitado. Por lo mucho que me cuidáis y por ser la luz que necesito cuando estoy apagada. Bea y Mayi, me dáis vida. Gracias.

Gracias a Nacho, por confiar tanto en mí y tener la risa más divertida del mundo. A Delba, por quedarte. A Huanma, por su alegría contagiosa. A Yaño, por ser magic. Y a Enri, por ser un amigo de verdad.

Gracias al baloncesto, por haberme regalado tantos buenos momentos y una de las amistades más sinceras y bonitas que he tenido nunca: mis dos grandes apoyos, Marta y Ampa, por tantas horas y canciones compartidas, y mis chicas Pati, Pili y Lillo.

Gracias a Irene, una de mis primeras amigas desde que tengo uso de razón. Por seguir a mi lado a pesar del tiempo y la distancia.

Gracias a Gabriel, Palas, Flum, Marcos y Gp, mis Originals, por haberme sabido aguantar desde el primer día en que llegué a Madrid. Sois un pilar fundamental.

Gracias a toda mi familia de Granada, por quererme tanto. Mención especial a Paula, por emocionarme con cada carta, y al yeyo, a quien tanto echo de menos. Ojalá hubieras podido ver cómo acabo esta etapa de mi vida.

Gracias a los que se han ido, a los que han llegado y a los que llegarán, porque siempre me sentiré afortunada por todos aquellos que dediquen parte de su tiempo en facilitarme este camino tan largo al que llamamos vida.

- GRACIAS -

Resumen

El *Machine Learning*, o aprendizaje automático, consiste en un subconjunto de técnicas de la Inteligencia Artificial que permite a las máquinas aprender a través de la *experiencia*, definida por grandes conjuntos de datos [1]. Dentro de este campo tan amplio y cuya expansión continúa en constante crecimiento, cabe destacar el *Deep Learning* o aprendizaje automático, un algoritmo estructurado en varias capas cuyo aprendizaje viene a asemejarse al procedimiento que siguen las redes neuronales del cerebro, hablándose así de las redes neuronales artificiales. Estos dos conceptos no sólo están en boca de todos, si no que también *en mano de todos*. Sin ir más lejos, los asistentes inteligentes que nos acompañan cada día en nuestros dispositivos móviles funcionan gracias al desarrollo de estas nuevas tecnologías.

Las redes neuronales artificiales están constituidas por varias capas: *Input Layer* o capa de entrada, *Hidden Layer* o capas ocultas y *Ouput Layer* o capa de salida. La complejidad de esta herramienta está en el modelo computacional de la neurona, operación que se desarrolla en las capas ocultas. El objetivo del aprendizaje de las redes es ajustar de la manera más óptima los parámetros internos que definen el modelo, de manera que se minimice el error cometido entre la salida estimada y la salida deseada o real. La función que define el error de la predicción se denomina Función de coste o de pérdida (*Loss Function*). El método utilizado para minimizar esta función y así ajustar los parámetros del modelo es el que se conoce como Descenso del Gradiente [9], que permitirá calcular cuánto es necesario modificar los parámetros para apreciar un cambio en la función de coste, tomando los valores que la minimizan. Este procedimiento se realizará de manera reiterativa hasta conseguir una buena precisión de los resultados.

La rápida integración de la Inteligencia Artificial en la vida cotidiana y la masiva cantidad de datos que maneja han despertado, además de interés, inquietudes sobre la vulnerabilidad de estos modelos de aprendizaje ante posibles manipulaciones. Hasta los modelos más robustos entrenados con miles de datos, como es el caso de la *ResNet-50*, un algoritmo entrenado con 50.000 imágenes, han mostrado su fragilidad ante ciertos ataques. Es aquí donde entra en juego el papel del Ataque Adversario.

Un ataque adversario se define como una manipulación aplicada a cualquier tipo de dato de entrada que se suministra a la red neuronal [20]. Se caracteriza por ser imperceptible al ojo humano al mismo tiempo que el modelo, previamente entrenado, lo percibe como un dato totalmente diferente al original, llevándole a cometer un error en su clasificación, como si de una ilusión óptica se tratara.

La programación de estos ataques resulta intuitiva [25]. Dada, por ejemplo, una imagen de entrada, el procedimiento para configurar el ataque será modificar la imagen lo

suficiente como para que sea inapreciable a simple vista a la vez que *burla* a la red. Para ello, se vuelve a hacer uso del Descenso del Gradiente, pero resulta interesante que su función en este caso será distinta a la mencionada con anterioridad. En el caso del entrenamiento de la red, el gradiente se calculaba respecto a los parámetros internos para ajustarlos y mejorar la precisión del modelo minimizando la función de coste. En el caso de los ataques, el gradiente se calculará con respecto a la propia entrada para obtener cuánto habría que modificar la imagen para apreciar un cambio en la función de coste (el error de la estimación). Además, en lugar de minimizarla interesará maximizarla para *forzar* a la red a predecir una clase diferente a la verdadera. La modificación de la imagen puede interpretarse como la adición de *ruido* inapreciable.

Un caso particular de los ataques son los ataques dirigidos. Como su propio nombre dice, su objetivo no es sólo conseguir que la red erre al clasificar a la entada, si no dirigir el ataque hacia una clase objetivo. En este caso, se maximizará la pérdida con respecto a la clase original (minimizando la probabilidad de ser la clase verdadera) y se minimizará la pérdida con respecto a la clase objetivo (maximizando su probabilidad). Estos ataques pueden conseguir que los *engaños* a la red sean más sorprendentes seleccionando una clase objetivo totalmente diferente a la original.

Un ejemplo visual para facilitar al lector la asimilación y comprensión de toda esta información sería el siguiente:

En primer lugar, se seleccionada una entrada y se comprueba que la red logra calificarla correctamente con una precisión casi del 100 %.

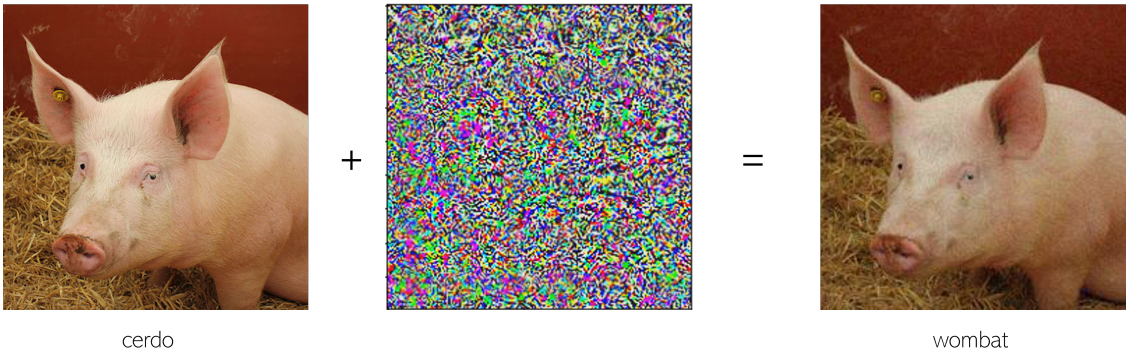
(a) Imagen original.

```
Predicted class:  hog
Predicted probability: 0.9961252808570862
```

(b) Predicción de la red.

Figura 1: Imagen original y predicción del modelo antes de aplicar el ataque adversario.

A continuación, se aplicará una perturbación que conseguirá cambiar la salida del modelo. En este caso se ha dirigido de manera que la red lo confunda con un *wombat*, algo que para el ser humano sería imposible de predecir a partir de la nueva imagen obtenida.



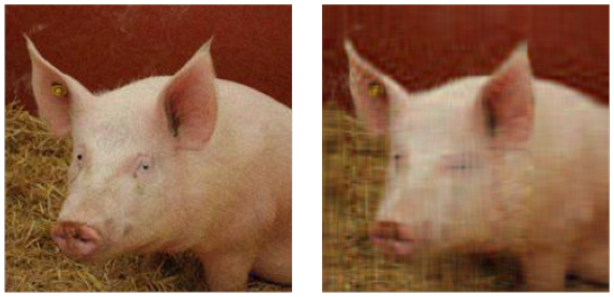
(a) Adición de ruido a la imagen original.

```
True class probability: 0.00031623418908566236
Predicted class:  wombat
Predicted probability: 0.9989402890205383
```

(b) Predicción de la red.

Figura 2: Imagen resultante y predicción del modelo tras el ataque adversario dirigido.

Para prevenir estos ataques se propone en este trabajo la aplicación del *Principal Component Analysis* (PCA) o Análisis de Componentes Principales [26]. Consiste en un método estadístico que permite transformar espacios de dimensión p a otra más pequeña k ($k < p$) a la vez que conserva la mayor información posible. El objetivo de recurrir al uso de PCA es el siguiente: comprimir la imagen [27] a la que le ha sido aplicado el ataque adversario, de manera que se modifique el *ruido* y que, tras enseñar la nueva imagen a la red neuronal, esta la califique correctamente.



(a) Ejemplo adversario antes y después de aplicar PCA.

```
Predicted class:  hog
Predicted probability: 0.7525399923324585
```

(b) Predicción de la red.

Figura 3: Imagen resultante de la aplicación del Análisis de Componentes Principales. El modelo consigue clasificarla como la clase verdadera.

Pero, ¿por qué ocurre esto? A partir de un conjunto de datos sintéticos se estudiará por qué ocurre este fenómeno, buscando respuestas para conocer cómo la dispersión y la dimensión de los datos afectan a la vulnerabilidad de las redes ante diferentes ataques adversarios.

Palabras clave

Red neuronal artificial, ataque adversario, ruido, perturbación, Análisis de Componentes Principales (PCA), dimensión, dispersión, vulnerabilidad.

Códigos UNESCO

1201.10 (Álgebra lineal), 1203.02 (Lenguajes algorítmicos), 1203.04 (Inteligencia Artificial), 1203.23 (Lenguajes de programación), 1209.03 (Análisis de datos)

Índice general

Capítulo 1

Introducción

1.1. La IA, Machine Learning y Deep Learning

La Inteligencia Artificial (IA) es la disciplina que capacita a las máquinas a imitar comportamientos y capacidades propias de los seres humanos, como el razonamiento, el aprendizaje y la creatividad. Permite que las máquinas se relacionen con el entorno, y a base de recibir datos ya preparados, los procesen y resuelvan problemas respondiendo a ellos. Dentro de este campo, existen a su vez otros dos que es importante diferenciar y que, hoy en día, están en boca de todos: *Machine Learning* y *Deep Learning*. [1]

El *Machine Learning*, o aprendizaje automático, es un subconjunto de técnicas de Inteligencia Artificial que permite a las máquinas aprender a través de la experiencia. Esta *experiencia* se consigue a partir de un conjunto de datos de entrenamiento que se aporta al modelo y que deben estar preparados y bien organizados, pues sus futuras predicciones dependerán de ellos. Además de los datos será necesario hacer uso de diferentes modelos que tras el entrenamiento llevarán a cabo diferentes algoritmos, elegidos en función del tipo y volumen de los datos de entrenamiento y del tipo de problema que haya que resolver. Se diferencian tres grupos en función de la metodología que siga el aprendizaje:

- **Aprendizaje supervisado (*Supervised Machine Learning*)**: El conjunto de datos de entrenamiento está previamente etiquetado y clasificado, por lo que el algoritmo aprenderá a qué categoría pertenece cada uno de ellos. Es un método recurrente para problemas de clasificación y regresión. (Figura 1.1)
- **Aprendizaje no supervisado (*Unsupervised Machine Learning*)**: El conjunto de datos de entrada no está etiquetado y no se dispone conocimiento previo acerca de ellos, por lo que el algoritmo buscará patrones subyacentes. Se suele utilizar para encontrar reglas de asociación o agrupar datos con características similares (*clustering*). (Figura 1.2)
- **Aprendizaje por refuerzo (*Reinforcement Learning*)**: Los algoritmos característicos de este aprendizaje presentan grandes similitudes con la psicología conductista de los humanos, aprendiendo a base de errores (penalizaciones) y aciertos (recompensas). Se utiliza, entre otras cosas, para el aprendizaje de los robots. (Figura 1.3)

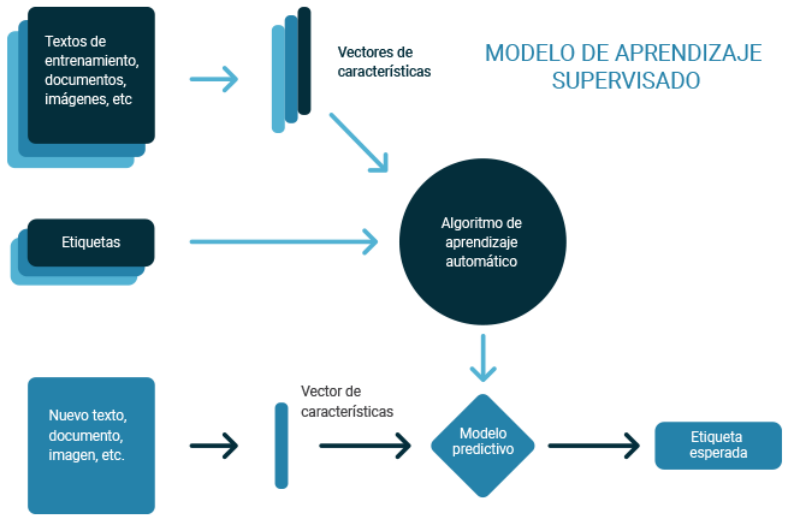


Figura 1.1: Esquema de la metodología del aprendizaje supervisado [2].

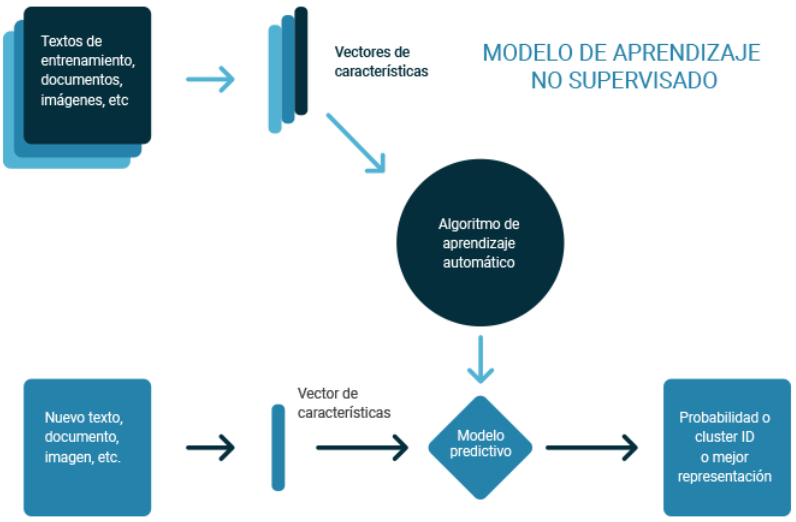


Figura 1.2: Esquema de la metodología del aprendizaje no supervisado [2].

El entrenamiento consistirá en un proceso que se repetirá hasta conseguir que los distintos algoritmos obtengan el resultado esperado y con una buena precisión. Será entonces cuando se obtenga un modelo de *Machine Learning* que pueda ser aplicado con nuevos datos que nunca antes haya visto. Por lo general, cuanto mayor sea el tiempo de aprendizaje y mayor sea el número de datos, mejor será el rendimiento de esta herramienta.

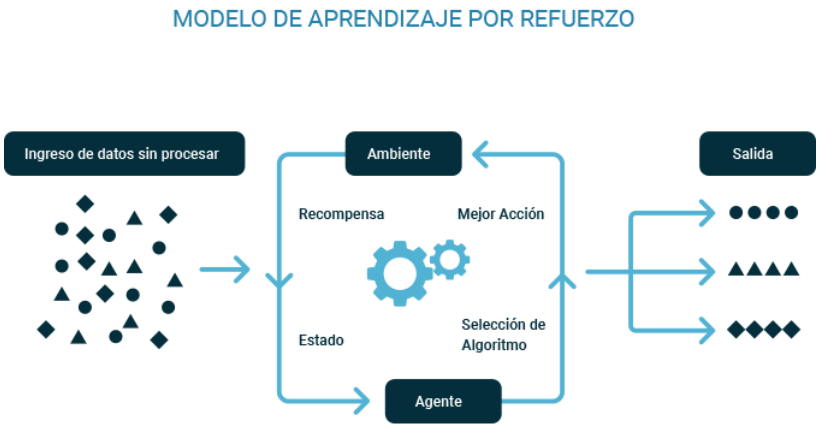


Figura 1.3: Esquema de la metodología del aprendizaje por refuerzo [2].

Por último, cabe mencionar el *Deep Learning* o aprendizaje profundo, que pertenece a un subcampo del *Machine Learning*. Se trata de un algoritmo automático estructurado que sigue un proceso por capas, simulando la función que realizan las redes neuronales del cerebro, de ahí a que este aprendizaje se caracterice por utilizar redes neuronales artificiales entrelazadas.

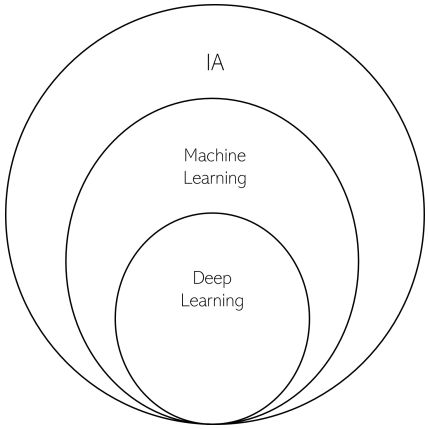


Figura 1.4: Esquema que representa el orden de jerarquía entre la IA, el *Machine Learning* y el *Deep Learning*.

1.2. La importancia de los ataques adversarios

El *Machine Learning* y el *Deep Learning* han incrementado drásticamente el apetito por comprender adecuadamente y extraer conclusiones de los datos en los últimos años. Según un artículo de enero de 2022 de *Computerworld* [3], una revista sobre tecnologías de información y comunicación, en el primer lugar de la lista de los 20 trabajos más demandados en el mercado laboral español están los puestos relacionados con las ingenierías del desarrollo y las operaciones, en cuarta posición, el de ingeniero de *Machine Learning*, y en quinto lugar, el perfil de experto en ciberseguridad. Este auge en el interés por contratar y formar plantilla con perfiles tan específicos relacionadas con la tecnología refleja la transformación de la era en la que vivimos que, tal y como señala Klaus Schwab, fundador y presidente ejecutivo del Foro Económico Mundial y autor de La cuarta revolución industrial, se caracteriza por una «fusión de tecnologías que difumina las fronteras entre lo físico, lo digital y lo biológico». Pero, al igual que incrementa la expectación por estas tecnologías, también se disparan a un ritmo vertiginoso una gran variedad de incertidumbres: *¿Acabarán las máquinas dirigiendo la vida de las personas? ¿Alcanzará la inteligencia artificial el nivel humano? ¿Cuántos empleos desaparecerán y cómo se sobrellevará su pérdida? ¿Dónde está el límite de estas tecnologías?* El temor a lo desconocido.

Sin embargo, estas no son las únicas inquietudes que existen con el desarrollo de la inteligencia artificial. ¿Cómo de vulnerables son, por ejemplo, las redes neuronales ante ataques de manipulación? Pese a ser métodos muy robustos, se ha demostrado que engañar a una red neuronal es algo más sencillo de lo que parece, y cuando su aplicación está tan implementada especialmente en sistemas de seguridad de muchos dispositivos móviles (detección facial, de huella dactilar, etc), esta vulnerabilidad comienza a resultar un tanto preocupante.

Según un artículo en *New Scientist* [4], un equipo de investigadores de la Universidad Bar-Ilan, en colaboración con el equipo de IA de Facebook, demostró el funcionamiento del algoritmo Houdini [5], que inserta rasgos inapreciables para un humano tanto en archivos de audio como en imágenes que hacen que la máquina altere totalmente su interpretación de la realidad. Para ello, sólo hay que añadir una capa de ruido imperceptible para el ojo u oído humano que contiene patrones concretos que la red asocia con otras palabras. Su aplicación en un clip de audio resultó de lo más interesante. Uno de los clips originales decía:

– *The fact that a man can recite a poem does not show he remembers any previous ocassion on which he has recited it or read it.*

Tras pasar este a través de Google Voice, el programa lo transcribió como:

– *The fact that a man can* ***decide*** *a poem does not show he remembers any previous ocassion on which he has* ***work cited*** *or read it.*

A excepción de esas dos expresiones, consigue transcribir el resto del audio correctamente. Sin embargo, al reproducir el audio modificado, percibido por humanos idéntico al original, la transcripción en esta ocasión fue:

– *The fact that* ***I can rest I'm just not sure that you heard there ir*** *any previous ocassion* ***I am at he has your side*** *it or read it.*

Estos ejemplos, todavía pendientes de investigación, permiten testear la fiabilidad y la robustez de los algoritmos de *Machine Learning*. Pero, *¿y si se les da un mal uso y se emplean para engañar a los sistemas de inteligencia artificial?*

1.3. Objetivos

Este tipo de *ilusiones* que permiten engañar a los modelos de *Machine Learning* son las que se conocen como Ataques Adversarios. El objetivo de este Trabajo de Fin de Grado (TFG) se basa en el estudio teórico de la vulnerabilidad de las redes ante diferentes ataques adversarios y en la propuesta de una técnica para prevernirlos.

En primer lugar, a partir de la red neuronal convolucional *ResNet-50*, entrenada previamente con 50.000 imágenes, se diseñará un ejemplo adversario correspondiente a una de las clases de entrenamiento, de manera que el modelo lo identifique perteneciente a una clase errónea. Se analizará en detalle la programación del ataque, y cómo, mediante la aplicación del Análisis de Componentes Principales (PCA) se consigue prevenir.

En segundo lugar, a partir de un conjunto de datos sintéticos distribuidos en forma de espiral, se hará un estudio teórico sobre cómo afecta la dimensión y la dispersión de los datos a la vulnerabilidad ante los ataques.

Una vez terminado el TFG, se terminará de completar la investigación para su publicación en un artículo científico.

Capítulo 2

Redes neuronales artificiales

2.1. ¿Qué son las redes neuronales artificiales?

El nombre asignado a este algoritmo se basa en el extremo parecido que comparte con las neuronas del cerebro.

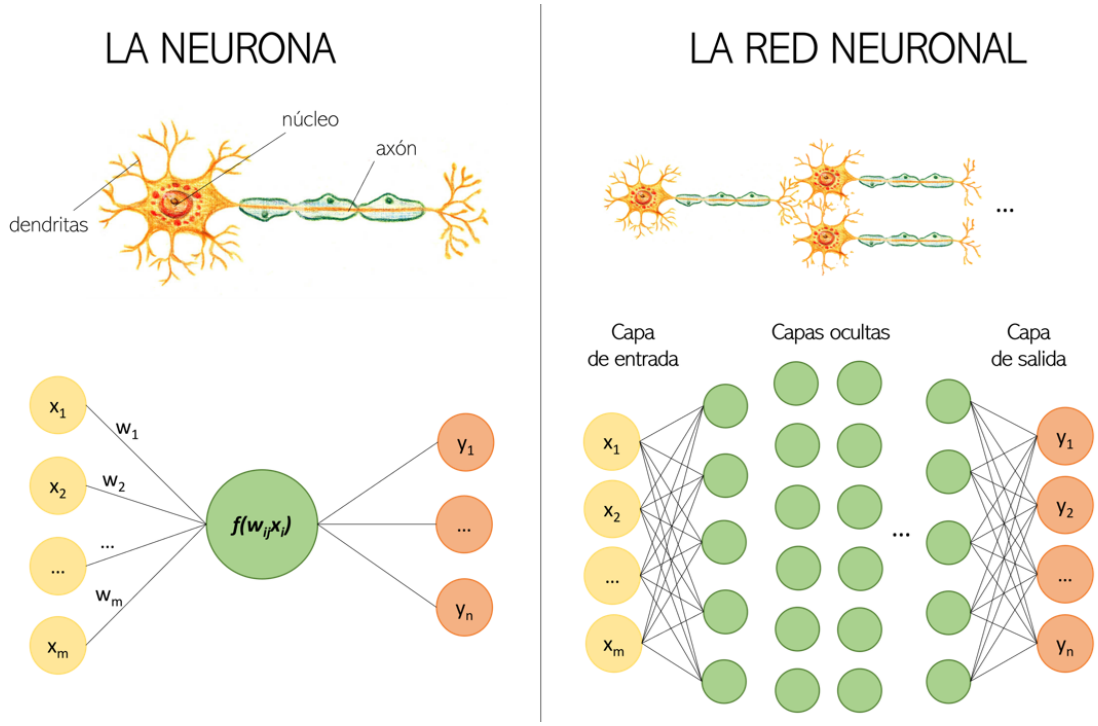


Figura 2.1: Comparación entre las estructura de una neurona y red neuronal naturales y unas artificiales.

La red está formada por la *Input Layer* o capa de entrada, *Hidden Layer* o capas ocultas y la *Ouput Layer* o capa de salida.

En la imagen anterior se plantea la estructura que sigue el modelo matemático de la red neuronal en comparación con una neurona biológica. Ahora bien, visto su funcionamiento no es difícil darse cuenta de que no sólo comparten la misma apariencia física. Las neuronas reciben señales externas de entrada por sus dendritas (lo que serían los datos de entrada (x_1, x_2,..., x_m)), y a través del axón, que se ramifica, se producen unas

señales de salida (y_1, y_2,..., y_n). Durante la sinapsis, que es clave en el proceso de gestión y control del cuerpo, la neurona transmitirá esa información a otra neurona a través de sus dendritas, formando así una red neuronal. Dependiendo del número de capas ocultas (representadas en verde), donde se procesa la información proporcionada por la capa de entrada, se hablará de red neuronal simple o profunda.

La complejidad de esta herramienta está en el modelo computacional de la neurona, lo que ocurriría en el axón. Los m valores de entrada, X, que son variables independientes, son multiplicados por sus respectivos pesos, W, que equivalen a la fuerza de la señal que se transmite por cada sinapsis y proporcionan la importancia de la entrada dentro de la función de agregación de la neurona. Como resultado se obtiene una combinación lineal de las entradas y los pesos, que se denominará función de ponderación, a la que se le añade un sesgo, b [16].

$$z_j = b_j + \sum_i x_i \cdot w_{ij} \tag{2.1}$$

Donde x_i hace referencia a cada una de las m entradas para una capa formada por n neuronas, y w_{ij} al peso asociado a la sinapsis que conecta la entrada i-ésima con la neurona j-ésima.

Seguidamente, se aplicará una función activación, a, encargada de transmitir la información generada a partir de la función de ponderación hacia las conexiones de salida (de la que se hablará con más detalle más adelante), tal que:

$$y_j = a\left(b_j + \sum_i x_i \cdot w_{ij}\right) \tag{2.2}$$

Y finalmente, el resultado obtenido se propaga a la salida, siendo la nueva entrada de una neurona o bien siendo el resultado final, la respuesta a nuestra entrada. [6]

El número de capas y el número de neuronas por capas son algunos de los hiperparámetros que dependerán de la elección del programador, al igual que la función de activación. Sin embargo, los parámetros internos, los pesos y sesgos, será la propia red quien los modifique. El cómo se verá en el apartado dedicado al descenso del gradiente.

2.2. Funciones de activación

Una función de activación, como se ha mencionado en el anterior apartado, es una función que transmite la información generada a partir de la función de ponderación hacia las conexiones de salida [12]. Estas se utilizan para proporcionar un comportamiento no lineal a la red, permitiendo que se adapten y sean capaces de resolver problemas más complejos. En la mayoría de casos, la aplicación de las funciones de activación es necesaria ya que, en caso contrario, la concatenación de capas de neuronas equivaldría a la concatenación de varias operaciones lineales en una sola operación lineal que, como ya se ha visto antes, sería lo mismo que una sola capa. Por tanto serán necesarias para poder adquirir esa gran potencia que se obtiene de añadir muchas capas. Según el tipo de salida requerido, se distinguen las siguientes funciones:

- Función escalón [*Threshold Logic Units*]: Se trata de una función de activación binaria que propaga un 0 si el valor de z es negativo, o un 1 si es positivo. Es la función más rígida y clasifica de forma estricta.

$$y = u(z) = \begin{cases} 0 & \text{si } z < 0 \\ 1 & \text{si } z \geq 0 \end{cases} \tag{2.3}$$

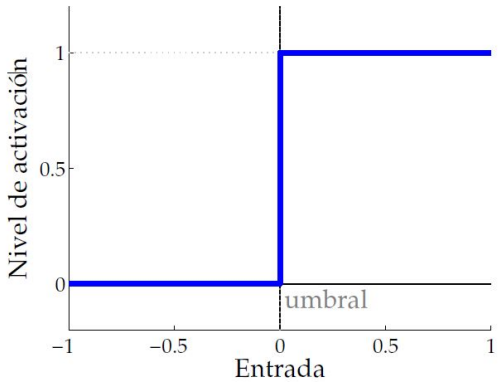


Figura 2.2: Función escalón. [7]

- Función sigmoide: A su vez existen diferentes tipos de funciones sigmoidales:
  - Función Logística: por su aplicación en la Regresión Logística, es una función de activación binaria que se caracteriza por ser derivable, lo que es útil en redes que se entrenan usando *backpropagation*. Útil para clasificar con valores categóricos y para predecir las probabilidades que tienen los datos de entrada de pertenecer a cada categoría, siendo 0 un suceso imposible y 1 un suceso seguro.

$$y = \sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.4}$$

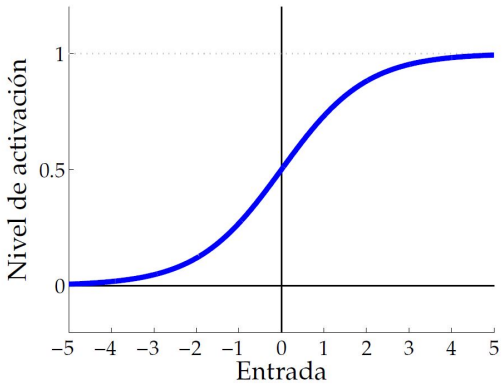


Figura 2.3: Función sigmoide. [7]

– Función tangente hiperbólica: Esta función se utiliza en trigonometría esférica, y no es más que una versión derivada de la función logística. En este caso los valores estarán comprendidos entre -1 y 1, y al tener media 0 funcionará bien para conjuntos de datos cuya media esté cercano a este valor.

$$y = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} = \frac{1 - e^{-2z}}{1 + e^{-2z}} \tag{2.5}$$

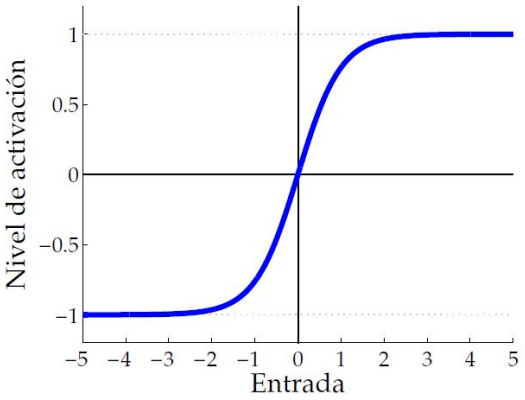


Figura 2.4: Función tangente hiperbólica. [7]

- Función lineal rectificadora (ReLU): Se trata de una función que obvia todas aquellas entradas que tras haber formado la función de ponderación tengan un valor negativo, mientras que mantiene el valor de aquellas con valor positivo. Por tanto los valores obtenidos siempre serán mayor o igual que 0, eliminando la restricción de estar comprendidos entre 0 y 1.

$$y = f_{\text{relu}}(z) = \begin{cases} z & \text{si } z \leq 0 \\ 0 & \text{si } z < 0 \end{cases} \tag{2.6}$$

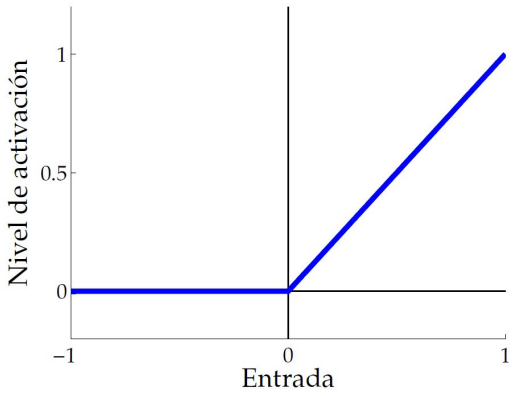


Figura 2.5: Función ReLU. [7]

2.3. Función Softmax

Volviendo a la estructura de la red, es necesario recordar que si se plantea un problema de clasificación con más de dos clases, la capa de salida de la red deberá incluir necesariamente más neuronas, en concreto el número de clases que se distingan en la entrada. Es decir, para K clases diferentes se utilizarán K neuronas en la capa de salida.

En problemas de clasificación en los que existan más de 2 clases diferentes, puede resultar interesante interpretar los niveles de activación de salida como estimaciones de la probabilidad que tenga un dato nuevo de pertenecer a una de las clases planteadas en el problema. Estas entradas netas, denominadas *logits*, no están acotadas. Para conseguir que la red neuronal devuelva una distribución de probabilidad definida sobre las diferentes de clases de partida en forma de vector de probabilidades se utiliza la función *Softmax* [7]:

$$y_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \tag{2.7}$$

Siendo z el valor de las neuronas de la última capa.Los K valores obtenidos se interpretan como las probabilidades que estima la red de que cada clase pueda ser la clase correcta para el ejemplo correspondiente a la entrada de la red. Como en cualquier distribución de probabilidad, la suma de todas debe ser igual a 1. Es habitual que la capa de salida de una red que utilice esta función reciba el nombre de capa *Softmax*.

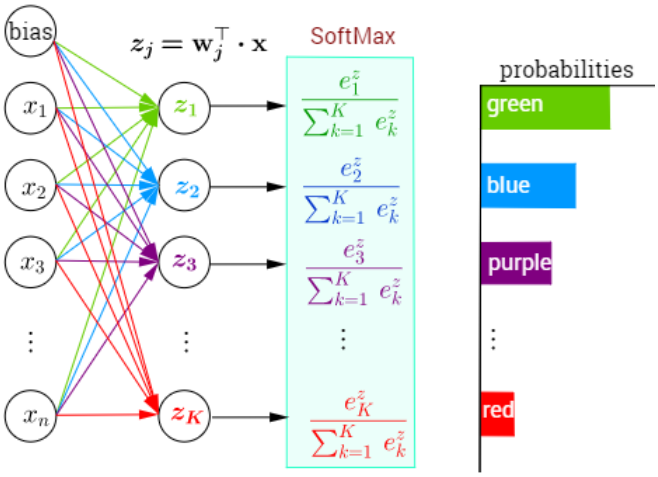


Figura 2.6: Ejemplo de aplicación de la función *Softmax* en un problema de clasificación. Los valores de probabilidades representados son los que la red asocia a cada una de las salidas.

Con este ejemplo se entiende de manera visual el resultado proporcionado por la función *Softmax*. Partiendo del dato de entrada, la red tras su entrenamiento estima las probabilidades que tiene de pertenecer a las diferentes clases, concluyendo que la clase correcta será la *clase green*, por su mayor probabilidad.

2.4. Funciones de coste

El objetivo del entrenamiento de la red es que los valores de salida sean lo más parecido posible a los valores reales que se deberían obtener. Los pesos por los que se multiplican los valores de entrada comienzan siendo valores aleatorios (se inicializan como un vector aleatorio muestreado de un gaussiano multidimensional), por lo que interesará ajustarlos consiguiendo que la red aprenda en la dirección que se desea. Es entonces cuando entra en juego la función de coste, también denominada función de pérdida (*Loss Function*), que trata de determinar el error entre el valor estimado y el valor real, con el fin de optimizar los parámetros de la red neuronal y minimizar la diferencia con cada iteración [13]. Algunas de ellas son:

- Error Cuadrático Medio (MSE)
- Error Absoluto Medio (MAE)
- Error Absoluto Medio Escalado (MASE)
- Entropía cruzada categórica (*Categorial Cross-Entropy*)
- Entropía cruzada binaria (*Binary Cross-Entropy*)

2.4.1. Error Cuadrático Medio (MSE)

Junto a la función *Cross-Entropy*, que es la que ha sido la utilizada en este proyecto y en la que se entrará más en detalle más adelante, el cálculo del Error Cuadrático Medio (MSE) es otra de las funciones de coste más utilizadas. Se trata del criterio de evaluación del funcionamiento de la red más utilizado en problemas de regresión, especialmente en casos de aprendizaje automático supervisado.

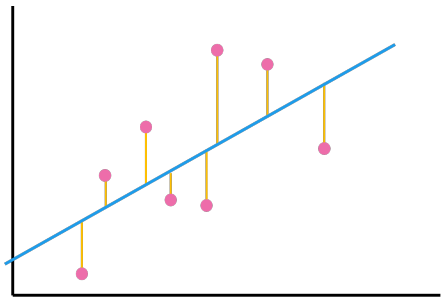


Figura 2.7: Aplicación del Error Cuadrático Medio.

En la Figura 2.7 se representa un ejemplo de la aplicación de la función del Error Cuadrático Medio función para facilitar su entendimiento:

- Los puntos rosas hacen referencia a la fuente de datos aleatorios de entrada en la red.
- La línea azul es la predicción que hace la red, tratando de ajustarse al máximo a los puntos verdes.
- Las líneas amarillas representan los errores de la predicción, la distancia desde el valor real (punto rosa) hasta su valor estimado (proyección del punto rosa en la línea azul).

El cálculo del error lo proporcionará la siguiente ecuación matemática [17]:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2 \tag{2.8}$$

Siendo n el número total de datos (puntos rosas), y_i la coordenada en el eje y del valor real e $\tilde{y}_i$ la coordenada del valor estimado. El objetivo de esta función de coste será, por tanto, minimizar el promedio de los errores elevados al cuadrado. Visto la forma que adopta el MSE en la siguiente gráfica, es fácil ver que amplifica los mayores errores obtenidos al calcular el cuadrado de estos.

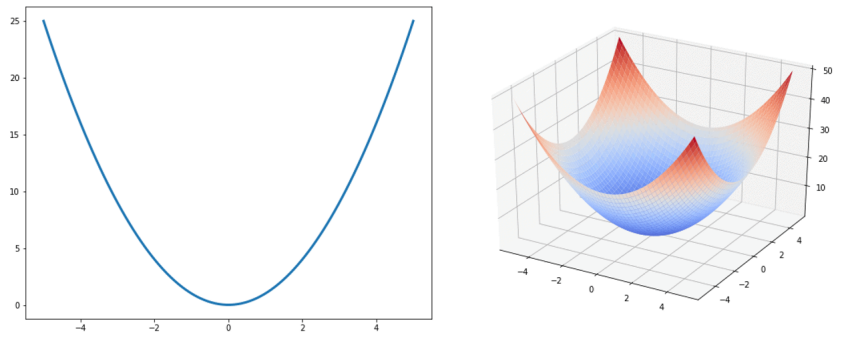


Figura 2.8: Representación del Error Cuadrático Medio. [14]

2.5. Entrenamiento de la red multicapa

Una red neuronal multicapa será la que se utilice en *Deep Learning*, formada por una capa de entrada, múltiples capas ocultas y una capa de salida. Dado un conjunto de datos de entrenamiento en forma (x, y), tal que x sea la entrada e y su salida correspondiente, el objetivo del algoritmo de la red será aproximar una función f de manera que para cada entrada x se obtenga una salida $y_j = f(x)$ lo más aproximada posible a la salida objetivo de entrenamiento, t_j (*target*). Los datos de entrada incluirán ruido que dificulte a priori el entrenamiento, pero a su vez lo potencie, proporcionando a la red la capacidad de generalizar con nuevas entradas que no haya visto nunca antes y obtener salidas apropiadas. Para ello habrá que ajustar correctamente tanto los parámetros internos (pesos y sesgo) como los hiperparámetros (número de capas y número de neuronas por capa).

2.5.1. Ajuste de los parámetros internos. Descenso del gradiente

El objetivo de conseguir una salida estimada de la red que sea similar a la salida real es el que genera la necesidad de minimizar el error de estimación, y con ello la función de coste o pérdida. El caso más sencillo de este problema sería el caso de una función convexa, que solo cuenta con un mínimo local, que coincide con el global, y que para calcularlo habría que resolver una sola ecuación, $f'(x) = 0$. El problema surge cuando la función es no-convexa y presenta varios mínimos locales. Además, puede aumentar la dificultad aparecen más zonas en la función con pendiente nula, como serían máximos locales o puntos de silla, lo que incrementaría además el número de ecuaciones. Es aquí dónde se valora la importancia del algoritmo del Descenso de Gradiente.

La inicialización de los pesos es preferible que sea aleatoria, para evitar cualquier simetría en la red y que todas las neuronas de una capa aprendan lo mismo, mientras que los parámetros de sesgo se inicializarán a cero. Una vez obtenida la función de coste se procederá a su optimización mediante el descenso del gradiente.

Este algoritmo calcula el gradiente como la derivada multivariable de la función de coste con respecto a todos los parámetros de la red, que proporcionará un vector con la dirección y sentido del máximo incremento positivo de la función (lo que gráficamente sería la pendiente). Sin embargo, como el objetivo es minimizar la función, interesará el sentido opuesto al señalado, buscando así la dirección que más rápido descienda para tratar de alcanzar el mínimo global. Para profundizar más consultar en [10] [9].

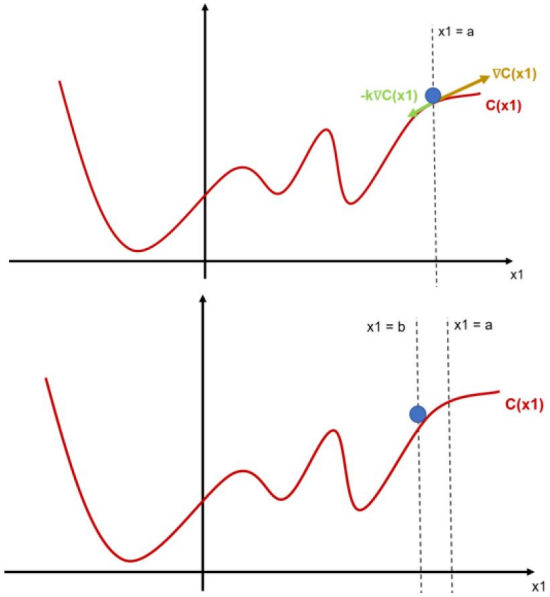


Figura 2.9: Interpretación gráfica del Descenso del gradiente. El vector amarillo corresponde al de dirección y sentido del máximo incremento positivo de la función de coste en ese punto de partida. El que interesará por lo tanto es el verde, de sentido contrario, siguiendo el objetivo de encontrar el mínimo de la función.[11]

El cálculo del gradiente no es algo sencillo debido a la gran cantidad de parámetros y a la multiplicidad de capas de la red, por lo que será necesario otro algoritmo como herramienta adicional.

Entre las décadas de 1950 y 1960, Frank Rosenblatt, una de las figuras más notables en el campo de la inteligencia artificial, desarrolló el *Perceptron*, el primer algortimo que impulsó y potenció el nacimiento y el desarrollo de las redes neuronales artificiales. Consiste en un tipo de clasificador binario lineal cuya frontera de decisión viene delimitada por una línea recta en un espacio dimensional, un plano en uno tridimensional, etc. Para ello, Rosenblatt introdujo el concepto de los pesos w asociados a cada una de las entradas, y que expresaban la *importancia* de la respectiva entrada con la salida, a los que él mismo asignaba valor manualmente [15]. Sin embargo, las limitaciones de este algoritmo, como que sólo fuera capaz de clasificar correctamente clases que fueran linealmente separables, dieron lugar al comienzo de un nuevo invierno de la IA. Se conoce así al periodo de tiempo en el que apenas se invierte en el campo o, como Kai-Fu Lee define en

su libro 'Superpotencias de la inteligencia artificial' (2020), al momento en que «la decepcionante falta de resultados prácticos conducía a importantes recortes de financiación».

Fue en 1986 cuando David Rumelhart y G.Hinton descubrieron el algoritmo de *backpropagation*, basado en la propagación hacia atrás del error, que hizo posible entrenar redes multicapa de manera supervisada. Este es el que permitirá calcular el gradiente necesario para minimizar la función de coste.

La idea que intenta plasmar *backpropagation* [8] es cómo varía la función de coste respecto a cada uno de los parámetros de la red: los pesos, w_{ij}, y el sesgo, b_j.

$$\frac{\partial C}{\partial w^L}, \frac{\partial C}{\partial b^L} \tag{2.9}$$

Con el superíndice se hará referencia a la capa sobre la que se calcula la derivada. En este caso se refiere a la última capa, *last* (L).

Para facilitar el entendimiento de este cálculo, es aconsejable analizar el *camino* que siguen estos parámetros hasta la función de coste. En primer lugar, comienzan combinándose para formar una función ponderada Z, a la que se le aplica una función de activación a, y que finalmente se evalúa en la función de coste C, proporcionando el error de la estimación. Este camino sería equivalente a una composición de funciones $C\left(a\left(Z^L\right)\right)$, cuya derivada es "fácil" de calcular aplicando la Regla de la cadena.

$$\frac{\partial C}{\partial w_{ij}^L} = \frac{\partial C}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial z_j^L} \cdot \frac{\partial z_j^L}{\partial w_{ij}^L} \tag{2.10}$$

$$\frac{\partial C}{\partial b_j^L} = \frac{\partial C}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial z_j^L} \cdot \frac{\partial z_j^L}{\partial b_j^L} \tag{2.11}$$

A su vez, el cálculo de estas derivadas parciales tienen un cálculo sencillo para el que no se requieren más conocimientos de los adquiridos en el instituto.

En primer lugar, se calculará la variación del coste respecto a la salida de la función de activación. El resultado de aplicar esta función es la propia salida de la red, por lo que a partir de ahora se hará referencia a la función a como y_j para facilitar la comprensión del algoritmo. La función de coste C, si se escoge, por ejemplo, el Error Cuadrático Medio, seguirá la siguiente expresión:

$$C\left(y_j^L\right) = \frac{1}{2}\sum_j \left(t_j - y_j^L\right)^2 \tag{2.12}$$

Donde y_j se corresponde con la predicción de la red (salida de la red y función de activación), y t la salida deseada. Su derivada, por lo tanto, será:

$$\frac{\partial C}{\partial y_j^L} = \left(y_j^L - t_j\right) \tag{2.13}$$

De la misma manera, la derivada de la función de activación con respecto a la suma ponderada z_j, será la derivada de la propia función de activación (o salida de la red). Para una sigmoide, la expresión quedaría:

$$\sigma(z_j^L) = \frac{1}{1 + e^{-z_j^L}} \tag{2.14}$$

$$\frac{\partial y_j^L}{\partial z_j^L} = \frac{\partial \sigma(z_j^L)}{\partial z_j^L} = \sigma'(z_j^L) = \frac{e^{-z_j^L}}{\left(1+e^{-z_j^L}\right)^2} = \sigma(z_j^L) \cdot (1 - \sigma(z_j^L)) \tag{2.15}$$

Finalmente, la variación de la suma ponderada con respecto al término de sesgo b_j y a los pesos w_{ij} quedarían tal que:

$$z_j^L = \sum_i y_i^{L-1} \cdot w_{ij}^L + b_j^L$$

$$\frac{\partial z_j^L}{\partial w_{ij} L} = y_i^{L-1} \tag{2.16}$$

$$\frac{\partial z^L}{\partial b^L} = 1$$

La derivada con respecto al sesgo resulta 1 por tratarse del término independiente, mientras que la correspondiente a los pesos es el valor de entrada a la neurona por el que se multiplica, es decir, el valor de la salida de las neuronas de la capa anterior.

Partiendo de las expresión de la Regla de la cadena 2.10 y 2.11, los dos primeros términos se pueden agrupar en un término que cuantifica cómo varía el coste con la variación de la suma ponderada de la neurona z_j, o en otras palabras, qué responsabilidad tendrá la neurona en el error de la estimación:

$$\frac{\partial C}{\partial y_j^L} \cdot \frac{\partial y_j^L}{\partial z_j^L} = \frac{\partial C}{\partial z_j^L} = \delta_j^L \tag{2.17}$$

Por lo tanto, las derivadas del coste respecto a los parámetros de la red quedarían así:

$$\begin{array}{l} \frac{\partial C}{\partial w_{ij}^L} = \delta_j^L \cdot y_i^{L-1} \\ \frac{\partial C}{\partial b_j^L} = \delta_j^L \end{array} \tag{2.18}$$

Ahora bien, estas expresiones están calculadas para la última capa de la red. Para calcular el resto simplemente habrá que extrapolar las obtenidas y definir una última ecuación.

Para la capa anterior, $L-1$, a partir de la Regla de la cadena se obtendrían estas dos expresiones:

$$\left\{ \begin{array}{l} \frac{\partial C}{\partial w_{ki}^{L-1}} = \frac{\partial C}{\partial y_j^L} \cdot \frac{\partial y_j^L}{\partial z_j^L} \cdot \frac{\partial z_j^L}{\partial y_i^{L-1}} \cdot \frac{\partial y_i^{L-1}}{\partial z_i^{L-1}} \cdot \frac{\partial z_i^{L-1}}{\partial w_{ki}^{L-1}} \\ \frac{\partial C}{\partial b_i^{L-1}} = \frac{\partial C}{\partial y_j^L} \cdot \frac{\partial y_j^L}{\partial z_j^L} \cdot \frac{\partial z_j^L}{\partial y_i^{L-1}} \cdot \frac{\partial y_i^{L-1}}{\partial z_i^{L-1}} \cdot \frac{\partial z_i^{L-1}}{\partial b_i^{L-1}} \end{array} \right. \tag{2.19}$$

A pesar de la apariencia tan poco amigable, satisface darse cuenta que varios de los términos de las dos expresiones ya han sido calculados previamente:

$$\frac{\partial C}{\partial y_j^L} \cdot \frac{\partial y_j^L}{\partial z_j^L} = \delta_j^L \tag{2.20}$$

$$\frac{\partial y_i^{L-1}}{\partial z_i^{L-1}} = \sigma'\left(z_i^{L-1}\right) = \sigma(z_i^{L-1}) \cdot (1 - \sigma(z_i^{L-1})) \tag{2.21}$$

$$\frac{\partial z_i^{L-1}}{\partial w_{ki}^{L-1}} = y_k^{L-2} \tag{2.22}$$

$$\frac{\partial z_i^{L-1}}{\partial b_i^{L-1}} = 1 \tag{2.23}$$

Por lo que quedaría por resolver cómo varía la suma ponderada de una capa en función de cómo varía la salida de una neurona de la capa anterior, que sería la propia matriz de parámetros W que conecta ambas capas.

$$\frac{\partial z_j^{L}}{\partial y_i^{L-1}} = w_{ki} \tag{2.24}$$

Por lo tanto, para la capa $L-1$, si se agrupan de nuevo los términos que hacen referencia al error asociado a la neurona de esta capa, quedaría:

$$\begin{cases} \frac{\partial C}{\partial w_{ki}} = \delta_j^L \cdot w_{ki} \cdot \sigma'\left(z_i^{L-1}\right) \cdot y_k^{L-2} = \delta_i^{L-1} \cdot y_k^{L-2} \\ \frac{\partial C}{\partial b_i^{L-1}} = \delta_j^L \cdot w_{ki} \cdot \sigma'\left(z_i^{L-1}\right) = \delta_i^{L-1} \end{cases} \tag{2.25}$$

Este conjunto de operaciones es lo que recibe el nombre de *backpropagation*, cuya aplicación reiterada permitirá obtener el gradiente sobre el que se medirá el error y que llevará al ajuste de los parámetros de la red hasta minimizarlo. Además permite calcular la regla de la cadena de una forma muy eficiente en las redes neuronales, lo que optimiza el tiempo de computación.

A modo de recapitulación, se resume en tres pasos los cálculos de este algoritmo:

1. Cálculo del error de la última capa asociado a la neurona:

$$\delta_j^L = \frac{\partial C}{\partial y_j^L} \cdot \frac{\partial y_j^L}{\partial z_j^L} \tag{2.26}$$

2. Retropropagación del error a la capa anterior:

$$\delta_i^{L-1} = w_{ij}^L \cdot \delta_j^L \cdot \frac{\partial y_i^{L-1}}{\partial z_i^{L-1}} \tag{2.27}$$

3. Cálculo de las derivadas de la capa mediante el error y las salidas de la capa previa:

$$\frac{\partial C}{\partial w_{ki}^{L-1}} = \delta_i^{L-1} y_k^{L-2} \quad \frac{\partial C}{\partial b_i^{L-1}} = \delta_i^{L-1} \tag{2.28}$$

Además del gradiente, existe otro valor fundamental para el entrenamiento de las redes multicapa. Este es el *Learning rate* o tasa de aprendizaje (η), que afecta a la actualización de los parámetros con la siguiente expresión [7]:

$$\Delta w_{ij} = -\eta \frac{\partial C}{\partial w_{ij}} \tag{2.29}$$

La tasa de aprendizaje define lo grande que se desea el salto entre los parámetros tras cada iteración, y determina la velocidad de convergencia del algoritmo. La elección de este valor es un factor crítico a la hora de aplicar con éxito el descenso del gradiente. Interesará que sea lo suficientemente pequeño para que el algoritmo converja a la solución

óptima, pero a su vez lo suficientemente grande para que lo haga lo más rápido posible. Por ello es habitual utilizar tasas adaptativas de forma que, como su propio nombre dice, se adapten a lo que requiere la situación de cada iteración.

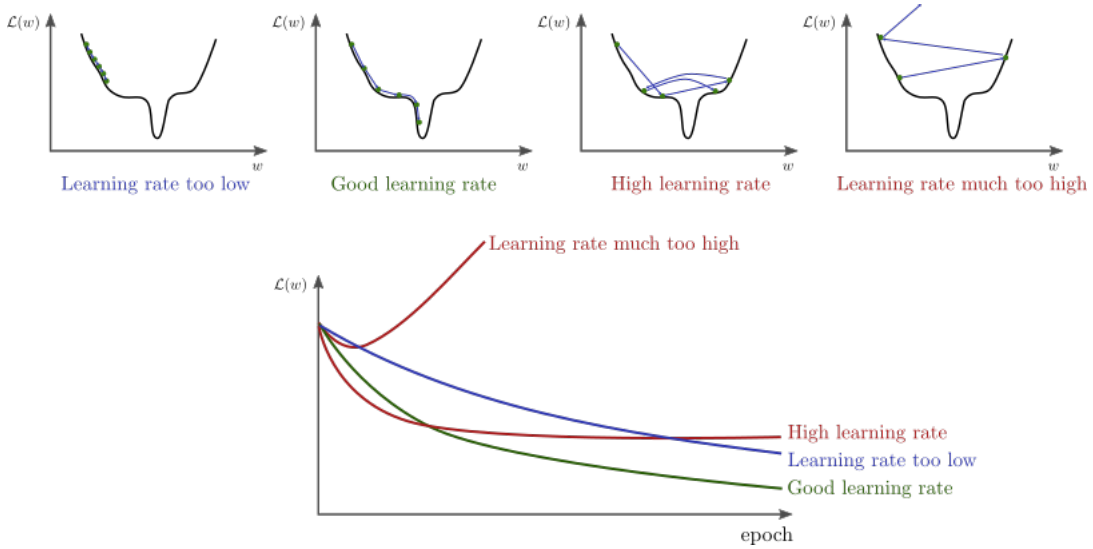


Figura 2.10: Aplicación del Descenso del gradiente para diferentes valores del *Learning rate* [10]

2.5.2. Cross-Entropy Loss

En la actualización de los pesos de la red mediante el cálculo del descenso del gradiente, pese a tratarse de un problema sencillo gracias a la aplicación del algoritmo *backpropagation*, que calcula la regla de la cadena de manera eficiente, existía un factor problemático asociado a la función de activación de la neurona: $\frac{\partial y_j}{\partial z_j}$ [7].

La salida de la neurona se corresponde con el resultado de la función de activación, por lo que esta derivada parcial, en el caso de una neurona sigmoidal, se corresponde con:

$$\begin{gathered} y_j = \sigma(z_j) = \frac{1}{1+e^{-z_j}} = (1+e^{-z_j})^{-1} \\ \frac{dy_j}{dz_j} = \frac{d\sigma(z_j)}{dz_j} = \frac{-1(e^{-z_j})}{(1+e^{-z_j})^2} = \left(\frac{1}{1+e^{-z_j}}\right)\left(\frac{e^{-jz}}{1+e^{-z_j}}\right) = -y_j(1-y_j) \end{gathered} \tag{2.30}$$

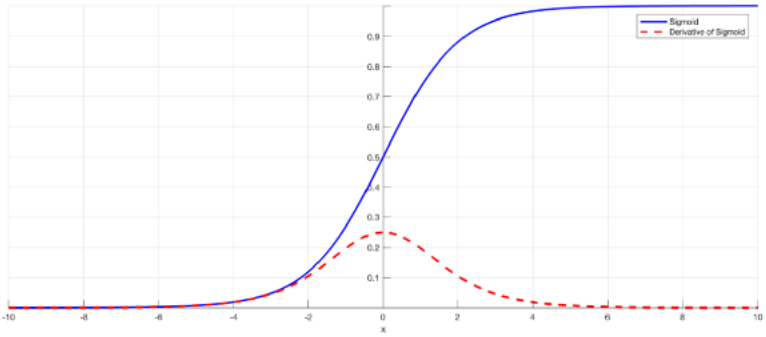


Figura 2.11: Representación de la función sigmoide y su derivada.

El problema surge cuando, a partir de cierto valor, la derivada de la función sigmoide se anula, como se ve en la Figura 2.11. Para conseguir suprimir este efecto, se incluye este factor en el denominador de la derivada de la función del error o coste, $\frac{\partial C}{\partial y_j}$. La nueva función de error, por lo tanto, quedaría así definida:

$$\frac{\partial CE}{\partial yj} = -\frac{t_j - y_j}{y_j(1-y_j)} \tag{2.31}$$

De manera que al incorporar esta nueva función en el cálculo de la actualización de los pesos, la expresión se simplificaría y el término de la derivada sigmoidal desaparecería:

$$\begin{aligned} \Delta w_{ij} &= -\eta \cdot \frac{\partial CE}{\partial y_j} \cdot \frac{\partial y_j}{\partial z_j} \cdot \frac{\partial z_j}{\partial w_{ij}} \\ &= -\eta \cdot [-(t_j - y_j)] \cdot x_i \\ &= \eta \cdot x_i \cdot (t_j - y_j) \end{aligned} \tag{2.32}$$

Siendo η la tasa de aprendizaje. Con esta modificación, los pesos ahora dependerán de la magnitud del error y de las entradas, fomentando una aceleración en el aprendizaje cuanto mayor sea el error cometido (tal y como hacen las personas). Al conocer su derivada, bastará con integrar la función para obtener la nueva función de coste:

$$\begin{aligned} CE &= \int f(y)dy = \int \frac{y-t}{y(1-y)}dy \\ &= \int \frac{y}{y(1-y)}dy - \int \frac{t}{y(1-y)}dy \\ &= \int \frac{1}{1-y}dy - t\int \frac{1}{y(1-y)}dy \\ &= \int \frac{1}{1-y}dy - t\int \left(\frac{1}{y} + \frac{1}{1-y}\right)dy \\ &= -\log(1-y) - t(\log y - \log(1-y)) \\ &= (t-1)\log(1-y) - t\log(y) \end{aligned} \tag{2.33}$$

Dicha función es a la que se denomina *Cross-entropy* (CE), entropía cruzada, y sustituye al error cuadrático en problemas de clasificación. Se suele expresar de la forma:

$$CE = -[t\log(y) + (1-t)\log(1-y)] \tag{2.34}$$

Donde t se corresponde con el valor objetivo *target* e y con la salida proporcionada por la neurona. Esta función se asemeja a un clasificador binario, y se puede generalizar para el caso de más de dos clases. La entropía cruzada de dos distribuciones de probabilidad p, correspondiente a la distribución de salida deseada partiendo de los datos observados y q, referente al modelo utilizada para realizar las predicciones se expresa:

$$H(p,q) = -\sum_x p(x)\log q(x) \tag{2.35}$$

La entropía, por tanto, devuelve la información que falta para poder predecir correctamente los valores asociados a ejemplos reales. La función $q(x)$ se corresponderá con la función *Softmax* que ha sido definida más arriba.

2.6. Redes Neuronales Convolucionales

Las Redes Neuronales Convolucionales, o *Convolutional Neural Network*(CNN), son las redes habitualmente utilizadas para el procesamiento de imágenes, imitando el comportamiento del cortex visual del ojo humano. Permiten identificar patrones y características, y con ello resolver problemas de clasificación de imágenes, detección de objetos... Hoy en día aparecen en un sinfín de aplicaciones, como en los detectores faciales que permiten desbloquear dispositivos móviles, o en el proceso de aprendizaje de los coches autónomos.

A diferencia de las redes multicapa, que recibían un vector de entrada de variables independientes, las redes convolucionales reciben vectores (1D), matrices (2D) o tensores (>2D) en los que existe una relación física entre sus componentes [7]. En el caso de señales bidimensionales, como son las imágenes, una red multicapa interpretaría la entrada como un vector plano en el que sus valores se corresponderían con los píxeles, perdiéndose la información de la posición de estos dentro de la imagen, que es de fundamental importancia. Es por esto que surgen las redes convolucionales [19].

El tamaño de la capa de entrada de la red viene definido por el número de píxeles de la imagen. Si por ejemplo, la imagen está en escala de grises y es de dimensiones 32×32 píxeles de alto y ancho, la capa de entrada estará formada por 1024 neuronas. Si además estuviera a color, habría que contar con los canales RGB (*Red, Green, Blue*), por lo que en este caso se constituiría por $32 \times 32 \times 3 = 3072$ neuronas. Los colores vienen dados por valores de 0 a 255, que se divirán entre 255 para trabajar con valores entre 0 y 1.

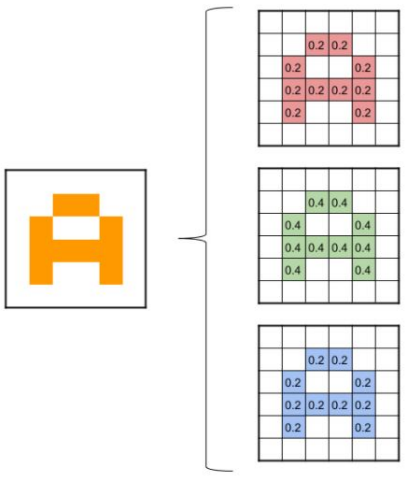


Figura 2.12: Descomposición de los píxeles de una imagen a color según los canales RGB [18].

La CNN se caracteriza por contar con un tipo de capa en el que se realiza una operación de convolución, la capa convolucional. Matemáticamente hablando, la convolución es una operación matemática sobre dos funciones de la que se obtiene una tercera función, interpretada como una versión modificada, o filtrada, de una de las funciones originales [7]. Así, la convolución de f y g se define como la integral del producto de ambas funciones

después de desplazar una de ellas una distancia t:

$$(f \star g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau = \int_{-\infty}^{\infty} f(t-\tau)g(\tau)d\tau \tag{2.36}$$

En el caso del procesamiento digital de las imágenes, al utilizar señales discretas, la definición de la operación quedaría tal que:

$$(f \star h)[x,y] = \sum_{k_1=0}^{K_1-1} \sum_{k_2=0}^{K_2-1} h\left[k_1,k_2\right] f\left[x+k_1, y+k_2\right] \tag{2.37}$$

Siendo f la señal que se desea procesar, $[x,y]$ las coordenadas de los píxeles de la imagen y h el filtro o *kernel* con el que se procesa la señal, de tamaño $K_1 \times K_2$.

En el procesamiento de las imágenes, estas convoluciones no son más que productos escalares de matrices. Los parámetros de esta capa consisten en un conjunto de filtros, cuyos valores dependerán de la función que se quiera realizar: detección de fronteras, suavizado de imágenes, efecto de grabados... Cada filtro es de pequeña dimensión en comparación con la imagen, y se extiende por toda la profundidad de la entrada como se explica a continuación.

El punto de partida comienza con la identificación de la imagen como matriz de entrada de la red, que este caso será en blanco y negro para facilitar la explicación, y con la selección del filtro de aplicación, cuyos valores se irán ajustando mediante *backpropagation* durante el entrenamiento de la red convolutiva (equivaldría al ajuste de los pesos en el caso de la red multicapa).

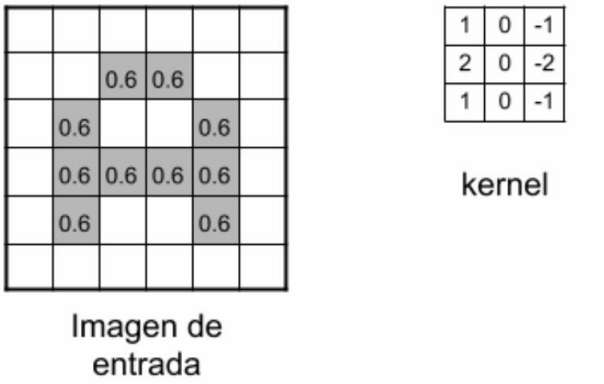


Figura 2.13: Imagen de entrada a la red y el *kernel* del proceso [18].

A continuación, se irá situando el filtro sobre las diferentes posiciones de la imagen, y por cada posición se realizará el producto matricial de la matriz de la imagen correspondiente y el filtro.

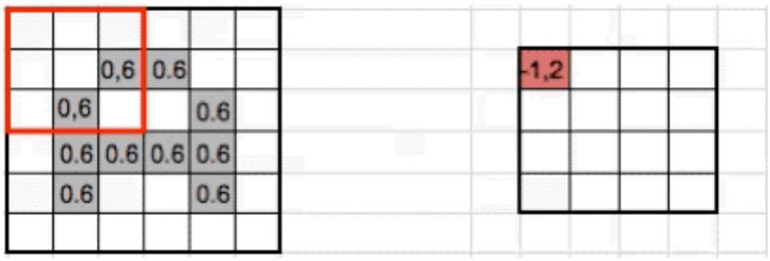


Figura 2.14: Producto matricial de los píxeles de la imagen correspondientes al tamaño del filtro con el filtro. El valor obtenido para este caso sería: $0 \times 1 + 0 \times 0 + 0 \times (-1) + 0 \times 2 + 0 \times 0 + 0{,}6 \times (-2) + 0 \times 1 + 0{,}6 \times 0 + 0 \times (-1) = -1{,}2$ [18].

Si se repite esta operación para todas las posiciones en las que es posible situar el *kernel*, se obtiene una nueva matriz correspondiente a una nueva imagen. Esta presenta un tamaño 4×4 cuando el de la original era 6×6. Si se deseara obtener una imagen del mismo tamaño bastaría con rodear los píxeles de la imagen con ceros, lo que se conoce como *zero padding*. Finalmente, pero no necesariamente, es habitual incluir una capa de tipo ReLU a continuación de la convolución, que proporciona el carácter no lineal a las imágenes obtenidas.

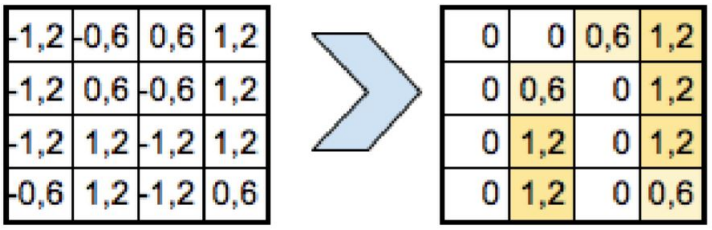


Figura 2.15: Aplicación de capa ReLU, que consiste en *f(x)=max(0,x)* [18].

Ahora bien, este es el cálculo que se realiza para cada filtro, pero cada capa no cuenta con un sólo filtro, si no que está formado por un conjunto de ellos que determina la profundidad de la capa convolutiva. El número de filtros K, hiperparámetro de la capa, determina el número de características que es capaz de detectar, lo que hace que también adopten los filtros el nombre de detectores. Cada matriz o imagen que devuelven los filtros se denomina mapa de características [*feature maps*], por lo que la salida de una capa capaz de detectar K características diferentes estará formada por K mapas de características, que serán K imágenes en las que cada fragmento de la imagen original se representa de K maneras diferentes.

Otra diferencia que existe con las redes multicapa es que, mientras que todas las neuronas de sus capas estaban conectadas a todas las entradas, en las capas convolutivas sólo se conectan a una zona local de la capa de entrada. Esta región está definida por el hiperparámetro F, el tamaño del filtro, siendo los valores más típicos 3×3 (F=3), 5×5 (F=5) y 7×7 (F=7) ($3 \times 3 \times 3$, $5 \times 5 \times 3$ y $7 \times 7 \times 3$ en el caso de imágenes de color, respectivamente).

Tras la operación de convolución, el número de neuronas de la próxima capa se multiplicaría por el número de mapas de características generado, lo que podría alcanzar

un número de neuronas desmesurado. Esto a su vez aumentaría el número de parámetros a calcular en el procesamiento de las imágenes, que conllevaría un elevado coste computacional. Para tratar de reducirlo se emplean las capas de *pooling* o submuestreo [*subsampling/downsampling*]. El objetivo de estas capas es reducir el tamaño de las imágenes ya filtradas, para que las capas posteriores de convolución puedan operar con datos de entrada reducidos. La aplicación que más se utiliza es la de *Max Pooling*, que combina los píxeles de entrada con la función máximo. Volviendo al caso anterior, para un *max pooling* 2×2, se escogerían fragmentos de la imagen de esa dimensión y se guardaría el valor máximo de cada uno.

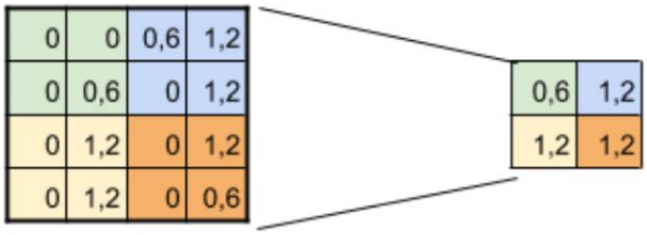


Figura 2.16: Aplicación de una capa *max pooling* 2×2 [18]

Este procedimiento *Convolución - ReLU - Pooling* se repetirá sucesivas veces, hasta obtener un tamaño pequeño de la dimensión de la imagen. Será entonces cuando pasará a una red multicapa que se encargará del problema de clasificación .

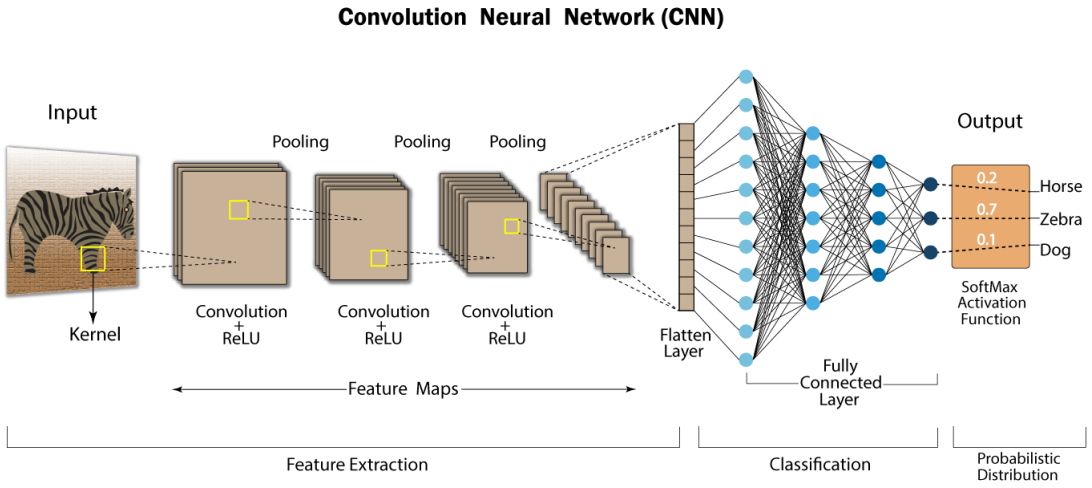


Figura 2.17: Esquema de las capas de CNN. La imagen de entrada pasará por fases reiterativas de *Convolución - ReLU - Pooling* en la que se entrenará a la red con la extracción de patrones de la imagen original. Posteriormente, una red multicapa convencional resolverá el problema de clasificación.

Capítulo 3

Ataques adversarios

3.1. Concepto

Pese a que en los últimos años se ha observado un desarrollo progresivo en los modelos de *Deep Learning* hasta alcanzar un nivel de desempeño que iguala o hasta supera las capacidades del ser humano, estos no dejan de ser vulnerables a ciertos ataques.

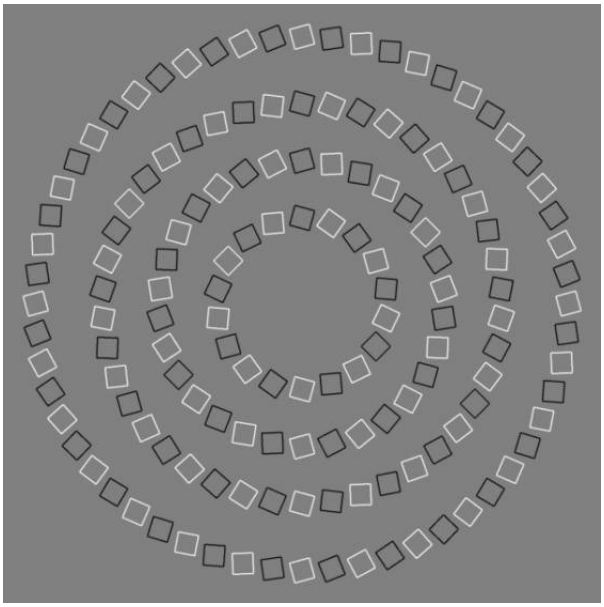


Figura 3.1: Ejemplo de ilusión óptica. La disposición y el color de los cuadrados hace pensar que los círculos se cruzan, cuando en realidad estos son concéntricos. Fue creada por el psicólogo italiano Baingio Pinna en 2002.

La Figura 3.1, además de resultar hipnotizante, es capaz de confundir al cerebro y hacerle percibir una idea errónea de la realidad. Se trata de una ilusión óptica. Son percepciones visuales que no se ajustan a la realidad del mundo que nos rodea, y ponen en manifiesto que la información que interpreta y reelabora el cerebro puede ser en ocasiones defectuosa. Esta manera en la que se engaña al cerebro humano es lo que, para el caso de *Machine Learning* y *Deep Learning*, se conoce como Ataque Adversario.

Un ataque adversario se define como una manipulación aplicada a cualquier tipo de dato de entrada que se suministra a la red neuronal caracterizada por ser imperceptible al ojo humano y que, sin embargo, el modelo percibirá como un dato totalmente diferente, llevándole a cometer un error en su clasificación [23].

Un ejemplo con el que es sencillo entender esta manipulación en con los ataques a imágenes digitales. Una vez entrenada la red neuronal, convolucional en este caso al trabajar con imágenes, se toma una imagen a la que se le añade una perturbación, un ruido aleatorio inapreciable para el ojo humano como el que se presenta en la Figura 3.2. A la izquierda, se observa la imagen original, calificada por la red como gato con una precisión del 87 %. La imagen de la derecha sin embargo, pese a parecer exactamente la misma que la anterior, el modelo la califica como una tostadora con una precisión del 98 %.

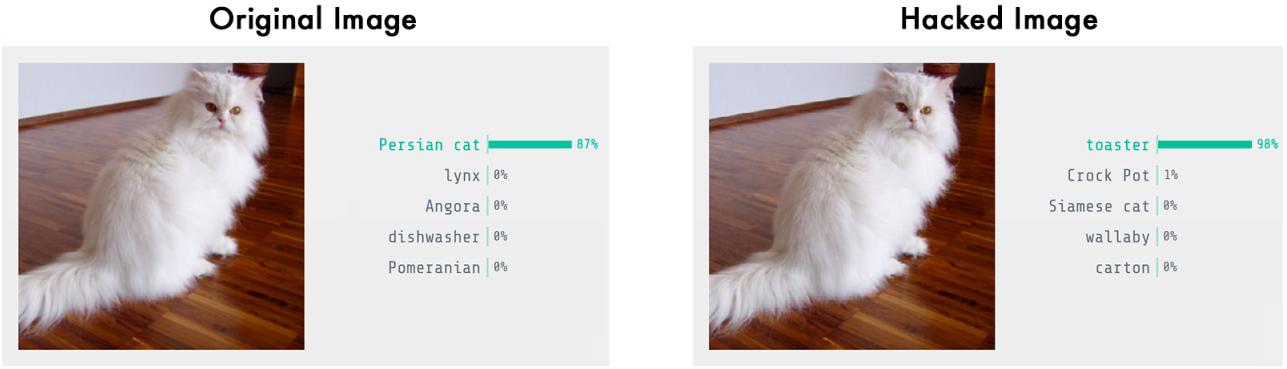


Figura 3.2: Aplicación de un ataque adversario a una imagen digital. A la izquierda, imagen original de un gato, y a la derecha, misma imagen con una pequeña perturbación añadida [24]

Estos ataques no sólo pueden aplicarse a imágenes digitales, sino que también a imágenes del mundo real. Unas ligeras modificaciones en señales de tráfico, tal y como se muestra en la Figura 3.3, podría ocasionar una mala interpretación por parte de un coche autónomo, y que la tradujera como una señal de límite de velocidad o, directamente, ni la detectara. Esto supondría un grave problema de cara a la adopción de los coches autónomos.

Figura 3.3: Aplicación de un ataque adversario a una señal de STOP que, tan solo añadiendo cintas, podría ser interpretada como una señal de límite de velocidad por un coche autónomo [22].

Además, por si no fuera poco, todos estos ataques comparten una propiedad: la transferibilidad. Esta permite que, una vez diseñado un ataque adversario para una red en particular, sea muy probable que permita engañar a otras redes de mayor o menor complejidad.

La pregunta entonces es, *¿se pueden evitar de alguna forma estos ataques?*.

3.2. Posibles estrategias para combatir los ataques adversarios

En los últimos años, el efecto de estas manipulaciones ha despertado mucho interés en la comunidad del *Machine Learning*. La búsqueda de medios que permitieran evitar o prevenir los ataques adversarios, ha derivado en dos posibles estrategias, aunque no muy efectivas [20].

- **Estrategias proactivas**: Su objetivo es incrementar la robustez del modelo. Para ello, la red se entrenará con imágenes originales y con imágenes perturbadas, de manera que sea capaz de identificar estas últimas como falsas. Sin embargo, debido a la transferibilidad que caracteriza a los ataques, tarde o temprano la red acabaría siendo engañada.
- **Estrategias reactivas**: Su objetivo es hacer más difícil el ataque. Para ello, la red ocultará detalles del modelo, como no mostrar el vector de probabilidades para cada categoría que se obtiene de la última capa de la red e indicar sólo la categoría más probable. La falta de información dificultaría estimar si el ataque modifica o no la distribución. Sin embargo, este método tampoco resulta de total eficacia y tarde o temprano la manipulación se efectuaría.

Otra idea interesante y más efectiva es la que se plantea en el artículo *'Adversarial Examples Are Not Bugs, They Are Features'* [21]. Los autores diferencian dos tipos de características atribuidas a las redes, las *robustas*, las más intuitivas que aportan mayor información a simple vista, y las *no robustas*, detalles más susceptibles difíciles de percibir pero que contienen información esencial para la clasificación que hace la red. Los resultados obtenidos tras la investigación demuestran que los modelos entrenados con set de imágenes robustos tienen una precisión alta ante los ejemplos adversarios, al contrario que los del set no robusto. Esto les permite concluir que los ataques adversarios tienden a afectar a las características no robustas de los datos, por lo que eliminar la dependencia de estas características permitirá obtener modelos menos susceptibles a ser atacados.

3.3. Programación de un ataque adversario

¿Cómo se implementa un ataque adversario? Quizás uno puede esperar que para crear un algoritmo que permita engañar a una red neuronal profunda entrenada con miles de datos haya que ser un *hacker* profesional. Sin embargo, basta con tener una base de cómo funcionan y se entrenan las redes y soltura en el manejo de Python para poder hacerlo. A continuación, previamente a la programación, se introducirá el fundamento matemático de los ataques [25].

En primer lugar se define un modelo $h_\theta : \mathcal{X} \to \mathbb{R}^k$, en el que para una entrada $\mathcal{X}$, correspondiente en este caso a un tensor tridimensional, se obtiene un vector de K dimensiones, siendo K el número de clases que predice la red. El vector θ simboliza el conjunto de parámetros que caracterizan al modelo y que se optimizan durante el entrenamiento (los *kernel* de las capas convolucionales, los pesos de la red multicapa, etc). La salida de la red vendrá definida por lo tanto por $y_j = h_\theta(x_i)$.

La diferencia entre el resultado devuelto por el modelo y la salida esperada da información sobre el error de la predicción, que define la función de pérdida o coste C. Por lo tanto, el primer argumento hace referencia a la salida del modelo y el segundo al índice (entre 1 y k) correspondiente a la clase correcta:

$$C(h_\theta(x), t) \tag{3.1}$$

Esta función devuelve la pérdida que obtiene el clasificador al predecir la clase de la entrada $x \in \mathcal{X}$ siendo $t \in \mathbb{Z}$ la clase verdadera, e interesará que sea mínima. La expresión más utilizada para calcular la pérdida es la entropía cruzada, también llamada pérdida *softmax*, debido a la función de la que deriva. La función *Softmax* se definía:

$$y_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \tag{3.2}$$

Su función era transformar los valores devueltos por el modelo en una distribución de probabilidad, de manera que la suma de todos diera 1. El objetivo del problema de clasificación será maximizar la probabilidad correspondiente a la clase verdadera. Una alternativa muy utilizada es maximizar el logaritmo de la probabilidad, por lo que la expresión quedaría dada por:

$$\log\left(\frac{\exp(h_\theta(x)_t)}{\sum_{k=1}^{K}\exp(h_\theta(x)_k)}\right) = h_\theta(x)_t - \log\left(\sum_{k=1}^{K}\exp(h_\theta(x)_k)\right) \tag{3.3}$$

Como el objetivo del entrenamiento será, por lo tanto, minimizar la pérdida, bastará con utilizar la negación (al tratarse de una función logarítmica) de la expresión anterior para tener la función de pérdida final:

$$C\left(h_\theta(x), t\right) = \log\left(\sum_{k=1}^{K}\exp\left(h_\theta(x)_k\right)\right) - h_\theta(x)_t \tag{3.4}$$

Hasta aquí sería el resumen del fundamento matemático de cómo minimizar el error del entrenamiento de la red, que como ya se vio en apartados anteriores, consistía en un proceso reiterativo en el que se aplicaba el descenso del gradiente para actualizar los pesos mediante *backpropagation* para conseguir la correcta clasificación de la imagen de entrada. Pero, *¿y esto qué tiene que ver con los ataques adversarios?*

El ataque adversario trata de modificar una imagen inicial para conseguir *engañar* un modelo ya entrenado. La clave de estos ataques es que, mientras que para el entrenamiento de la red se calculaba el gradiente con respecto a los parámetros de la red (*¿Cuánto*

habría que modificar los parámetros del modelo para apreciar un cambio en la función de coste?), en este caso se calculará el gradiente con respecto a la propia entrada (*¿Cuánto habría que modificar la entrada para apreciar un cambio en la función de coste?*).

Por lo tanto, para crear un ejemplo que contradiga los resultados esperados de la red, habrá que ajustar la imagen de manera que se maximice la pérdida, en lugar de minimizarla, pues la finalidad del ataque es que la red lo clasifique como cualquier clase, excepto como la verdadera.

$$\max_{\hat{x}} C\left(h_\theta(\hat{x}), t\right) \tag{3.5}$$

Donde $\hat{x}$ es el ejemplo adversario. Lo interesante de los ejemplos de ataques que se han mencionado a lo largo del trabajo es que para el ojo u oído humano estos cambios eran imperceptibles: el ejemplo adversario $\hat{x}$ debe estar cerca de la entrada inicial, x. Esto se consigue optimizando el *ruido* que se aplica sobre x, al que se designará la nomenclatura δ. Para asegurar que esta *perturbación* sea pequeña, se acotará de manera que no supere una magnitud entre [-ϵ, ϵ]. El problema final de optimización a seguir entonces será:

$$\max_{\delta \in \Delta} C\left(h_\theta(x+\delta), t\right) \tag{3.6}$$

Donde Δ representa el conjunto válido de perturbaciones.

La puesta en práctica es sencilla (Anexo Código A.1). En primer lugar, se escoge un modelo previamente entrenado para el que se diseñará el ataque adversario. En este proyecto se ha optado por el modelo ResNet50, que está entrenado con 50.000 imágenes, y se ha trabajado con la librería PyTorch. La imagen escogida para *engañar* a la red ha sido la de un cerdo:

Figura 3.4: Imagen original sobre la que se aplicará el ataque adversario.

En primer lugar, antes de cualquier modificación, se ha comprobado que la red clasifica la imagen correctamente con alta fiabilidad.

```
Predicted class:  hog
Predicted probability: 0.9961252808570862
```

Figura 3.5: Predicción del modelo ResNet50 al recibir como entrada la imagen original del cerdo.

Posteriormente, se aplica la perturbación a la imagen y se obtendrá una imagen aparentemente similar pero que, sin embargo, la red no clasificará como un cerdo.

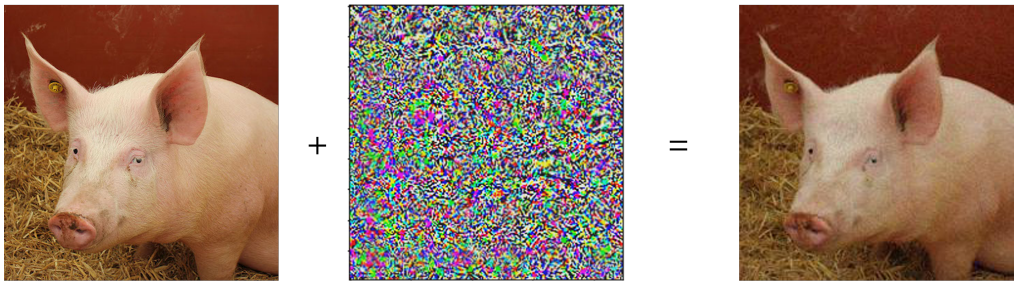


Figura 3.6: Imagen que resulta tras aplicar el ruido a la imagen original y que, aparentando ser la misma imagen, la red identifica como un *wombat*.

```
True class probability: 0.00031623418908566236
Predicted class:  wombat
Predicted probability: 0.9989402890205383
```

Figura 3.7: Predicción del modelo ResNet50 al recibir como entrada el ejemplo adversario.

Para los lectores que no sepan qué es un *wombat*, se trata de este animal tan adorable:

Figura 3.8: Imagen de un wombat.

Otro técnica que se podría utilizar es la de los ataques dirigidos (Anexo Código A.2). En el caso anterior, el procedimiento maximizaba la pérdida de la clase verdadera para evitar que la red la identificara como la correcta y eligiera la que tuviera mayor probabilidad, siendo una clase aleatoria. Los ataques dirigidos permiten, además, elegir la clase a la que se desea *convertir* la imagen original (t_{ataque}). Por lo tanto, al mismo tiempo que se maximiza la pérdida de la clase correcta, se minimiza la de la clase objetivo. El nuevo problema de optimización quedaría definido como:

$$\max_{\delta \in \Delta} \left(C\left(h_\theta(x+\delta), t\right) - C\left(h_\theta(x+\delta), t_{\text{ataque}}\right) \right) \tag{3.7}$$

En el ejemplo que se ha implementado, la clase elegida como objetivo del ataque ha sido la correspondiente a una ambulancia:

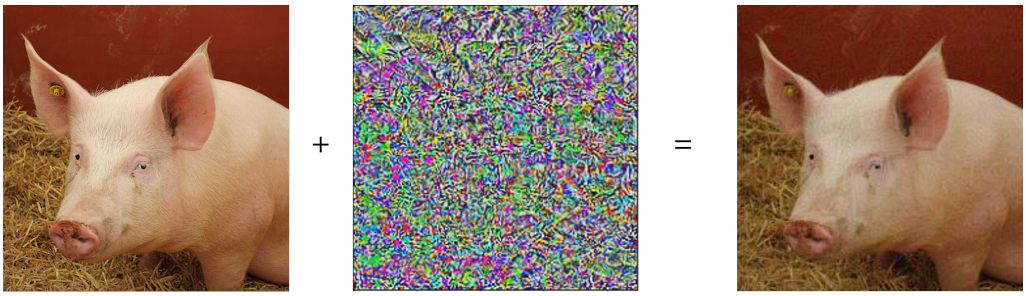


Figura 3.9: Imagen que resulta tras aplicar el ataque adversario dirigido a la imagen original y que, aparentando ser la misma imagen, la red identifica como una ambulancia.

```
True class probability: 9.00517066606899e-24
Predicted class:  ambulance
Predicted probability: 1.0
```

Figura 3.10: Predicción del modelo ResNet50 al recibir como entrada el ejemplo adversario dirigido.

Con el desarrollo de estos dos tipos de ataques concluye este capítulo, demostrando lo fácil que resulta *burlar* las predicciones de las redes neuronales. Hasta entonces, las estrategias que se han estudiado para su defensa han sido las descritas en el apartado previo, las estrategias proactivas y reactivas. En el siguiente capítulo se desarrollará la metodología que se ha investigado y ha dado sentido a este proyecto, la prevención de ataques mediante la aplicación de algoritmos de compresión.

Capítulo 4

Prevención de ataques adversarios mediante algoritmos de compresión

4.1. Análisis de Componentes Principales

Principal Component Analysis (PCA) o el Análisis de Componentes Principales es un método estadístico correspondiente al campo del aprendizaje no supervisado [*unsupervised learning*] que permite transformar espacios de dimensión p a otra más pequeña k (k<p) a la vez que conserva la mayor información posible [26]. En otras palabras, de una muestra de p variables, PCA permite encontrar k variables que explican prácticamente lo mismo que las p variables originales. Estas k variables son las que dan nombre a este método: las componentes principales.

A diferencia del entrenamiento de las redes que se ha estudiado en los apartados anteriores, un método de *supervised learning* en el que el modelo predecía una respuesta Y, en *unsupervised learning* el objetivo es aprender de los propios datos, descubrir patrones de los que se pueda extraer información que permita, por ejemplo, identificar características comunes para formar subgrupos o crear reglas de asociación.

4.1.1. Cálculo de las componentes principales

El método PCA está muy relacionado con el álgebra lineal. Cada una de las componentes está asociada a un vector propio [*eigenvector*], y están ordenadas según el orden decreciente establecido por sus autovalores [*eigenvalue*] correspondientes.

La interpretación geométrica del proceso ayuda al entendimiento de PCA de una manera más intuitiva. Se parte de un conjunto de muestras definididos por dos variables (X_1, X_2). La primera componente principal, Z_1, estará asociada al vector definido según la dirección de máxima varianza de los datos (línea roja). La proyección de cada punto sobre dicho vector equivale a la primera componente principal para esa muestra (z_{i1}. La segunda componente, Z_2 seguirá la segunda dirección en la que los datos presentan una mayor varianza y que además no esté correlacionada con la primera, o lo que es lo mismo, que sean ortogonales (línea verde).

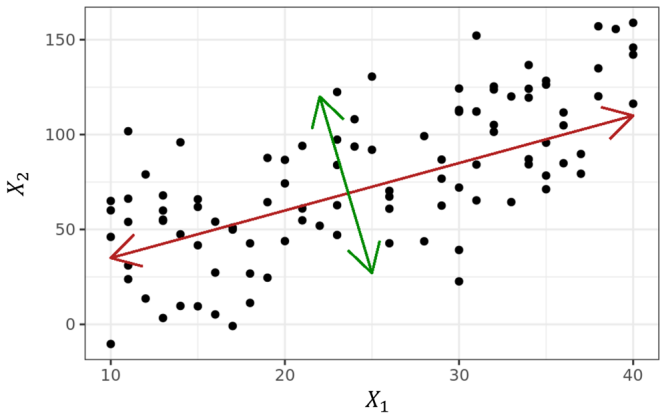


Figura 4.1: Interpretación geométrica de las componentes principales [26].

Cada componente principal Z_i se calcula a partir de la combinación linea de las variables originales. La primera componente principal, Z_1, correspondiente a la dirección de mayor varianza,

$$Z_1 = \phi_{11}X_1 + \phi_{21}X_2 + \ldots + \phi_{p1}X_p \tag{4.1}$$

se caracteriza por estar normalizada. Esto quiere decir que los coeficientes $\phi_{11}, ..., \phi_{1p}$, llamados *loadings* y que vienen a ser unos pesos que definen la importancia que tiene cada variable en la independiente, cumplen:

$$\sum_{j=1}^{p} \phi_{j1}^2 = 1 \tag{4.2}$$

El cálculo de la primera componente principal de un conjunto de datos X con n observaciones y p variables consiste en:

1. Centralización de las variables: restar a cada valor la media de la variable a la que pertenece.

2. Maximización de la varianza: resolver un problema de optimización para encontrar los *loadings* que la maximizan. Para ello es habitual obtener los vectores y valores propios de la matriz de covarianzas.

El proceso será el mismo para la segunda componente, Z_2, teniendo en cuenta la condición de que debe ser perpendicular a Z_1. Este se repetirá iterativamente hasta calcular todas las posibles componentes.

El PCA transformaba espacios de dimensión p a otra más pequeña k (k<p) a la vez que conserva su información. Para conocer cuánta información es capaz de captura el método, pues parece evidente que al reducir la dimensión parte de esta se perderá, se recurre a la proporción de varianza explicada por cada componente principal. Esta viene definida por el ratio entre la varianza explicada y la varianza total.

- Varianza explicada por la componente m:

$$\frac{1}{n}\sum_{i=1}^{n} z_{im}^2 = \frac{1}{n}\sum_{i=1}^{n}\left(\sum_{j=1}^{p}\phi_{jm}x_{ij}\right)^2 \tag{4.3}$$

- Varianza total:

$$\sum_{j=1}^{p} \text{Var}\left(X_j\right) = \sum_{j=1}^{p} \frac{1}{n} \sum_{i=1}^{n} x_{ij}^2 \tag{4.4}$$

Por lo que la proporción de varianza explicada por la componente m seguirá la expresión:

$$\frac{\sum_{i=1}^{n} \left(\sum_{j=1}^{p} \phi_{jm} x_{ij}\right)^2}{\sum_{j=1}^{p} \sum_{i=1}^{n} x_{ij}^2} \tag{4.5}$$

El valor de la proporción de varianza explicada acumulada será el que permita conocer qué número de componentes principales sera el óptimo de calcular. Interesará reducir la dimensionalidad utilizando el mínimo número de componentes suficientes que permitan almacenar la información de los datos. Este número se corresponderá con el límite a partir del cual el incremento de la proporción de la varianza explicada acumulada, como se representa en la siguiente gráfica, deja de ser fundamental.

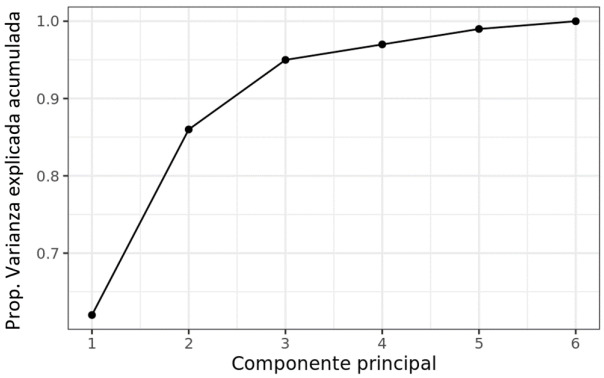


Figura 4.2: Relación entre la proporción de varianza explicada acumulada y el número de componentes principales [26]

4.1.2. Aplicación

Antes de comenzar este capítulo, el camino de este TFG parecía encaminado con temas de *Deep Learning* y de redes neuronales multicapa y convolucionales, por el interés por la vulnerabilidad de los modelos ante ataques adversarios... *¿Qué relación tiene todo esto con el Análisis de Componentes Principales?*

Recapitulando lo que ya se ha visto de ataques adversarios, estos consisten en imágenes a las que se les ha añadido una perturbación tan pequeña que consiguen *engañar* a la red, que las clasifica como una clase incorrecta, siendo el cambio inapreciable para el ojo humano. Por otro lado, recapitulando el método PCA, este consigue reducir un conjunto de datos a una menor dimensión a la vez que guarda prácticamente la información original... y puede utilizarse para comprimir imágenes. *¿Se ve ahora alguna relación?*

El objetivo de recurrir al método PCA es el siguiente: comprimir el ataque adversario, de manera que se modifique el *ruido*, y que, tras enseñar la nueva imagen a la red neuronal, esta la clasifique correctamente. El objetivo es, en otras palabras, prevenir los ataques

adversarios.

Para comprobar si el Análisis de Componentes Principales consigue excluir los ejemplos adversarios, se retomará el ejemplo con el que se trabajó en el capítulo anterior:

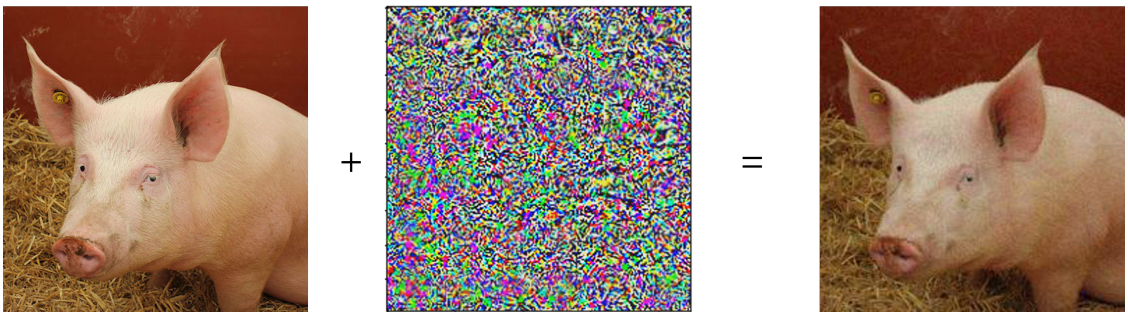


Figura 4.3: Imagen que resulta tras aplicar el ruido a la imagen original y que, aparentando ser la misma imagen, la red identifica como un *wombat*.

Según el número componentes principales utilizados, la imagen comprimida presentará en menor o en mayor medida la información y apariencia de la imagen original. Para diferentes componentes, la imagen clasificada como *wombat* quedará así:

Figura 4.4: Imagen comprimida reconstruida con un número creciente de componentes principales [27].

El siguiente paso será escoger el número de componentes principales óptimo. Partiendo de tres conjuntos de diferente número de componentes principales que alcanzan más del 90 % de la varianza acumulada original, se analizan qué factores será necesario tener en cuenta y la consideración sobre cada uno de ellos:

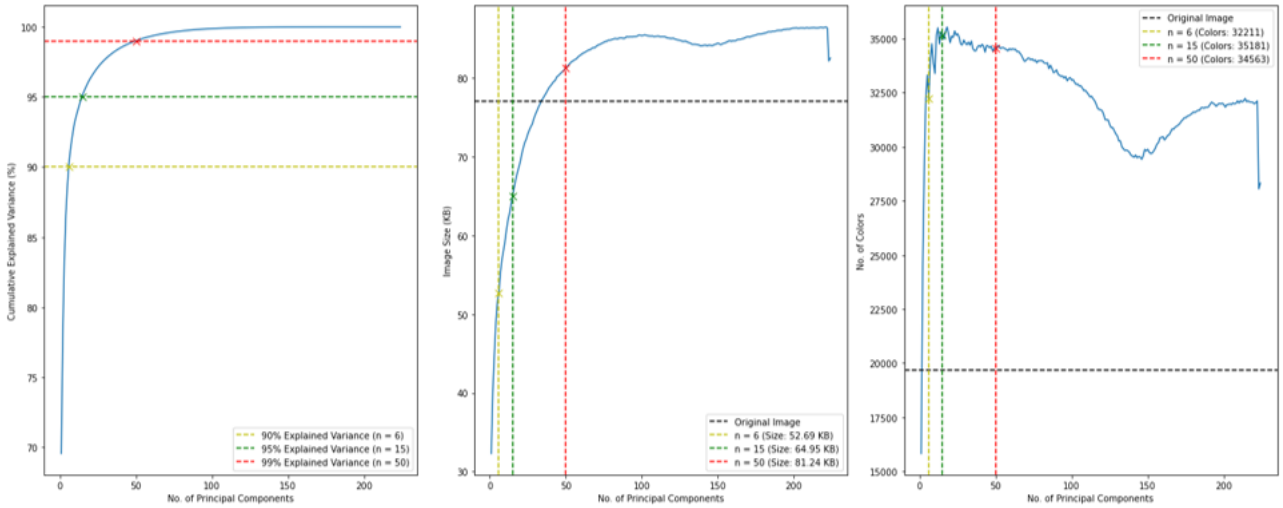


Figura 4.5: Factores a tener en cuenta para escoger el número óptimo de componentes principales para la compresión de la imagen, de izquierda a derecha: varianza explicada acumulada (%), tamaño de la imagen (KB) y número de colores [27].

- **Varianza explicada acumulada (Gráfica de la izquierda de la Figura 4.5)**: Para alcanzar un 90 %, 95 % y 99 % de la varianza explicada de la imagen original, se necesitarán 6, 15 o 50 componentes principales, respectivamente.
- **Tamaño de la imagen (Gráfica central de la Figura 4.5)**: Tras comparar esta gráfica con la anterior, se deduce que cuanto mayor es la varianza explicada acumulada, mayor es el tamaño de la imagen comprimida. La línea discontinua negra indica el tamaño de la imagen original, por lo que interesará escoger una opción que esté por debajo de ella: es el caso de 6 o 15 componentes principales.
- **Número de colores (Gráfica de la derecha de la Figura 4.5)**: A diferencia de la varianza explicada, el número de colores no varía monótonamente en función del número de componentes.

Al no existir una diferencia notable entre el número de colores que consiguen los conjuntos de 6 o 15 componentes, se escogerá el conjunto de 15, ya que permite reducir el tamaño de la imagen conservando un 95 % de la varianza explicada acumulada. Por lo tanto, la para este número de componentes, la imagen comprimida, en comparación con la original, quedaría:

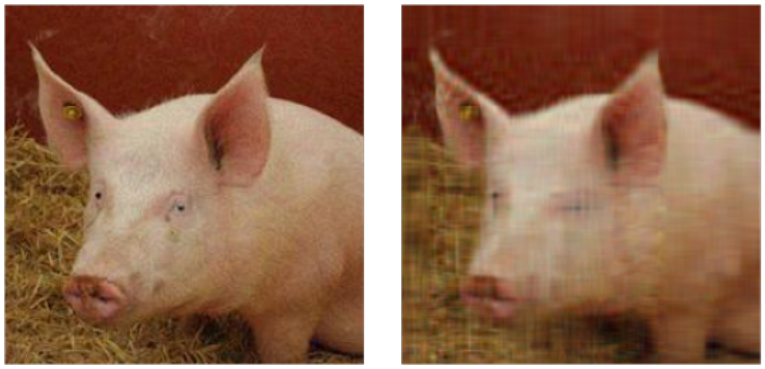


Figura 4.6: Imagen correspondiente al ejemplo adversario antes y después de ser comprimida, de izquierda a derecha.

Las características de cada imagen se recogen en la siguiente tabla:

Tipo de imagen	Nº colores	Tamaño imagen (kB)	Varianza Explicada Acumulada (%)
Original	19668	77.000977	100
PCA-Reducida	35181	64.946289	95.14994

Tabla 4.1: Características de la imagen antes y después de ser comprimida mediante PCA.

Una vez se ha reducido el tamaño y la resolución del ejemplo adversario, se procesará esta imagen de nuevo mediante el modelo ResNet50, para comprobar si esta compresión ha logrado filtrar el *ruido* que se introduce al ataque.

```
Predicted class:  hog
Predicted probability: 0.7525399923324585
```

Figura 4.7: Predicción del modelo ResNet50 al recibir como entrada el ejemplo adversario tras aplicarle el PCA.

Satisfactoriamente se obtiene lo esperado. Filtrar la imagen modificada mediante el método de Análisis de Componentes Principales ha permitido destruir el componente de ruido imperceptible que confundía a la red y hacía que la asociara a una clase distinta a la verdadera.

4.2. Estudio teórico de la vulnerabilidad ante ataques adversarios en función de la dispersión de los datos en un modelo que utiliza datos sintéticos

En este apartado, en lugar de trabajar con imágenes, se utilizará un conjunto de 600 datos sintéticos distribuidos en forma de espiral. Estos datos son más fáciles de modificar para entender en detalle cómo funcionan los ataques adversarios y cómo comprimir la imagen puede ayudar a prevenirlos. Cada una de las ramas, constituida por 200 puntos, se corresponderá con una clase diferente:

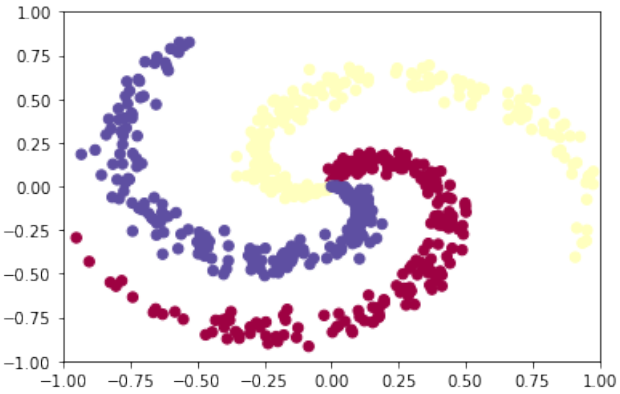


Figura 4.8: Conjunto de datos sintéticos con el que se estudiará la vulnerabilidad de la red ante ataques adversarios. Se distinguen tres clases: Clase '0' (rama roja), clase '1' (rama amarilla) y clase '2' (rama azul) [28].

Ahora bien, *¿cómo se aplicaría un ataque adversario a este conjunto de datos?* En apartados anteriores, se vio que para modificar una imagen bastaba con añadir una perturbación muy pequeña a los datos originales. Esta perturbación sería la analogía de mover uno de los puntos de la espiral en una dirección aleatoria tal que este se desplace a una región en la que el modelo, previamente entrenado, lo interpretaría como una clase diferente a la que pertenece.

En primer lugar, se estudiará el caso en el que los puntos estén distribuidos en el espacio 2D, como es el caso de la Figura 4.8, y posteriormente se convertirá al espacio 3D para diferentes valores de dimensión en el eje z. Lo que se espera obtener de esta investigación es que, cuantas más dimensiones de ruido se introduzcan, o en otras palabras, cuanta más dispersión exista entre los datos, más fácil será hacer un ataque adversario. Estas dimensiones extra también están presentes en las imágenes aunque no son tan fáciles de separar de las dimensiones relevantes como en este caso, donde los datos sintéticos están construidos para facilitar esto.

A la hora de realizar los ataques, se ha escogido un criterio común para todos los casos sobre cuándo detener el desplazamiento del punto que está siendo modificado. Como se puede intuir, una mayor dispersión de los datos dificultará el aprendizaje de la red y hará que la precisión del clasificador sea peor. La propuesta inicial consistía considerar el final de un ataque cuando la red clasificara un punto como una clase incorrecta con una probabilidad mayor del 90 %. Sin embargo, para algunos casos en los que la dispersión

de los puntos es muy grande, la red no logra clasificar los puntos con una precisión tan alta, por lo que el criterio que se ha optado es el siguiente: detener el desplazamiento del punto y dar por terminado el ataque adversario cuando la red adjudica al punto una probabilidad menor del 33 % de pertenecer a la clase verdadera.

4.2.1. Espiral en el espacio 2D

Para el caso bidimensional, los datos sintéticos que se utilizarán serán los representados en la espiral tricolor de la Figura 4.8. El objetivo del entrenamiento del modelo diseñado para esta entrada de datos es que sea capaz de reconocer al espacio correspondiente a a cada clase. El código utilizado puede encontrarse en el (Anexo Código A.3).

Debido a la disposición de los datos, un clasificador lineal no sería capaz de *aprender* a reconocer las regiones del espacio que, intuitivamente, se espera que correspondieran a cada clase. Su mejor predicción sería la que se muestra en la Figura 4.9.

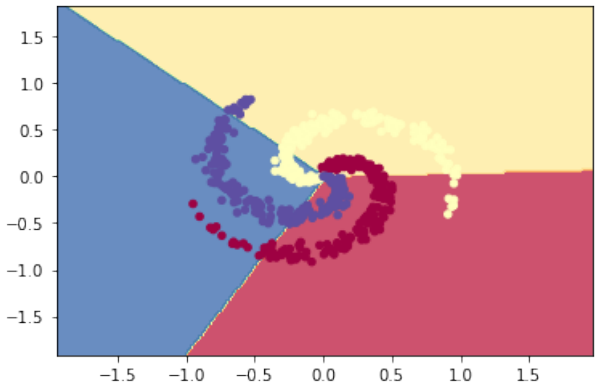


Figura 4.9: Clasificador lineal: las regiones coloreadas son las que, según los datos de entrada, el clasificador es capaz de asociar a cada una de las clases , con el color correspondiente [28].

Para obtener un clasificador que mejore su precisión adaptándose a datos que inicialmente no son fácilmente separables linealmente, se recurre a las redes neuronales. Un modelo cuyos parámetros se actualicen para minimizar la pérdida, permitirá que el clasificador se adapte mejor a la distribución de los puntos de la espiral. La precisión de la clasificación será mejor cuanto mayor sea el número de iteraciones del entrenamiento.

Se define una red neuronal entrenada durante 1500 épocas (iteraciones), con una tasa de aprendizaje de 0.1 y una dimensión de las capas intermedias constituida por 2000 neuronas. Una vez entrenado el modelo, se calcula su pérdida (Figura 4.10) y su precisión (Figura 4.11) para comprobar *cuánto de bien* ha aprendido de los datos. Las no linealidades presentes en la distribución de los datos se superarán mediante las redes neuronales, y lograrán clasificar los datos de la espiral con mayor exactitud (Figura 4.12) .

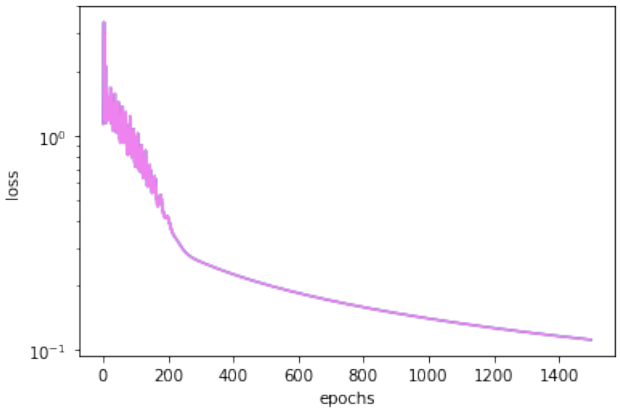


Figura 4.10: Cálculo de la pérdida mediante *Cross-Entropy*. Se reduce con el número de épocas, lo que significa que el error en la predicción se va reduciendo con la actualización de los parámetros internos de la red.

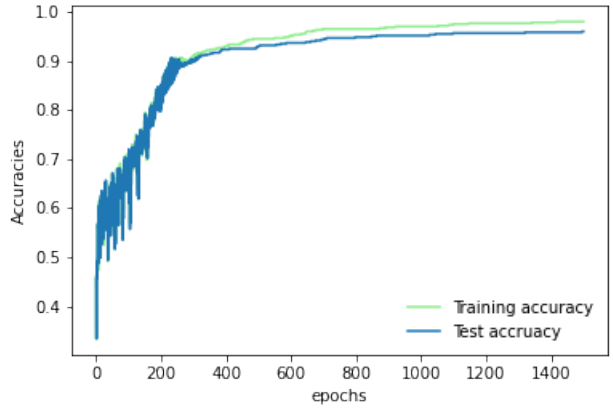


Figura 4.11: Cálculo de la precisión del aprendizaje. Al contrario que la pérdida, esta aumenta con el número de épocas de entrenamiento. *Training accuracy* hace referencia a la precisión que alcanza el modelo durante en el entrenamiento, mientas que *Test accuracy* se corresponde con la precisión obtenida tras introducir como entrada datos diferentes que la red no ha *visto* nunca antes.

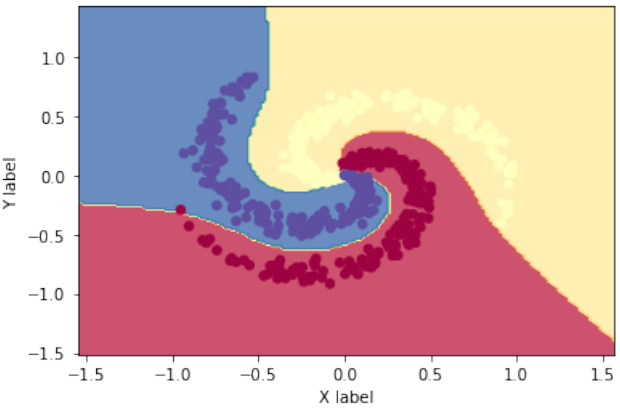


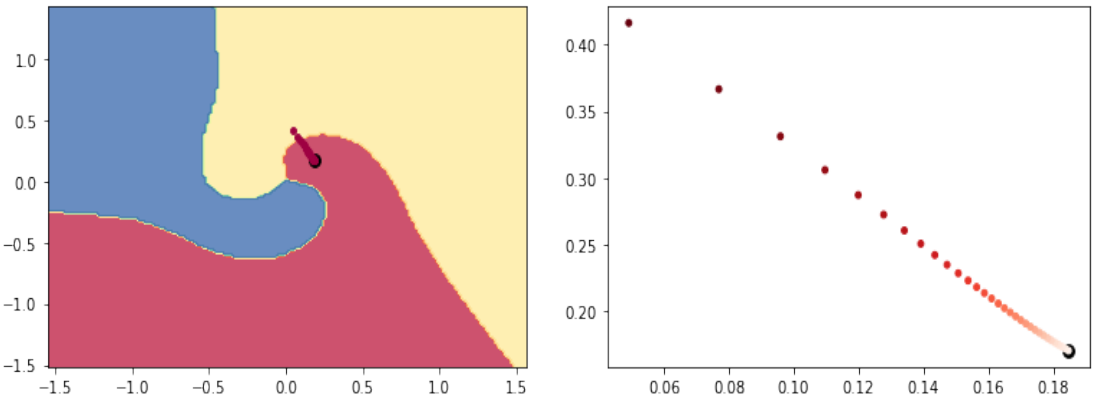
Figura 4.12: Clasificador obtenido tras la aplicación de las redes neuronales [28].

4.2.1.1. Aplicación de ataque adversario en el espacio bidimensional

La dispersión de los datos de la espiral en el plano *xy* se corresponde a la propia distribución original de los puntos. Este caso es, por tanto, al que será más complicado aplicar un ataque adversario, y el que necesitará desplazar los puntos del *training dataset* una mayor distancia para que la red los clasifique de manera incorrecta. Pese a tratarse de una *desventaja*, se aprovechará para representar la idea del desplazamiento de los puntos, lo que para el espacio 3D será más difícil.

El procedimiento será similar al de los ejemplos adversarios de las imágenes:

1. Selección del punto correspondiente al ataque adversario: cualquier punto de la espiral es válido para realizar la operación. En este caso se ha escogido uno de la clase '0' (color rojo).
2. Modificación de sus coordenadas: a las coordenadas iniciales del punto se añadirá un valor δ que se optimizará mediante el Descenso del gradiente reiteradamente.
3. Aplicación del criterio escogido para finalizar el ataque: detener el desplazamiento cuando la red adjudique al punto una probabilidad menor del 33 % de pertenecer a la clase verdadera.



(a) Desplazamiento del punto respecto al clasificador.

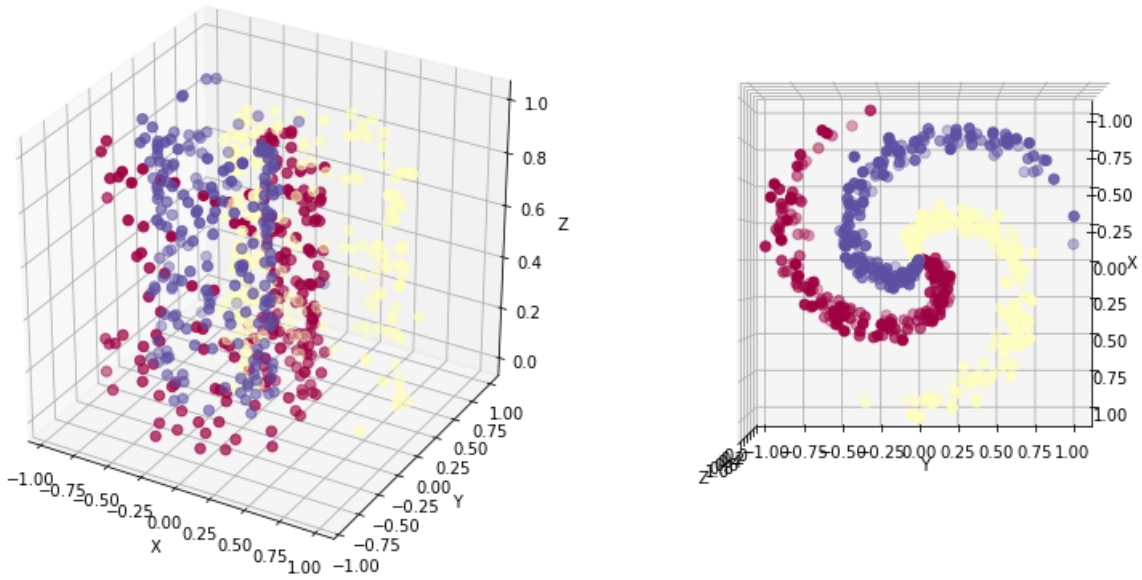
(b) Desplazamiento del punto en detalle.

Figura 4.13: Ataque adversario a un punto correspondiente a la clase '0'.El punto atacado se identifica con color negro, y el resto de puntos rojos define la traza que ha descrito hasta ser clasificado por la red como la clase '1'. En la imagen en detalle (b), el degradado de color ayuda a visualizar el movimiento del punto, que se hace más oscuro conforme llega a su destino.

A continuación, se trasladará esta idea al espacio tridimensional, del que se esperan sacar conclusiones de la vulnerabilidad de las redes.

4.2.2. Espiral en el espacio 3D

En este caso, el conjunto de datos sintéticos mantendrán la distribución de la espiral bidimensional, con la particularidad de que a cada uno de ellos se le asociará una altura aleatoria delimitada entre $[0, z]$ para diferentes valores de z (Anexo Código A.4).
Como caso general, se representa la espiral de valores comprendidos entre $[0, 1]$:



(a) Vista frontal de la espiral. **(b)** Vista desde arriba de la espiral.

Figura 4.14: Distribución de los datos correspondientes a la espiral de dispersión $z \in [0, 1]$.

Al igual que para el caso bidimensional, el uso de redes neuronales para el aprendizaje del modelo será más eficaz que uno lineal, permitirá alcanzar una mejor precisión y clasificará mejor los puntos de las diferentes clases. De forma análoga y con el mismo modelo definido anteriormente, se calcula la pérdida y la precisión de la red tras su entrenamiento.

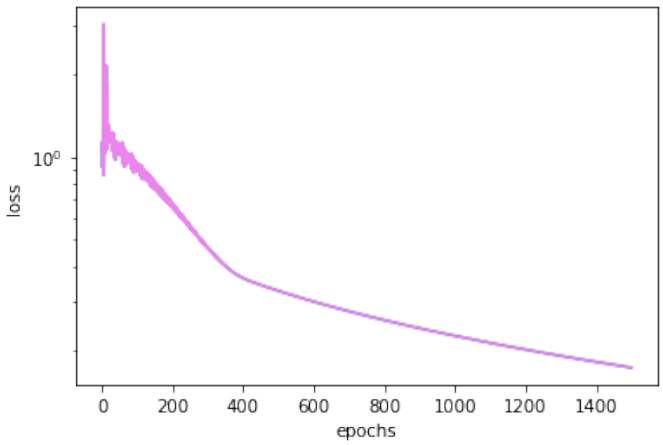


Figura 4.15: Cálculo de la pérdida mediante *Cross-Entropy* para la espiral de altura $z \in [0, 1]$.

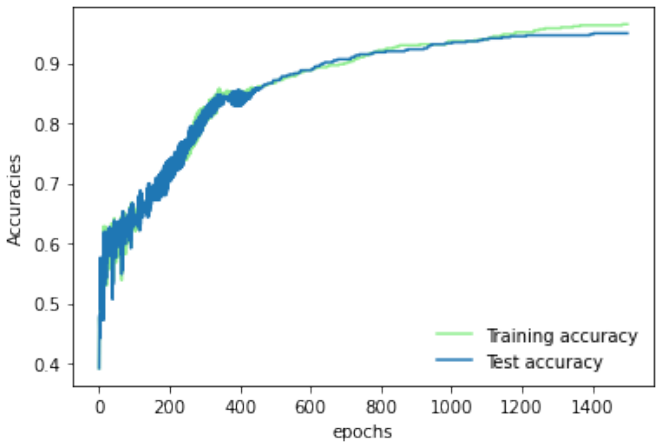


Figura 4.16: Cálculo de la precisión del aprendizaje. *Training accuracy* hace referencia a la precisión que alcanza el modelo durante en el entrenamiento y *Test accuracy* a laobtenida tras introducir como entrada datos diferentes a los de entrenamiento.

En las Figuras 4.15 y 4.16 se observa cómo los resultados han empeorado respecto a la espiral bidimensional, algo que se podía intuir al haber aumentado la dispersión de los datos. El clasificador tridimensional obtenido tras entrenar el modelo con la espiral de dimensión $z \in [0, 1]$ tendrá la siguiente forma:

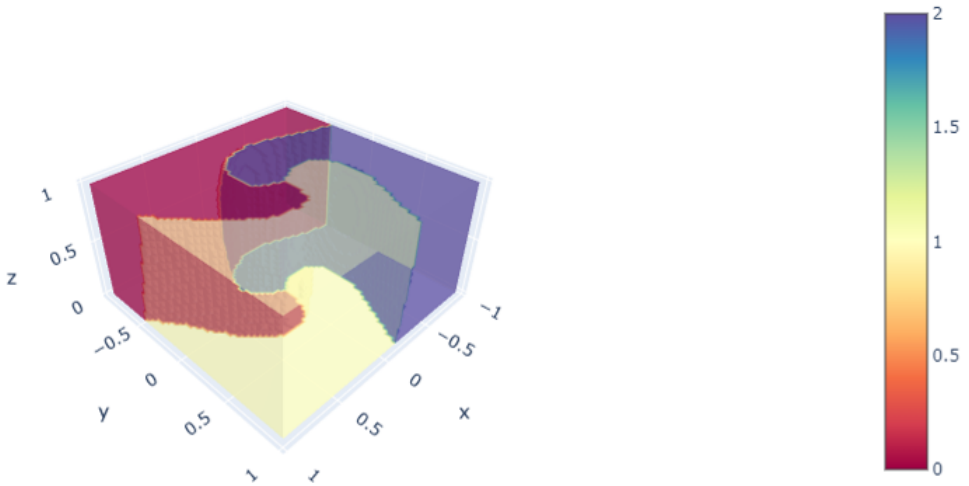
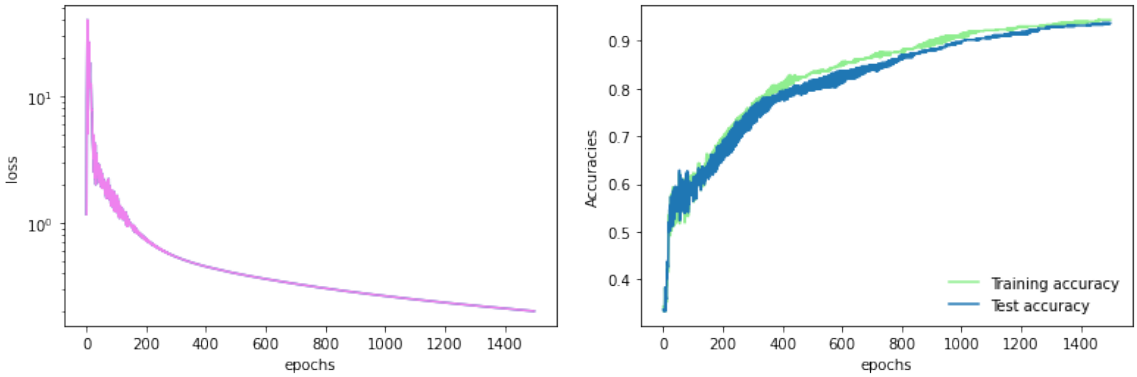


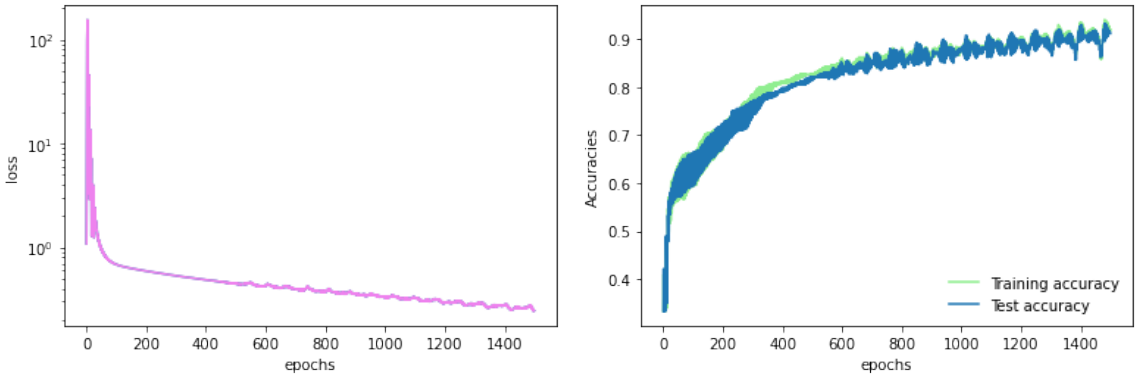
Figura 4.17: Clasificador obtenido tras la aplicación de las redes neuronales sobre los datos de la espiral con dispersión $z \in [0, 1]$.

Antes de realizar los diferentes ataques, será interesante ver cómo la precisión y la pérdida del modelo varían con el aumento de la dispersión de los datos para un mismo número de épocas de entrenamiento. Un conjunto de datos más compacto facilitará el aprendizaje, y lo dificultará en caso contrario.



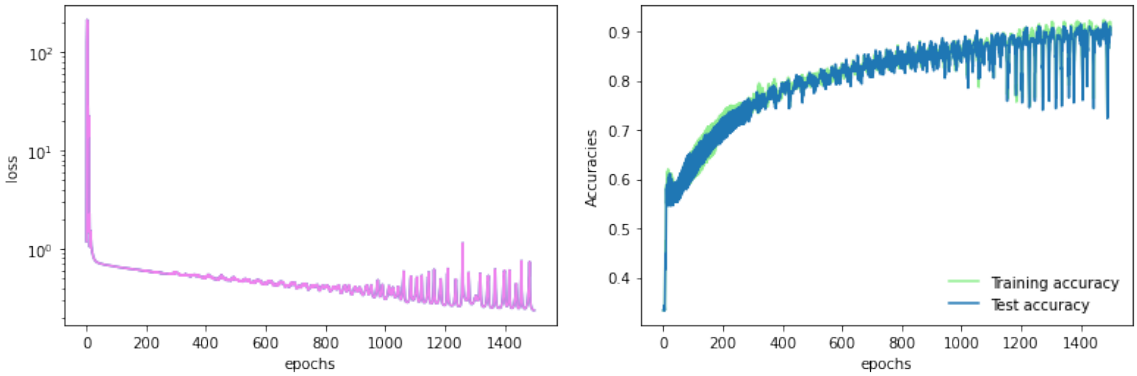
(a) Cálculo de la pérdida del entrenamiento. **(b)** Precisión del entrenamiento y del *test*.

Figura 4.18: Caso espiral con $z \in [0, 5]$.



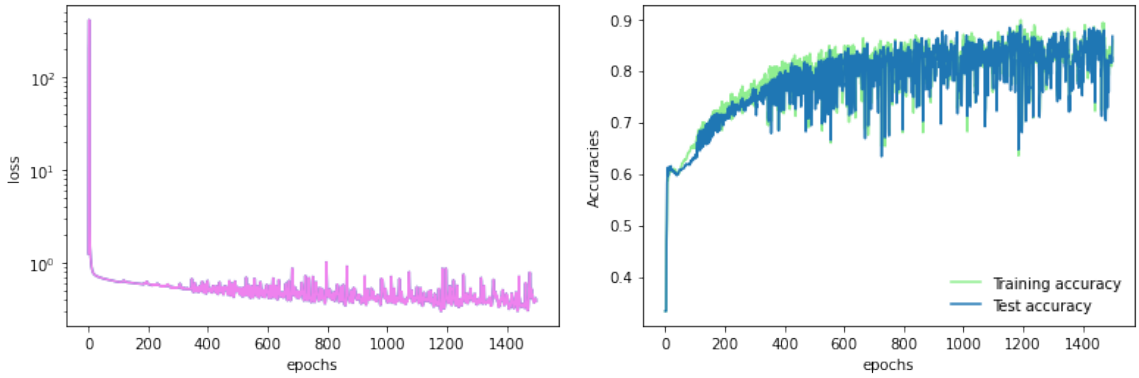
(a) Cálculo de la pérdida del entrenamiento. **(b)** Precisión del entrenamiento y del *test*.

Figura 4.19: Caso espiral con $z \in [0, 10]$.



(a) Cálculo de la pérdida del entrenamiento. **(b)** Precisión del entrenamiento y del *test*.

Figura 4.20: Caso espiral con $z \in [0, 15]$.



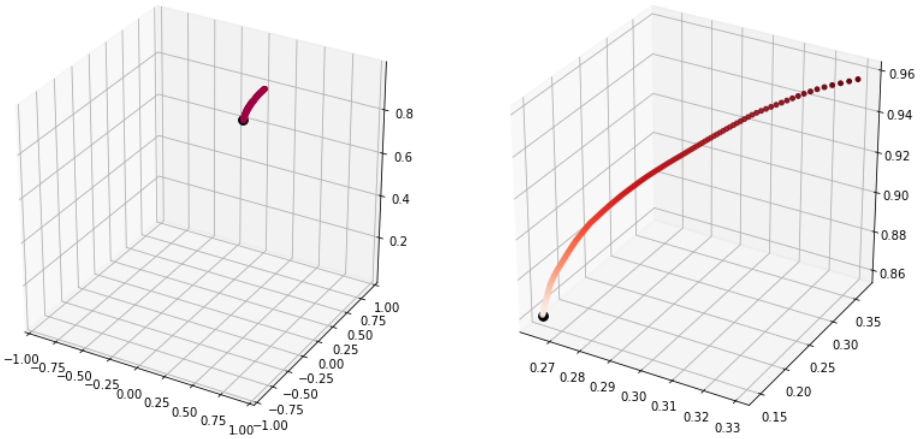
(a) Cálculo de la pérdida del entrenamiento. **(b)** Precisión del entrenamiento y del *test*.

Figura 4.21: Caso espiral con $z \in [0, 20]$.

4.2.2.1. Aplicación de ataques adversarios en el espacio tridimensional

De forma análoga al caso bidimensional, se procede a aplicar los ejemplos adversarios a partir del siguiente planteamiento:

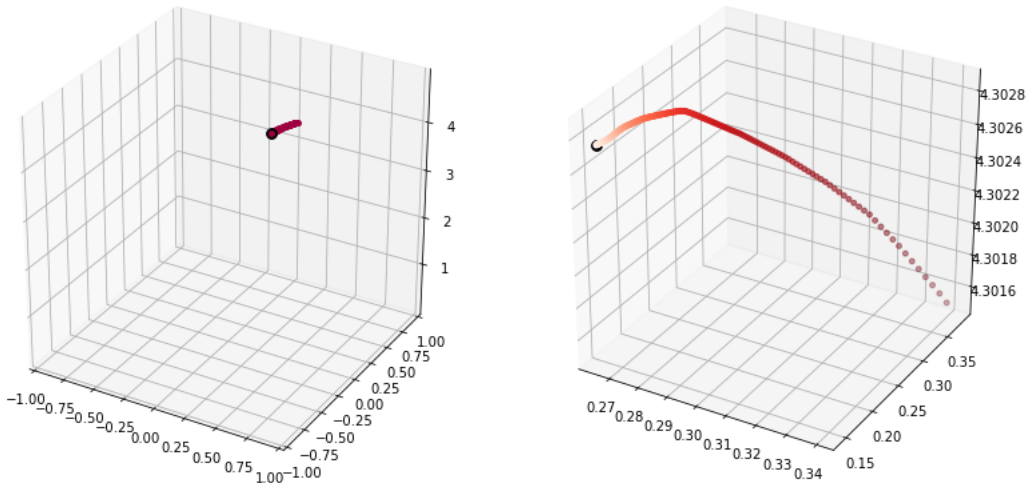
1. Selección del punto correspondiente al ataque adversario
2. Modificación de sus coordenadas: a las coordenadas iniciales del punto se añadirá un valor δ que se optimizará mediante el Descenso del gradiente reiteradamente.
3. Aplicación del criterio escogido para finalizar el ataque: detener el desplazamiento cuando la red adjudique al punto una probabilidad menor del 33 % de pertenecer a la clase verdadera.



(a) Desplazamiento del punto en el espacio 3D. **(b)** Desplazamiento del punto en detalle.

Figura 4.22: Ataque adversario aplicado a un punto de la clase '0' de la espiral definida para $z \in [0, 1]$. La marca negra indica la posición inicial del punto antes de ser modificadas sus coordenadas, y el degradado de color en la imagen (b) contribuye a un mejor seguimiento de la trayectoria que realiza. Se oscurece conforme llega a la posición en la que la red le atribuye una probabilidad menor del 33 % de pertenecer a la clase verdadera.

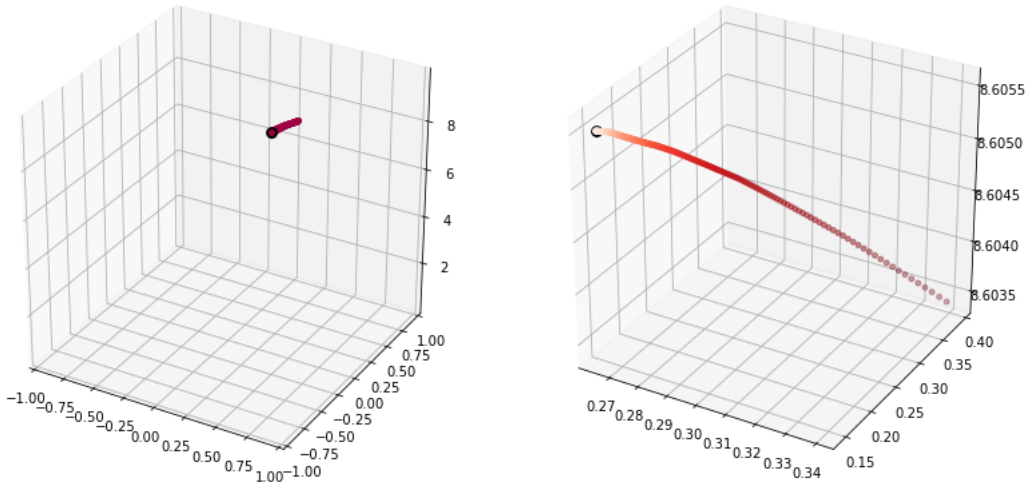
Este se repetirá para todas las dimensiones de la espiral que se han escogido para el estudio. A modo de facilitar la visión espacial del desplazamiento que recorren los puntos, se adjuntan dos tipos de figura por cada caso: una sobre el espacio tridimensional original, para poder apreciar el movimiento relativo, y otra en la que se representa el desplazamiento con más detalle.



(a) Desplazamiento del punto en el espacio 3D.

(b) Desplazamiento del punto en detalle.

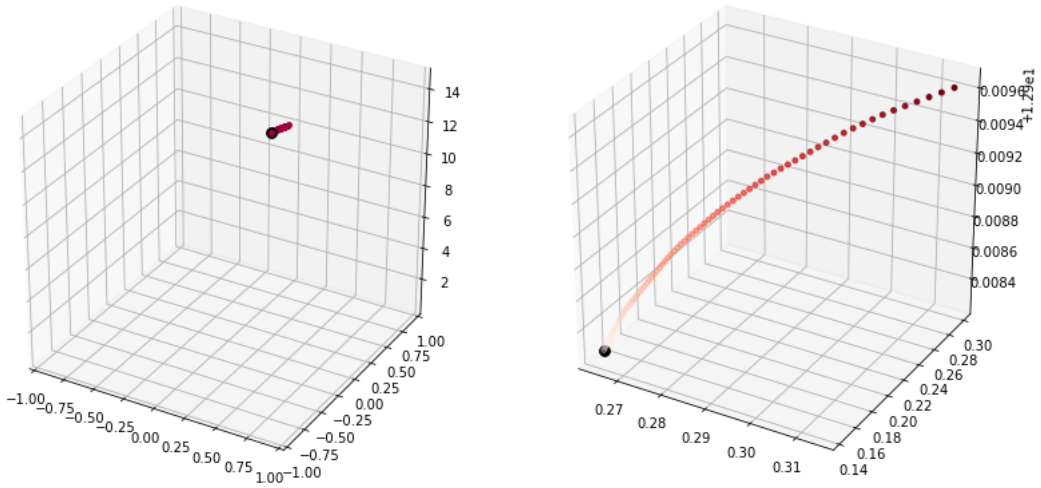
Figura 4.23: Ataque adversario aplicado a un punto de la clase '0' para $z \in [0, 5]$.La marca negra indica la posición inicial del punto antes de ser desplazado, y el degradado de color en la imagen (b) contribuye a un mejor seguimiento de la trayectoria que realiza.



(a) Desplazamiento del punto en el espacio 3D.

(b) Desplazamiento del punto en detalle.

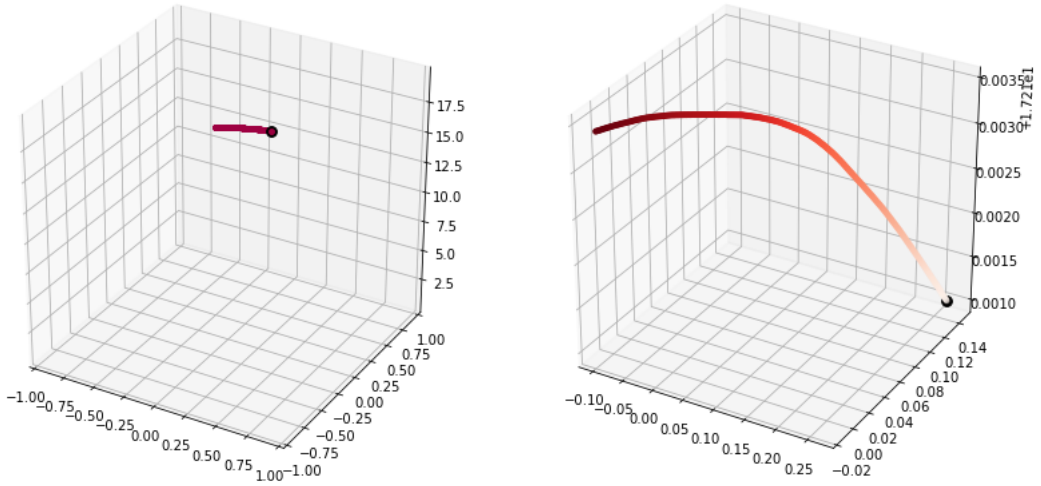
Figura 4.24: Ataque adversario aplicado a un punto de la clase '0' para $z \in [0, 10]$. La marca negra indica la posición inicial del punto antes de ser desplazado, y el degradado de color en la imagen (b) contribuye a un mejor seguimiento de la trayectoria que realiza.



(a) Desplazamiento del punto en el espacio 3D.

(b) Desplazamiento del punto en detalle.

Figura 4.25: Ataque adversario aplicado a un punto de la clase '0' para $z \in [0, 15]$. La marca negra indica la posición inicial del punto antes de ser desplazado, y el degradado de color en la imagen (b) contribuye a un mejor seguimiento de la trayectoria que realiza.



(a) Desplazamiento del punto en el espacio 3D.

(b) Desplazamiento del punto en detalle.

Figura 4.26: Ataque adversario aplicado a un punto de la clase '0' para $z \in [0, 20]$. La marca negra indica la posición inicial del punto antes de ser desplazado, y el degradado de color en la imagen (b) contribuye a un mejor seguimiento de la trayectoria que realiza.

A diferencia del caso dimensional, en el que el ataque adversario se entendía de manera visual gracias a la representación del desplazamiento sobre el propio clasificador, en el caso tridimensional, el trazado de los puntos no aporta suficiente información que permita evaluar el ataque tan fácilmente. Para remediarlo, se ha optado por medir dos valores durante los distintos ataques:

- **Distancia total recorrida**: suma de todos los desplazamientos que experimenta el punto, desde la posición inicial hasta que se detiene el ataque adversario.
- **Desplazamiento final**: mide el cambio de posición que experimenta el punto, tomando como referencia la posición inicial y la posición final.

Para calcular estos valores se ha utilizado la fórmula de la distancia entre dos puntos:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2} \tag{4.6}$$

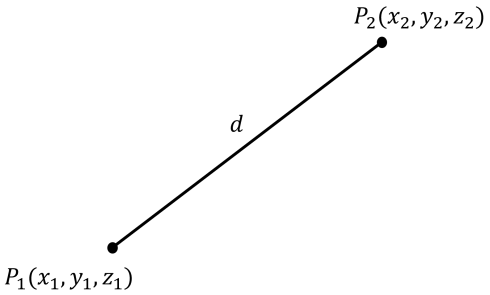


Figura 4.27: Representación de la distancia entre dos puntos.

El cálculo de la distancia de un sólo punto no sería representativo, pues dependiendo de donde esté inicialmente situado la proximidad a la región correspondiente a una clase diferente puede ser mayor o menor. Para evitar resultados poco significativos, se ha tomado una muestra aleatoria de 15 puntos de la espiral para aplicar los ataques adversarios y se ha obtenido el valor promedio de las distancias recorridas. Esta operación se ha repetido para los diferentes casos de magnitud de los datos.

z	Distancia total recorrida media	Desplazamiento final medio
$z \in [0, 1]$	0.206115	0.202078
$z \in [0, 5]$	0.210302	0.208688
$z \in [0, 10]$	0.222402	0.217519
$z \in [0, 15]$	0.222503	0.221282
$z \in [0, 20]$	0.209389	0.207148

Tabla 4.2: Distancia media recorrida por 15 puntos aleatorios de la espiral, independientemente de la clase inicial, para todos los casos de dispersión.

Según cómo se interpreten, estos resultados (Tabla 4.2) pueden ser o no lo que se esperaba. Lo que decía la intuición, previamente a realizar el estudio y como se ha visto de forma práctica en el caso de ejemplos adversarios con imágenes, es que, cuánto mayor sea la perturbación o ruido agregado a los datos originales, más fácil será violar la vulnerabilidad de la red. Trasladando esta idea al contexto de la espiral tridimensional,la conclusión a la que se esperaba llegar era que, para una mayor dispersión de los datos, menor sería la

distancia necesaria que habría que desplazar un punto para que el modelo lo clasificara como una clase incorrecta.

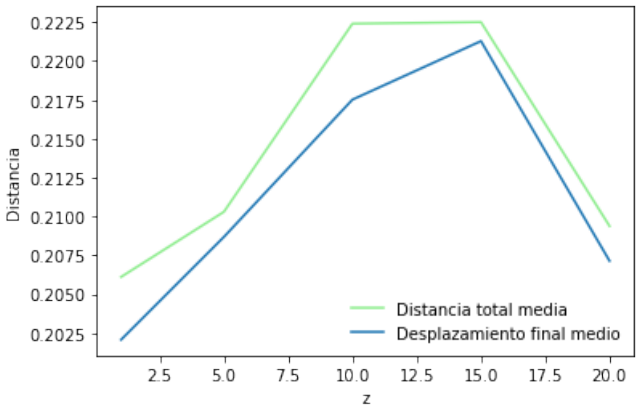


Figura 4.28: Representación gráfica de los valores obtenidos de las distancias recorridas para apreciar mejor la variación y la tendencia que toman.

Analizando los valores recogidos en la Tabla 4.2, que han sido representados en Figura 4.28, las distancias medidas absolutas no tienden a reducirse conforme aumenta la dimensión del espacio de los datos. Sin embargo, al haberse medido en espacios de diferentes dimensiones, podría decirse que las distancias relativas son menores cuanto mayor es la dispersión de los datos.

4.2.2.2. Aplicación de ataques adversarios dirigidos en el espacio tridimensional

Finalmente, se han aplicado ataques dirigidos a diversos puntos de la espiral para comprobar si la distancia recorrida se puede ver afectada. Para una muestra de 15 puntos aleatorios del conjunto de datos, se han programado los siguientes ataques según la clase verdadera de procedencia:

- Puntos de la clase '0' → se convertirán a la clase '1'.
- Puntos de la clase '1' → se convertirán a la clase '2'.
- Puntos de la clase '2' → se convertirán a la clase '0'.

Tras la aplicación de los ataques, se han vuelto a calcular las distancias totales y los desplazamientos medios de la nueva muestra de puntos, obteniéndose los siguientes resultados:

z	Distancia total recorrida media	Desplazamiento final medio
$z \in [0, 1]$	0.273288	0.271650
$z \in [0, 5]$	0.262874	0.262254
$z \in [0, 10]$	0.255814	0.255427
$z \in [0, 15]$	0.269896	0.268756
$z \in [0, 20]$	0.285303	0.282257

Tabla 4.3: Distancia media recorrida por 15 puntos aleatorios de la espiral, para todos los casos de dispersión, tras aplicarles un ataque adversario dirigido.

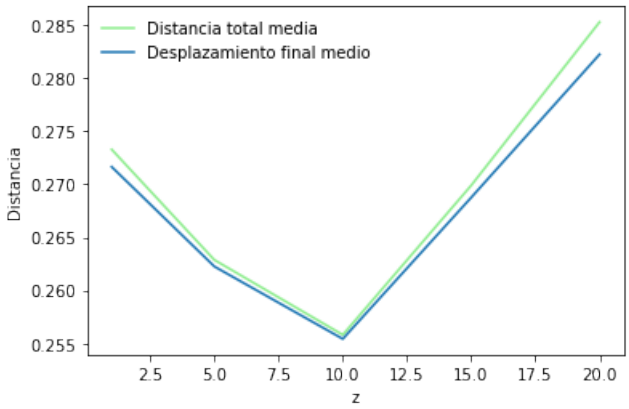


Figura 4.29: Representación gráfica de los valores obtenidos de las distancias recorridas durante los ataques adversarios dirigidos.

Por lo general, para los ataques dirigidos establecidos, parece que la distancia registrada por los puntos (Tabla 4.3) es superior al desplazamiento que recorren cuando los ataques no son dirigidos.

Se observa que ni para el caso de ataques no dirigidos ni para los dirigidos la distancia mantiene un crecimiento o decrecimiento monótono, si no que sube, baja y viceversa. Se espera que la inexistencia de una tendencia monótona decreciente dependa del criterio seleccionado para detener el ataque. Durante esta investigación se escogió que el desplazamiento del punto se detuviera cuando la red le asignara una probabilidad menor del 33 % de pertenecer a la clase verdadera. Se espera obtener mejores resultados con la selección de un criterio más riguroso y un efecto más claro al aumentar el número de dimensiones. Esto requerirá un estudio más en profundidad que se pospone para un trabajo futuro.

Capítulo 5

Conclusiones y líneas futuras

5.1. Conclusiones

A lo largo de este proyecto se han podido alcanzar diferentes conclusiones:

- **Aplicación de PCA para prevenir ataques adversarios**: como se ha demostrado con el caso de la imagen del cerdo, el método de Análisis de Componentes Principales sirve para filtrar la perturbación de los ataques. Este es capaz de transformar espacios de dimensión p a otras más pequeñas conservando la mayor información posible, lo que permite recuperar los datos originales previos a la adición de *ruido*.
- **La dispersión de los datos afecta a la precisión del modelo**: para un conjunto de datos sintéticos, cuanto mayor es la dispersión y la dimensión del espacio en el que se definen los datos, más difícil es para el modelo encontrar patrones que hagan eficaz su aprendizaje. El objetivo del entrenamiento de la red es minimizar las pérdidas obtenidas en las predicciones y mejorar su precisión, para lo que se actualizan los parámetros internos de la red iterativamente. Por lo tanto, cuanto mayor sea la dispersión de los datos sintéticos, más difícil será el ajuste de dichos parámetros. El caso en el que se ha trabajado durante el proyecto es un caso especial porque los datos siguen estando separados en clases atendiendo solo a la distribución en las dos primeras componentes, por lo que el problema tiene la misma complejidad al añadir dimensiones extra. Sin embargo, al modelo le cuesta más encontrar la solución del problema (cómo separar los puntos en las tres clases).
- **La distancia necesaria para cambiar un punto de clase se modifica al introducir la dispersión**: mientras que la distancia recorrida por los puntos de la espiral tridimensional tras la aplicación de ataques adversarios no decrece con la dispersión (contrario a lo que se esperaba), la distancia relativa sí que se hace más pequeña, pues para mayores dispersiones la distancia recorrida se mantiene casi constante.

5.2. Líneas futuras

Con la intención de retomar este proyecto en un futuro próximo con el objetivo de profundizar más en el estudio de la vulnerabilidad de los datos con el aumento de la dispersión, se recogen las siguiente siguientes propuestas que han quedado pendientes de estudio:

1. **Encontrar criterio óptimo para detener el ataque**: se estudiarán los posibles criterios que existen para encontrar aquel con el que las distancias recorridas por los puntos disminuyan con el aumento de la dispersión. Esto confirmará lo que ocurría con las imágenes, y es que ante una mayor cuantía de perturbaciones (en el caso de la espiral, la dispersión de los datos) más fácil será *burlar* las predicciones de la red mediante ataques adversarios.

2. **Sobredimensionar la espiral**: además del caso bidimensional y tridimensional, se estudiará cómo afectan los ataques en el caso de tener más dimensiones. Se esperaría que los efectos vistos a lo largo de este TFG se vieran mucho más claros con dimensiones extra.

3. **Comprimir y descomprimir los datos sintéticos**: una vez resueltos los puntos anteriores, será interesante estudiar cómo afecta la compresión de los datos a los ataques adversarios en función del número de dimensiones adicionales que se hayan incluido.

4. **Utilización de funciones de pérdida dinámicas**: Se ha comprobado que entrenar la red con funciones de coste dinámicas (*dynamical loss functions*) mejora la generalización de los modelos [29]. Estas funciones de pérdida están inspiradas en la manipulación de las superficies de potencial en problemas físicos [30] para facilitar la minimización y conseguir que lleven a cabo funciones específicas. En el caso de redes profundas, usar las funciones de pérdida dinámicas facilita enormemente el entrenamiento. La mejora proviene de que los mínimos locales desaparecen cíclicamente, ayudando la minimización. Este efecto se puede entender mejor usando modelos sencillos que generan superficies de potencial que se pueden estudiar analíticamente [31, 32, 33]. En estos casos los cambios de temperatura o en el parámetro que controla la interacción entre las partículas da lugar a bifurcaciones que hacen que aparezcan o desaparezcan mínimos locales en la superficie de potencial. Por lo tanto, la aplicación de estas bifurcaciones en esta investigación facilitarían el entrenamiento y ayudarían a encontrar soluciones que generalicen mejor. De la misma manera, sería interesante comprobar cómo afectan a los ataques adversarios otros protocolos de entrenamiento u otras arquitecturas que eviten el olvido catastrófico de los modelos [34].

Capítulo 6

Planificación temporal y presupuesto

6.1. Planificación temporal

La estructura temporal del proyecto puede descomponerse en cuatro periodos: familiarización con las redes neuronales y Python, estudio de los ataques adversarios y aplicación de PCA, estudio exhaustivo de la vulnerabilidad de un conjunto de datos sintéticos frente a ataques adversarios y redacción y perfeccionamiento del documento.

Familiarización con las redes neuronales y Python

A finales de octubre de 2020 tuvo lugar la asignación del TFG. Puesto a que los conocimientos sobre *Machine Learning* previos a la adjudicación eran limitados, el tutor facilitó herramientas para adentrarse en el mundo de las redes neuronales y en sus herramientas de programación:

- Curso sobre redes neuronaes convolucionales: *CS231n: Convolutional Neural Networks for Visual Recognition* [28].
- Introducción a Python.
- Introducción a Google Colab [35] y a Jupyter Notebook [36], ambas herramientas utilizadas durante el desarrollo del trabajo.
- Seminario de formación en Machine Learning.

Ataques adversarios y aplicación PCA

Tras una larga temporada dedicada a la lectura de artículos y desarrollo de tareas para familiarizarse con el código, estructura de las redes neuronales y funcionamiento de las redes neuronales convolucionales, decidió orientarse la investigación hacia el interesante y, en gran parte desconocido, campo de los ataques adversarios. Este fue un proceso más corto en comparación con el anterior que despertó mucho el interés de la autora.

Se investigó también sobre el Análisis de Principales Componentes (PCA), la compresión de imágenes en Python y la implementación de PCA sobre los ejemplos adversarios.

Estudio de la vulnerabilidad de un conjunto de datos frente a ataques adversarios

Una vez obtenidos resultados ilustrativos a partir de imágenes a las que se les aplicaban ejemplos adversarios, se ha estudiado cómo la adición de dimensiones y el incremento de la dispersión de un conjunto de datos sintéticos con forma de espiral, puede ver afectada su vulnerabilidad frente a diversos ataques.

Redacción y perfeccionamiento del trabajo

Durante los dos últimos meses se han buscando resultados representativos que demostraran la hipótesis que se pretendía demostrar: el aumento de la dispersión de los datos de entrada facilitaría la aplicación de los ataques adversarios. Al mismo tiempo se ha llevado a cabo la redacción del proyecto, que ha permitido asentar los conocimientos de toda la investigación.

6.1.1. Diagrama de Gantt

Distribución temporal		
Actividad	Fecha de Inicio	Fecha de Fin
Reuniones para la asignación del TFG	28-oct-20	30-oct-20
Formación y aprendizaje	31-oct-20	21-may-21
Estudio Ataques Adversarios y PCA	21-may-21	16-sep-21
Estudio sobredimensionamiento de la espiral	16-sep-21	22-nov-21
Aplicación Ataques Adversarios a la espiral	22-nov-21	27-ene-22
Redacción y correción del documento	27-ene-22	07-feb-22

Tabla 6.1: Tabla de la planificación del desarrollo del Trabajo.

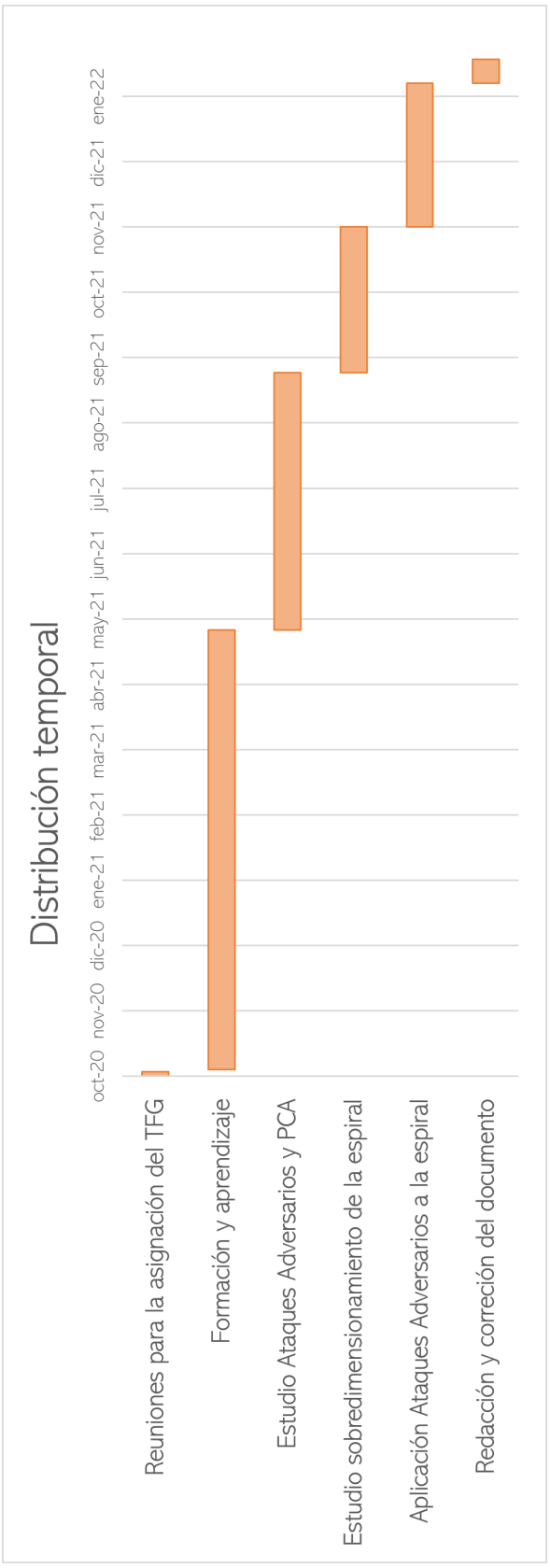


Figura 6.1: Diagrama de GANTT.

6.2. Presupuesto

El presupuesto total de un proyecto se contabiliza distinguiendo los recursos utilizados para su consecución, diferenciando los Recursos humanos y aquellos de índole material o soporte de este trabajo.

Recursos humanos

Por un lado, en lo que respecta al presupuesto asociado a los RR.HH., es necesario tener en cuenta la actividad realizada por el alumno y el tutor. Tanto el coste horario como las horas invertidas se recogen en la Tabla 6.2.

Coste del tiempo de trabajo			
Actividad	Presupuesto unitario [€/hora]	Horas	Subtotal [€]
Alumno	15	450	6750
Tutor	30	150	4500
Coste total			**11250**

Tabla 6.2: Tabla de coste del tiempo total de trabajo.

Recursos materiales

En este apartado se considerarán el ordenador y los programas utilizados para desarrollar el proyecto:

- **Ordenador**: el dispositivo utilizado es un ordenador MSI con procesador Intel CORE i7-10710U, que tiene un TDP de 25 W, y una GPU NVIDIA GeForce GTX 1650, con una TDP de 50 W, lo que equivale a un consumo de 75 Wh. Para un total de 450 horas que la alumna ha hecho uso del dispositivo, y un precio de 0.21€/kWh, el gasto total asociado al uso del ordenador será de 7.09€.
- **Software**: para el desarrollo del código, escrito en Python, se ha empleado el entorno online de Google Colab y Jupyter Notebook, y la redacción del trabajo se ha realizado en Overleaf, todos de manera gratuita. En consecuencia, el software no supondrá ningún coste.
- **Otros costes**: la necesidad de conexión a Internet durante los meses de duración del proyecto se estima en un total de 20€.

Por lo tanto, el presupuesto total asociado al proyecto será el calculado en la Tabla 6.3

Presupuesto	
Tipo de Coste	Coste [€]
Tiempo de Trabajo	11250
Consumo del Ordenador	7.09
Otros costes	20
Coste total TFG	**11277.09**

Tabla 6.3: Presupuesto total del Proyecto.

Bibliografía

[1] Attal, M. (2021, December 13). Machine Learning: definición, funcionamiento, usos. Retrieved February 5, 2022, from Formación en ciencia de datos — DataScientest.com website: https://datascientest.com/es/machine-learning-definicion-funcionamiento-usos

[2] Gonzalez, J. L. (2018, February 8). Tipos de aprendizaje automático - SoldAI - Medium. Retrieved February 5, 2022, from SoldAI website: https://medium.com/soldai/tipos-de-aprendizaje-autom%C3%A1tico-6413e3c615e2

[3] Responsables de automatización, sistemas "cloud" y "machine learning", los puestos de trabajo más demandados. (2022, January 31). Retrieved February 5, 2022, from Computerworld.es website: https://www.computerworld.es/tendencias/responsables-de-automatizacion-sistemas-cloud-y-machine-learning-los-puestos-de-trabajo-mas-demandados

[4] Reynolds, M. (2017, July 27). Sneaky attacks trick AIs into seeing or hearing what's not there. New Scientist. https://www.newscientist.com/article/2142059-sneaky-attacks-trick-ais-into-seeing-or-hearing-whats-not-there/

[5] Cisse, M., Adi, Y., Neverova, N., & Keshet, J. (2017). Houdini: Fooling deep structured prediction models. In arXiv [stat.ML]. http://arxiv.org/abs/1707.05373

[6] García, J. D. V. (2019, February 28). Redes neuronales desde cero (I) - Introducción. Retrieved February 5, 2022, from IArtificial.net website: https://www.iartificial.net/redes-neuronales-desde-cero-i-introduccion/

[7] Berzal, F. (2018). Redes Neuronales and Deep Learning

[8] Dot, C. S. V. [DotCSV]. (2018, October 3). ¿Qué es una Red Neuronal? Parte 3: Backpropagation — DotCSV. Youtube. https://www.youtube.com/watch?v=eNIqz_noix8&li&list=PL-Ogd76BhmcB9OjPucsnc2-piEE96jJDQ&index=4

[9] Dot, C. S. V. [DotCSV]. (2018a, February 4). ¿Qué es el Descenso del Gradiente? Algoritmo de Inteligencia Artificial — DotCSV. Youtube.https://www.youtube.com/watch?v=A6FiCDoz8_4

[10] Durán, J. (2019, September 4). Todo lo que Necesitas Saber sobre el Descenso del Gradiente Aplicado a Redes Neuronales. Retrieved February 5, 2022, from MetaDatos website: https://medium.com/metadatos/todo-lo-que-necesitas-saber-sobre-el-descenso-del-gradiente-aplicado-a-redes-neuronales-19bdbb706a78

[11] Ossorio, E. A. (2021, February 19) Análisis y predicción a corto plazo de la producción de un parque eólico mediante redes neuronales y otras técnicas de Machine Learning.

[12] Alberto, R. (2020, October 16). Explicación de las Funciones de activación en Redes Neuronales y práctica con Python. Retrieved February 5, 2022, from Medium website: https://rubialesalberto.medium.com/explicaci%C3%B3n-funciones-de-activaci%C3%B3n-y-pr%C3%A1ctica-con-python-5807085c6ed3

[13] Calvo, D. (2018, December 10). Función de coste - Redes neuronales. Retrieved February 5, 2022, from Diego Calvo website: https://www.diegocalvo.es/funcion-de-coste-redes-neuronales/

[14] Heras, J. M. (2018, December 28). Error Cuadrático Medio para Regresión. Retrieved February 5, 2022, from IArtificial.net website: https://www.iartificial.net/error-cuadratico-medio-para-regresion/

[15] Na. (2018a, September 12). Breve Historia de las Redes Neuronales Artificiales. Retrieved February 5, 2022, from Aprendemachinelearning.com website: https://www.aprendemachinelearning.com/breve-historia-de-las-redes-neuronales-artificiales/

[16] Ramírez, F. (2019, November 26). Las matemáticas del Machine Learning: Redes Neuronales (Parte I). Retrieved February 5, 2022, from Think Big website: https://empresas.blogthinkbig.com/las-matematicas-del-machine-learning-redes-neuronales-parte-i/

[17] Torres, A. (2021, October 14). Aprendizaje automático: Una introducción al error cuadrático medio y las líneas de regresión. Retrieved February 5, 2022, from freeCodeCamp.org website: https://www.freecodecamp.org/espanol/news/aprendizaje-automatico-una-introduccion-al-error-cuadratico-medio-y-las-lineas-de-regresion/

[18] Na. (2018b, November 29). Convolutional Neural Networks: La Teoría explicada en Español. Retrieved February 5, 2022, from Aprendemachinelearning.com website: https://www.aprendemachinelearning.com/como-funcionan-las-convolutional-neural-networks-vision-por-ordenador/

[19] Dot, C. S. V. [DotCSV]. (2020, November 12).¡Redes Neuronales CONVOLUCIONALES! ¿Cómo funcionan? Youtube. https://www.youtube.com/watch?v=V8j1oENVz00

[20] Ataques Adversarios: ¿una debilidad de las Redes Neuronales? (n.d.). Retrieved February 5, 2022, from Codificando Bits website: https://www.codificandobits.com/blog/ataques-adversarios/

[21] Ilyas, A., Santurkar, S., Tsipras, D., Engstrom, L., Tran, B., & Madry, A. (2019). Adversarial examples are not bugs, they are features. In arXiv [stat.ML]. http://arxiv.org/abs/1905.02175

[22] Data, S. B. (2019, May 1). Ciberseguridad y vulnerabilidades de redes neuronales. Retrieved February 5, 2022, from sitiobigdata.com website: https://sitiobigdata.com/2019/05/01/ciberseguridad-vulnerabilidades-de-redes-neuronales/

[23] C. S. V. [DotCSV]. (2018a, January 17). ¿Cómo engañar a una RED NEURONAL? Ataques Adversarios — DATA COFFEE #6. Retrieved February 5, 2022, from https://www.youtube.com/watch?v=lPyogLghTKo

[24] Geitgey, A. (2017, August 16). Machine learning is fun part 8: How to intentionally trick neural networks. Retrieved February 5, 2022, from Medium website: https://medium.com/@ageitgey/machine-learning-is-fun-part-8-how-to-intentionally-trick-neural-networks-b55da32b7196

[25] Kolter, Z., & Madry, A. (n.d.). Chapter 1 - Introduction to adversarial robustness. Retrieved February 5, 2022, from Adversarial-ml-tutorial.org website: https://adversarial-ml-tutorial.org/introduction/

[26] Rodrigo, J. A. (n.d.). Análisis de Componentes Principales (Principal Component Analysis, PCA) y t-SNE. Retrieved February 5, 2022, from Cienciadedatos.net website: https://www.cienciadedatos.net/documentos/35_principal_component_analysis

[27] Tjandra, T. (n.d.). Image Compression.Ipynb at master · tomytjandra/UL-image-compression.

[28] CS231n Convolutional Neural Networks for Visual Recognition. (n.d.). Retrieved February 5, 2022, from Github.io website: https://cs231n.github.io/

[29] Ruiz-Garcia, M., Zhang, G., Schoenholz, S. S., & Liu, A. J. (2021, July). Tilting the playing field: Dynamical loss functions for machine learning. In International Conference on Machine Learning (pp. 9157-9167). PMLR.

[30] Ruiz-García, M., Liu, A. J., & Katifori, E. (2019). Tuning and jamming reduced to their minima. Physical Review E, 100(5), 052608.

[31] Ruiz-Garcia, M., Bonilla, L. L., & Prados, A. (2017). Bifurcation analysis and phase diagram of a spin-string model with buckled states. Physical Review E, 96(6), 062147.

[32] Ruiz-García, M., Bonilla, L. L., & Prados, A. (2016). STM-driven transition from rippled to buckled graphene in a spin-membrane model. Physical Review B, 94(20), 205404.

[33] Cea, T., Ruiz-Garcia, M., Bonilla, L. L., & Guinea, F. (2020). Numerical study of the rippling instability driven by electron-phonon coupling in graphene. Physical Review B, 101(23), 235428.

[34] Ruiz-Garcia, M. (2021). Some thoughts on catastrophic forgetting and how to learn an algorithm. arXiv preprint arXiv:2108.03940.

[35] Google Colaboratory (n.d.) https://colab.research.google.com/

[36] Project jupyter. (n.d.). Jupyter.Org. Retrieved February 7, 2022, from https://jupyter.org/

Índice de figuras

Índice de tablas

Acrónimos

CE	*Cross-entropy* (Entropía cruzada)
CNN	*Convolutional Neural Network* (Red Neuronal Convolucional)
IA	Inteligencia Artificial
MAE	Error Absoluto Medio
MASE	Error Absoluto Medio Escalado
MSE	Error Cuadrático Medio
PCA	*Principal Component Analysis* (Análisis de Componentes Principales)
ReLU	Función Lineal Rectificadora
TDP	*Thermal Design Power* (Potencia de diseño térmico)
TFG	Trabajo de Fin de Grado

Apéndices

Apéndice A

Códigos Python

A.1. Ataque adversario no dirigido

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

# In[1]:


from PIL import Image
from torchvision import transforms
import matplotlib.pyplot as plt

# read the image, resize to 224 and convert to PyTorch Tensor
pig_img = Image.open("pig.jpg")
preprocess = transforms.Compose([
   transforms.Resize(224),
   transforms.ToTensor(),
])
pig_tensor = preprocess(pig_img)[None,:,:,:]

plt.imshow(pig_tensor[0].numpy().transpose(1,2,0))
type(pig_img)


# In[2]:


import torch
import torch.nn as nn
from torchvision.models import resnet50

# simple Module to normalize an image
class Normalize(nn.Module):
    def __init__(self, mean, std):
        super(Normalize, self).__init__()
        self.mean = torch.Tensor(mean)
        self.std = torch.Tensor(std)
    def forward(self, x):
        return (x - self.mean.type_as(x)[None,:,None,None]) /
    ↪ self.std.type_as(x)[None,:,None,None]

# values are standard normalization for ImageNet images,
```

```
# from https://github.com/pytorch/examples/blob/master/imagenet/main.py
norm = Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

# load pre-trained ResNet50, and put into evaluation mode (necessary to
    ↪ e.g. turn off batchnorm)
model = resnet50(pretrained=True)
model.eval();


# In[3]:


# form predictions
pred = model(norm(pig_tensor))


# In[4]:


import json
with open("imagenet_class_index.json") as f:
    imagenet_classes = {int(i):x[1] for i,x in json.load(f).items()}
print("Predicted class: ", imagenet_classes[pred.max(dim=1)[1].item()])
print("Predicted probability:",
    ↪ nn.Softmax(dim=1)(pred)[0,pred.max(dim=1)[1]].item())


# In[5]:


# Aplicación ataque adversario
import torch.optim as optim
epsilon = 2./255

delta = torch.zeros_like(pig_tensor, requires_grad=True)
opt = optim.SGD([delta], lr=1e-1)

for t in range(30):
    pred = model(norm(pig_tensor + delta*3))
    loss = -nn.CrossEntropyLoss()(pred, torch.LongTensor([341]))
    if t % 5 == 0:
        print(t, loss.item())

    opt.zero_grad()
    loss.backward()
    opt.step()
    delta.data.clamp_(-epsilon, epsilon)

max_class = pred.max(dim=1)[1].item()
print("True class probability:", nn.Softmax(dim=1)(pred)[0,341].item())
print("Predicted class: ", imagenet_classes[max_class])
print("Predicted probability:",
    ↪ nn.Softmax(dim=1)(pred)[0,max_class].item())


# In[6]:


```

```
95 plt.imshow((pig_tensor + delta*3)[0].detach().numpy().transpose(1,2,0))
96
97
98 # In[ ]:
```

Código A.1: Ataque adversario no dirigido

A.2. Ataque adversario dirigido

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

# In[1]:


from PIL import Image
from torchvision import transforms
import matplotlib.pyplot as plt

# read the image, resize to 224 and convert to PyTorch Tensor
pig_img = Image.open("pig.jpg")
preprocess = transforms.Compose([
   transforms.Resize(224),
   transforms.ToTensor(),
])
pig_tensor = preprocess(pig_img)[None,:,:,:]

plt.imshow(pig_tensor[0].numpy().transpose(1,2,0))
type(pig_img)


# In[2]:


import torch
import torch.nn as nn
from torchvision.models import resnet50

# simple Module to normalize an image
class Normalize(nn.Module):
    def __init__(self, mean, std):
        super(Normalize, self).__init__()
        self.mean = torch.Tensor(mean)
        self.std = torch.Tensor(std)
    def forward(self, x):
        return (x - self.mean.type_as(x)[None,:,None,None]) /
   ↪ self.std.type_as(x)[None,:,None,None]

# values are standard normalization for ImageNet images,
# from https://github.com/pytorch/examples/blob/master/imagenet/main.py
norm = Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])

# load pre-trained ResNet50, and put into evaluation mode (necessary to
   ↪ e.g. turn off batchnorm)
model = resnet50(pretrained=True)
model.eval();


# In[3]:


# form predictions
pred = model(norm(pig_tensor))


```

```
# In[4]:


import json
with open("imagenet_class_index.json") as f:
    imagenet_classes = {int(i):x[1] for i,x in json.load(f).items()}
print("Predicted class: ", imagenet_classes[pred.max(dim=1)[1].item()])
print("Predicted probability:",
    ↪ nn.Softmax(dim=1)(pred)[0,pred.max(dim=1)[1]].item())


# In[31]:


# Aplicación ataque adversario dirigido. Clase objetivo: ambulancia
    ↪ (407)
import torch.optim as optim
epsilon = 2./255

delta = torch.zeros_like(pig_tensor, requires_grad=True)
opt = optim.SGD([delta], lr=1e-3)

for t in range(100):
    pred = model(norm(pig_tensor + delta*5))
    loss = (-nn.CrossEntropyLoss()(pred, torch.LongTensor([341])) +
    ↪ nn.CrossEntropyLoss()(pred,torch.LongTensor([407])))
    if t % 5 == 0:
        print(t, loss.item())

    opt.zero_grad()
    loss.backward()
    opt.step()
    delta.data.clamp_(-epsilon, epsilon)

max_class = pred.max(dim=1)[1].item()
print("True class probability:", nn.Softmax(dim=1)(pred)[0,341].item())
print("Predicted class: ", imagenet_classes[max_class])
print("Predicted probability:",
    ↪ nn.Softmax(dim=1)(pred)[0,max_class].item())


# In[32]:


plt.imshow((pig_tensor + delta*5)[0].detach().numpy().transpose(1,2,0))


# In[ ]:
```

Código A.2: Ataque adversario dirigido

A.3. Caso espiral 2D

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

# In[1]:


import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import matplotlib.colors as mcolors
from mpl_toolkits.mplot3d import Axes3D

import pandas as pd
import numpy as np

from random import random

import torch
import torch.nn as nn
import torch.nn.functional as F


# In[2]:


np.random.seed(0)
N = 200 # number of points per class
D = 2 # dimensionality cambiamos a 2D
K = 3 # number of classes
C = K

seed = 0

w_max_list = [1, 20] # 1 corresponds to no oscillations
last_period_no_osc = 1

total_time = 1000
height = 1 #dispersion of the data along the Z coordinate

T = int(total_time/20)

time_comp_density = T  # Time step to compute Hessian eigenvalues


# In[3]:


def make_dataset_torch(points_per_class, classes, revolutions=4):
    np.random.seed(0)

    X = np.zeros((N * C, D))
    y = np.zeros((N * C))

    #%matplotlib notebook
    fig=plt.figure()

```

```
    for j in range(K):
      ix = range(N*j,N*(j+1))
      r = np.linspace(0.0,1,N) # radius
      t = np.linspace(j*4,(j+1)*4,N) + np.random.randn(N)*0.2 # theta


      X[ix] = np.c_[r*np.sin(t), r*np.cos(t)]
      y[ix] = j

    fig = plt.figure()
    plt.scatter(X[:, 0], X[:, 1], c=y, s=35, cmap=plt.cm.Spectral)
    plt.xlim([-1,1])
    plt.ylim([-1,1])

    return torch.tensor(X, dtype=torch.float), torch.tensor(y,
    ↪ dtype=torch.long)



def make_dataset_torch_val(points_per_class, classes, revolutions=4):
    np.random.seed(1)


    X = np.zeros((N * C, D))
    y = np.zeros((N * C))

    #%matplotlib notebook
    fig=plt.figure()

    for j in range(K):
      ix = range(N*j,N*(j+1))
      r = np.linspace(0.0,1,N) # radius
      t = np.linspace(j*4,(j+1)*4,N) + np.random.randn(N)*0.2 # theta


      X[ix] = np.c_[r*np.sin(t), r*np.cos(t)]
      y[ix] = j

    fig = plt.figure()
    plt.scatter(X[:, 0], X[:, 1], c=y, s=35, cmap=plt.cm.Spectral)

    plt.xlim([-1,1])
    plt.ylim([-1,1])
    #print(y)

    return torch.tensor(X, dtype=torch.float), torch.tensor(y,
    ↪ dtype=torch.long)


# In[4]:


x_data_train_zeros,y_data_train_zeros =
    ↪ make_dataset_torch(points_per_class=N, classes=K)
x_data_test_zeros,y_data_test_zeros =
    ↪ make_dataset_torch_val(points_per_class=N, classes=K)

print("Clase 0: rojo")
```

```
111 print("Clase 1: amarillo")
112 print("Clase 2: azul")
113
114
115 # Clasificador lineal
116
117 # In[7]:
118
119
120 # Creación del modelo de la red
121 nn_input_dim = 2
122 nn_output_dim = 3
123 w_max_osc = 1
124
125 gamma_LR = 1
126 nn_width = 2000
127 learning_rate = 0.1
128
129 n_epochs = 1500
130 period_osc = 100
131
132 no_osc_last_epochs = 3
133
134 Softmax_com = torch.nn.Softmax(dim=1)
135
136 class Net(nn.Module):
137
138     def __init__(self):
139
140         super(Net, self).__init__()
141         self.ly1 = nn.Linear(nn_input_dim, nn_width)
142         self.ly2 = nn.Linear(nn_width, nn_width)
143         self.ly3 = nn.Linear(nn_width, nn_output_dim)
144
145     def forward(self, x):
146         x = self.ly1(x)
147         x = F.relu(x)
148         x = self.ly3(x)
149         return x
150
151 C = nn_output_dim
152
153 def c_fn(t, i, w_max, T):
154     slope = 2 * (w_max - 1) / T
155     w_main_class = np.where(t < T / 2., 1 + t * slope, 2 * w_max - t *
    ↪ slope - 1)
156     res = np.ones(C) + (w_main_class - 1) * np.eye(C)[i]
157     res = res / np.sum(res) * C
158     return torch.tensor(res, dtype=torch.float)
159
160 c_fn(10,0,w_max_osc,20)
161 print(w_max_osc)
162
163
164 # In[9]:
165
166
167 # Entrenamiento
```

```
net = Net()

optimizer = torch.optim.SGD(net.parameters(), lr=learning_rate)
scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer,
    ↪ gamma=gamma_LR)

def calc_accuracy(mdl,X,Y):
     max_vals, max_indices = torch.max(mdl(X),1)
     train_acc = (max_indices ==
    ↪ Y).sum().data.numpy()/max_indices.size()[0]
     return train_acc

loss_save = []
loss_unweighted_save = []
tr_accuracy_save = []
test_accuracy_save = []

tt = 0
ii = 0
cc = 0

batch_x, batch_y = make_dataset_torch(points_per_class=N, classes=K)

print('np.shape(batch_x)',np.shape(batch_x))
print('np.shape(batch_y)',np.shape(batch_y))

for epoch in range(n_epochs):

    optimizer.zero_grad()

    # in case you wanted a semi-full example
    outputs = net.forward(batch_x)

    # Oscilatorio
    if tt > period_osc:

        print('New Period ')
        print('epoch', epoch, 'LR', optimizer.param_groups[0]['lr'],
    ↪ 'loss', loss_forsaving_uw) #, 'training  acc: ', tr_accuracy,
    ↪ 'test_accuracy: ', test_accuracy)


        tt = 0
        cc = (cc + 1) % nn_output_dim

        if no_osc_last_epochs:

            if ii > n_epochs - no_osc_last_epochs*period_osc:
                print('last period without oscillations')
                w_max_osc = 1

    lossfunction = nn.CrossEntropyLoss(weight=c_fn(tt, cc, w_max_osc,
    ↪ period_osc))

    loss = lossfunction(outputs,batch_y)

    loss.backward()

```

```
221     optimizer.step()
222 
223     tt += 1
224     ii += 1
225 
226     scheduler.step()
227 
228     tr_accuracy =
    ↪ calc_accuracy(net,x_data_train_zeros,y_data_train_zeros)
229     test_accuracy =
    ↪ calc_accuracy(net,x_data_test_zeros,y_data_test_zeros)
230 
231     loss_forsaving = lossfunction(outputs,batch_y)
232 
233     lossfunction = nn.CrossEntropyLoss(weight=c_fn(tt, cc, 1,
    ↪ period_osc))
234     loss_forsaving_uw = lossfunction(outputs,batch_y)
235 
236     loss_save += [loss_forsaving.detach().numpy()]
237     loss_unweighted_save += [loss_forsaving_uw.detach().numpy()]
238 
239     tr_accuracy_save += [tr_accuracy]
240     test_accuracy_save += [test_accuracy]
241 
242 plt.figure()
243 plt.plot(loss_unweighted_save)
244 plt.plot(loss_save, c = 'violet')
245 plt.xlabel('epochs')
246 plt.ylabel('loss')
247 ax = plt.gca()
248 ax.set_yscale('log')
249 plt.show()
250 
251 plt.figure()
252 plt.plot(tr_accuracy_save, c = 'lightgreen', label = 'Training
    ↪ accuracy')
253 plt.plot(test_accuracy_save, label = 'Test accruacy' )
254 leg = plt.legend(loc="lower right", frameon=False)
255 plt.xlabel('epochs')
256 plt.ylabel('Accuracies')
257 
258 
259 # In[10]:
260 
261 
262 # Representación del clasificador
263 h = 0.02
264 x_min, x_max = x_data_train_zeros[:,0].min()-0.6,
    ↪ x_data_train_zeros[:,0].max()+0.6
265 y_min, y_max = x_data_train_zeros[:,1].min()-0.6,
    ↪ x_data_train_zeros[:,1].max()+0.6
266 
267 xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
268                      np.arange(y_min, y_max, h))
269 
270 Z = net(torch.tensor(np.c_[xx.ravel(), yy.ravel()], dtype=torch.float))
271 
272 Z = torch.argmax(Z, dim=1)
```

```
Z = Z.reshape(xx.shape)

fig = plt.figure()
plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral, alpha=0.8)
plt.scatter(x_data_train_zeros[:,0], x_data_train_zeros[:,1],
    ↪ c=y_data_train_zeros, s=30, cmap=plt.cm.Spectral)
plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())

plt.xlabel( 'X label')
plt.ylabel( 'Y label')


# APLICACIÓN ATAQUE ADVERSARIO

# In[11]:


import torch.optim as optim

delta = torch.zeros_like(x_data_train_zeros[50:51], requires_grad=True)
x_pred = torch.zeros_like(x_data_train_zeros[50:51])
opt = optim.SGD([delta], lr = 0.005)
epsilon = 1
y = y_data_train_zeros
print(" Coordenadas iniciales del punto:", x_data_train_zeros[50:51])
outputs_def = np.zeros([100000,2])
max_class = 0
t = 0

plt.figure()
plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral, alpha=0.8)
plt.scatter(x_data_train_zeros[50,0], x_data_train_zeros[50, 1],
    ↪ c=y[50:51], s=60, cmap = mcolors.ListedColormap(["black"]))

plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())

while  max_class == 0 or nn.Softmax(dim=1)(outputs_pred)[0,0].item() > 0
    ↪ .33:

  outputs_pred = net.forward(x_data_train_zeros[50:51]+delta)
  label =  torch.tensor(y_data_train_zeros[50:51], dtype=torch.long)
  x_pred = x_data_train_zeros[50:51]+delta
  x_pred = x_pred.detach().numpy()
  outputs_def[t] = x_pred

  loss = -nn.CrossEntropyLoss()(outputs_pred, label)

  if t % 5 == 0:
        print("Pérdida: ", t, loss.item())
        print("Probabilidades: ", outputs_pred)

  plt.scatter(x_pred[0,0], x_pred[0, 1], c=y[50:51], s=15,
    ↪ cmap=plt.cm.Spectral)
  print("Coordenadas: ", x_pred)

  opt.zero_grad()
```

```
  loss.backward()
  opt.step()
  delta.data.clamp_(-epsilon, epsilon)
  max_class = outputs_pred.argmax(dim=1)[0].item()
  t = t+1


print("True class probability:",
    ↪ nn.Softmax(dim=1)(outputs_pred)[0,0].item())
max_class = outputs_pred.argmax(dim=1)[0].item()
if max_class == 0:
  print("Predicted class: ", y_data_train_zeros[0].item())
if max_class == 1:
  print("Predicted class: ", y_data_train_zeros[200].item())
if max_class == 2:
  print("Predicted class: ", y_data_train_zeros[400].item())
print("Predicted probability:",
    ↪ nn.Softmax(dim=1)(outputs_pred)[0,max_class].item())


# In[12]:


outputs_def_zoom = np.zeros([t,2])
outputs_def_zoom = outputs_def[0:t]


# In[13]:


ycol = np.zeros([t,1])
for i in range (t):
  ycol[i] = i


# In[14]:


plt.figure()
cm1 = plt.cm.get_cmap('Reds')
plt.scatter(x_data_train_zeros[50,0], x_data_train_zeros[50, 1],
    ↪ c=y[50:51], s=60, cmap = mcolors.ListedColormap(["black"]))
plt.scatter(outputs_def_zoom[:,0], outputs_def_zoom[:,1], c=ycol, s=15,
    ↪ cmap=cm1)


# In[ ]:
```

Código A.3: Código que define la GNN.

A.4. Caso espiral 3D

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

# In[1]:


```

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm
import matplotlib.colors as mcolors
from mpl_toolkits.mplot3d import Axes3D

import pandas as pd
import numpy as np

from random import random

import torch
import torch.nn as nn
import torch.nn.functional as F


# In[2]:


np.random.seed(0)
N = 200 # number of points per class
D = 3 # dimensionality cambiamos a 3D
K = 3 # number of classes
C = K

seed = 0

w_max_list = [1, 20] # 1 corresponds to no oscillations
last_period_no_osc = 1

total_time = 30000
height = 1 # 20 #dispersion of the data along the Z coordinate

T = int(total_time/20)

time_comp_density = T  # Time step to compute Hessian eigenvalues


# In[5]:


def make_dataset_torch(points_per_class, classes, revolutions=4):
    np.random.seed(0)

    X = np.zeros((N * C, D))
    y = np.zeros((N * C))
    z = np.zeros(N)

    #%matplotlib notebook

    fig = plt.figure()
    fig.set_size_inches(7,7,7)
    ax = fig.add_subplot(111, projection = '3d')

    for j in range(K):
      ix = range(N*j,N*(j+1))
      r = np.linspace(0.0,1,N) # radius
      t = np.linspace(j*4,(j+1)*4,N) + np.random.randn(N)*0.2 # theta
```

```
65
66      for ind in range(N):
67        z[ind] = np.random.uniform(low=0.0, high=1.0)*height
68
69      X[ix] = np.c_[r*np.sin(t), r*np.cos(t), z]
70
71      y[ix] = j
72
73    fig = plt.figure()
74    ax.scatter3D(X[:, 0], X[:, 1], X[:,2], c=y, s=40,
    ↪ cmap=plt.cm.Spectral)
75    ax.set_xlabel( 'X')
76    ax.set_ylabel( 'Y')
77    ax.set_zlabel( 'Z')
78    plt.show()
79    fig = plt.figure()
80    ax.view_init(azim=0, elev=90)
81    plt.show()
82
83    return torch.tensor(X, dtype=torch.float), torch.tensor(y,
    ↪ dtype=torch.long)
84
85 def make_dataset_torch_val(points_per_class, classes, revolutions=4):
86    np.random.seed(1)
87
88    X = np.zeros((N * C, D))
89    y = np.zeros((N * C))
90    z = np.zeros(N)
91
92    #%matplotlib notebook
93    fig=plt.figure()
94    fig.set_size_inches(7,7,7)
95    ax = fig.add_subplot(111, projection = '3d')
96
97    for j in range(K):
98      ix = range(N*j,N*(j+1))
99      r = np.linspace(0.0,1,N) # radius
100     t = np.linspace(j*4,(j+1)*4,N) + np.random.randn(N)*0.2 # theta
101
102     for ind in range(N):
103       z[ind] = np.random.uniform(low=0.0, high=1.0)*height
104
105     X[ix] = np.c_[r*np.sin(t), r*np.cos(t), z]
106     y[ix] = j
107
108   fig = plt.figure()
109   ax.scatter3D(X[:, 0], X[:, 1], X[:,2], c=y, s=40,
    ↪ cmap=plt.cm.Spectral)
110   ax.set_xlabel( 'X')
111   ax.set_ylabel( 'Y')
112   ax.set_zlabel( 'Z')
113   plt.show()
114   fig = plt.figure()
115   ax.view_init(azim=0, elev=90)
116   plt.show()
117
118   return torch.tensor(X, dtype=torch.float), torch.tensor(y,
    ↪ dtype=torch.long)
```

```
119
120
121 # In[6]:
122
123
124 x_data_train_zeros,y_data_train_zeros =
    ↪ make_dataset_torch(points_per_class=N, classes=K)
125 x_data_test_zeros,y_data_test_zeros =
    ↪ make_dataset_torch_val(points_per_class=N, classes=K)
126
127
128 # In[7]:
129
130
131 # Creación del modelo de la red
132 nn_input_dim = 3
133 nn_output_dim = 3
134 w_max_osc = 1
135
136 gamma_LR = 1
137
138 nn_width = 2000
139 learning_rate = 0.1
140
141 n_epochs = 1500
142 period_osc = 100
143
144 no_osc_last_epochs = 3
145
146 Softmax_com = torch.nn.Softmax(dim=1)
147
148 class Net(nn.Module):
149
150     def __init__(self):
151
152         super(Net, self).__init__()
153         self.ly1 = nn.Linear(nn_input_dim, nn_width)
154         self.ly2 = nn.Linear(nn_width, nn_width)
155         self.ly3 = nn.Linear(nn_width, nn_output_dim)
156
157     def forward(self, x):
158         x = self.ly1(x)
159         x = F.relu(x)
160         x = self.ly3(x)
161         return x
162
163 C = nn_output_dim
164
165 def c_fn(t, i, w_max, T):
166     slope = 2 * (w_max - 1) / T
167     w_main_class = np.where(t < T / 2., 1 + t * slope, 2 * w_max - t *
    ↪ slope - 1)
168     res = np.ones(C) + (w_main_class - 1) * np.eye(C)[i]
169     res = res / np.sum(res) * C
170     return torch.tensor(res, dtype=torch.float)
171
172 c_fn(10,0,w_max_osc,20)
173 print(w_max_osc)
```

```
174
175
176 # In[8]:
177
178
179 # Entrenamiento
180 net = Net()
181
182 optimizer = torch.optim.SGD(net.parameters(), lr=learning_rate)
183 scheduler = torch.optim.lr_scheduler.ExponentialLR(optimizer,
    ↪ gamma=gamma_LR)
184
185 def calc_accuracy(mdl,X,Y):
186      max_vals, max_indices = torch.max(mdl(X),1)
187      train_acc = (max_indices ==
    ↪ Y).sum().data.numpy()/max_indices.size()[0]
188      return train_acc
189
190 loss_save = []
191 loss_unweighted_save = []
192 tr_accuracy_save = []
193 test_accuracy_save = []
194
195 tt = 0
196 ii = 0
197 cc = 0
198
199 batch_x, batch_y = make_dataset_torch(points_per_class=N, classes=K)
200
201 print('np.shape(batch_x)',np.shape(batch_x))
202 print('np.shape(batch_y)',np.shape(batch_y))
203
204 for epoch in range(n_epochs):
205
206     optimizer.zero_grad()
207
208     # in case you wanted a semi-full example
209     outputs = net.forward(batch_x)
210
211     # Oscilatorio
212     if tt > period_osc:
213
214         print('New Period ')
215         print('epoch', epoch, 'LR', optimizer.param_groups[0]['lr'],
    ↪ 'loss', loss_forsaving_uw) #, 'training  acc: ', tr_accuracy,
    ↪ 'test_accuracy: ', test_accuracy)
216
217
218         tt = 0
219         cc = (cc + 1) % nn_output_dim
220
221         if no_osc_last_epochs:
222
223             if ii > n_epochs - no_osc_last_epochs*period_osc:
224                 print('last period without oscillations')
225                 w_max_osc = 1
226
227     lossfunction = nn.CrossEntropyLoss(weight=c_fn(tt, cc, w_max_osc,
```

```
    ↪ period_osc))
228
229     loss = lossfunction(outputs,batch_y)
230
231     loss.backward()
232
233     optimizer.step()
234
235     tt += 1
236     ii += 1
237
238     scheduler.step()
239
240     tr_accuracy =
    ↪ calc_accuracy(net,x_data_train_zeros,y_data_train_zeros)
241     test_accuracy =
    ↪ calc_accuracy(net,x_data_test_zeros,y_data_test_zeros)
242
243     loss_forsaving = lossfunction(outputs,batch_y)
244
245     lossfunction = nn.CrossEntropyLoss(weight=c_fn(tt, cc, 1,
    ↪ period_osc))
246     loss_forsaving_uw = lossfunction(outputs,batch_y)
247
248     loss_save += [loss_forsaving.detach().numpy()]
249     loss_unweighted_save += [loss_forsaving_uw.detach().numpy()]
250
251     tr_accuracy_save += [tr_accuracy]
252     test_accuracy_save += [test_accuracy]
253
254
255 plt.figure()
256 plt.plot(loss_unweighted_save)
257 plt.plot(loss_save, c = 'violet')
258 plt.xlabel('epochs')
259 plt.ylabel('loss')
260 ax = plt.gca()
261 ax.set_yscale('log')
262 plt.show()
263
264 plt.figure()
265 plt.plot(tr_accuracy_save, c = 'lightgreen', label = 'Training
    ↪ accuracy')
266 plt.plot(test_accuracy_save, label = 'Test accuracy' )
267 leg = plt.legend(loc="lower right", frameon=False)
268 plt.xlabel('epochs')
269 plt.ylabel('Accuracies')
270
271 print(tr_accuracy)
272 print(test_accuracy)
273
274
275 # In[9]:
276
277
278 # Generación del clasificador
279 from mpl_toolkits import mplot3d
280
```

```

h = 0.03
x_min, x_max = x_data_train_zeros[:, 0].min(), x_data_train_zeros[:, 0
    ↪ ].max()
y_min, y_max = x_data_train_zeros[:, 1].min(), x_data_train_zeros[:, 1
    ↪ ].max()
z_min, z_max = x_data_train_zeros[:, 2].min(), x_data_train_zeros[:, 2
    ↪ ].max()
xx, yy, zz= np.meshgrid(np.arange(x_min, x_max, h),
                        np.arange(y_min, y_max, h),
                        np.arange(z_min, z_max, h))

Z = net(torch.tensor(np.c_[xx.ravel(), yy.ravel(), zz.ravel()],
    ↪ dtype=torch.float))
Z1 = torch.argmax(Z, dim=1)
Z2 = Z1.reshape(xx.shape)


# In[ ]:


# Representación clasificador 3D
import plotly.graph_objects as go
import numpy as np

fig = go.Figure(data=go.Volume(
    x=xx.flatten(),
    y=yy.flatten(),
    z=zz.flatten(),
    value=Z2.flatten(),
    isomin=0,
    isomax=2,
    opacity=0.5,
    surface_count=21,
    colorscale='Spectral'))

fig.show()


# APLICACIÓN ATAQUE ADVERSARIO A UN PUNTO DE LA ESPIRAL

# In[12]:


import torch.optim as optim

delta = torch.zeros_like(x_data_train_zeros[60:61], requires_grad=True)
x_pred = torch.zeros_like(x_data_train_zeros[60:61])
opt = optim.SGD([delta], lr = 0.0005)
epsilon = 1

y = y_data_train_zeros
outputs_def = np.zeros([100000,3])

dist_total = 0
dist_final = 0
max_class = 0
t = 0
```

```
335
336 print(" Coordenadas iniciales del punto:", x_data_train_zeros[60:61])
337 coord_in = x_data_train_zeros[60:61].detach().numpy()
338
339 fig = plt.figure()
340 fig.set_size_inches(7,7,7)
341 ax = fig.add_subplot(111, projection = '3d')
342 ax.scatter3D(x_data_train_zeros[60,0], x_data_train_zeros[60, 1],
    ↪ x_data_train_zeros[60, 2], c=y[60:61], s=60, cmap =
    ↪ mcolors.ListedColormap(["black"]))
343
344 while  max_class == 0 or nn.Softmax(dim=1)(outputs_pred)[0,0].item() > 0
    ↪ .33:
345
346   outputs_pred = net.forward(x_data_train_zeros[60:61]+delta)
347   label =  torch.tensor(y_data_train_zeros[60:61], dtype=torch.long)
348   x_pred = x_data_train_zeros[60:61]+delta
349   x_pred = x_pred.detach().numpy()
350   outputs_def[t] = x_pred
351
352   # Distancia primer desplazamiento
353   if (t==0):
354     dif = outputs_def[t] - coord_in
355     dist_total = dist_total + np.sqrt(pow(dif[0,0],2) + pow(dif[0,1],2)
     ↪ + pow(dif[0,2],2))
356
357   # Distancia desplazamientos intermedios
358   else:
359     dif = outputs_def[t] - outputs_def[t-1]
360     dist_total = dist_total + np.sqrt(pow(dif[0],2) + pow(dif[1],2) +
     ↪ pow(dif[2],2))
361
362   loss = -nn.CrossEntropyLoss()(outputs_pred, label)
363
364   if t % 10 == 0:
365         print("Pérdida", t, loss.item())
366         print("Probabilidades", outputs_pred)
367         print(nn.Softmax(dim=1)(outputs_pred)[0,max_class].item())
368
369   ax.scatter3D(x_pred[0,0], x_pred[0, 1], x_pred[0, 2], c=y[0:1], s=15,
     ↪ cmap=plt.cm.Spectral)
370   plt.xlim(-1,1)
371   plt.ylim(-1,1)
372   ax.set_zlim(x_data_train_zeros[:,2].min(),
     ↪ x_data_train_zeros[:,2].max())
373   print("Coordenadas:", x_pred)
374
375   opt.zero_grad()
376   loss.backward()
377   opt.step()
378   delta.data.clamp_(-epsilon, epsilon)
379
380   max_class = outputs_pred.argmax(dim=1)[0].item()
381   t = t+1
382
383 dif = outputs_def[t-1] - coord_in
384 dist_final = np.sqrt(pow(dif[0,0],2) + pow(dif[0,1],2) +
    ↪ pow(dif[0,2],2) )
```

```
385
386 print("True class probability:",
    ↪ nn.Softmax(dim=1)(outputs_pred)[0,0].item())
387 print("Total distancia recorrida: ", dist_total)
388 print("Desplazamiento final: ", dist_final)
389
390 max_class = outputs_pred.argmax(dim=1)[0].item()
391 if max_class == 0:
392   print("Predicted class: ", y_data_train_zeros[0].item())
393 if max_class == 1:
394   print("Predicted class: ", y_data_train_zeros[200].item())
395 if max_class == 2:
396   print("Predicted class: ", y_data_train_zeros[400].item())
397
398 print("Predicted probability:",
    ↪ nn.Softmax(dim=1)(outputs_pred)[0,max_class].item())
399
400
401 # In[13]:
402
403
404 outputs_def_zoom = np.zeros([t,3])
405 outputs_def_zoom = outputs_def[0:t]
406
407
408 # In[14]:
409
410
411 ycol = np.zeros([t,1])
412 for i in range (t):
413   ycol[i] = i
414
415
416 # In[15]:
417
418
419 # Representación desplazamiento en detalle
420 fig= plt.figure()
421 fig.set_size_inches(7,7,7)
422 ax = fig.add_subplot(111, projection = '3d')
423 cm1 = plt.cm.get_cmap('Reds')
424 ax.scatter3D(x_data_train_zeros[60,0], x_data_train_zeros[60, 1],
    ↪ x_data_train_zeros[60, 2], c=y[0:1], s=60, cmap =
    ↪ mcolors.ListedColormap(["black"]))
425 ax.scatter3D(outputs_def_zoom[:,0], outputs_def_zoom[:,1],
    ↪ outputs_def_zoom[:,2], c=ycol, s=15, cmap=cm1)
426 plt.show()
427
428
429 # APLICACIÓN ATAQUE ADVERSARIO NO DIRIGIDO A UNA MUESTRA DE 15 PUNTOS
430
431 # In[16]:
432
433
434 import torch.optim as optim
435 import random
436
437 dist_total_media = 0
```

```
dist_final_media = 0

for rep in range (15):
    punto = np.random.randint(0,600)
    coord_in = x_data_train_zeros[(punto):(punto+1)].detach().numpy()
    clase = y_data_train_zeros[punto].item()

    delta = torch.zeros_like(x_data_train_zeros[(punto):(punto+1)],
    ↪ requires_grad=True)
    x_pred = torch.zeros_like(x_data_train_zeros[(punto):(punto+1)])

    opt = optim.SGD([delta], lr = 0.0005)
    epsilon = 1
    y = y_data_train_zeros
    outputs_def = np.zeros([100000,3])
    dist_total = 0
    dist_final = 0
    dif = 0
    t = 0
    max_class = clase

    while  max_class == clase or
    ↪ nn.Softmax(dim=1)(outputs_pred)[0,clase].item() > 0.33:
       outputs_pred =
    ↪ net.forward(x_data_train_zeros[(punto):(punto+1)]+delta)
       label =  torch.tensor(y_data_train_zeros[(punto):(punto+1)],
    ↪ dtype=torch.long)
       x_pred = x_data_train_zeros[(punto):(punto+1)]+delta
       x_pred = x_pred.detach().numpy()
       outputs_def[t] = x_pred

       # Distancia primer desplazamiento
       if (t==0):
         dif = outputs_def[t] - coord_in
         dist_total = dist_total + np.sqrt(pow(dif[0,0],2) +
    ↪ pow(dif[0,1],2) + pow(dif[0,2],2))

       # Distancia desplazamientos intermedios
       else:
         dif = outputs_def[t] - outputs_def[t-1]
         dist_total = dist_total + np.sqrt(pow(dif[0],2) + pow(dif[1],2)
    ↪ + pow(dif[2],2))

       loss = -nn.CrossEntropyLoss()(outputs_pred, label)

       opt.zero_grad()
       loss.backward()
       opt.step()
       delta.data.clamp_(-epsilon, epsilon)

       max_class = outputs_pred.argmax(dim=1)[0].item()
       t = t+1

    dif = outputs_def[t-1] - coord_in
    dist_final = np.sqrt(pow(dif[0,0],2) + pow(dif[0,1],2) +
    ↪ pow(dif[0,2],2) )

    dist_total_media = dist_total_media + dist_total
```

```
    dist_final_media = dist_final_media + dist_final

    print("Repetición",rep)
    print("Punto:", punto)
    print("Clase:", clase)
    print("Total distancia recorrida: ", dist_total)
    print("Desplazamiento final: ", dist_final)
    print("\n")

distancia_total_media = dist_total_media/15
distancia_final_media = dist_final_media/15

print("Distancia total recorrida media por 15 puntos escogidos al azar:
    ↪ ", distancia_total_media)
print("Desplazamiento final medio recorrido por 15 puntos escogidos al
    ↪ azar: ", distancia_final_media)



# APLICACIÓN ATAQUE ADVERSARIO DIRIGIDO A UNA MUESTRA DE 15 PUNTOS

# In[17]:


# Se realizarán los siguientes ataques dirigidos:
# Si el punto elegido al azar es de clase 0, se modificará para que la
    ↪ red lo clasifique de clase 1
# Si el punto elegido al azar es de clase 1, se modificará para que la
    ↪ red lo clasifique de clase 2
# Si el punto elegido al azar es de clase 2, se modificará para que la
    ↪ red lo clasifique de clase 0

import torch.optim as optim
import random

dist_total_media = 0
dist_final_media = 0

for rep in range (15):
    punto = np.random.randint(0,600)
    coord_in = x_data_train_zeros[(punto):(punto+1)].detach().numpy()
    clase = y_data_train_zeros[punto].item()

    delta = torch.zeros_like(x_data_train_zeros[(punto):(punto+1)],
    ↪ requires_grad=True)
    x_pred = torch.zeros_like(x_data_train_zeros[(punto):(punto+1)])

    opt = optim.SGD([delta], lr = 0.0005)
    epsilon = 1
    y = y_data_train_zeros
    outputs_def = np.zeros([100000,3])
    dist_total = 0
    dist_final = 0
    dif = 0
    t = 0
    max_class = clase

    while  max_class == clase or
```

```
    ↪ nn.Softmax(dim=1)(outputs_pred)[0,clase].item() > 0.33:
541
542         outputs_pred =
    ↪ net.forward(x_data_train_zeros[(punto):(punto+1)]+delta)
543         label_correct =
    ↪ torch.tensor(y_data_train_zeros[(punto):(punto+1)],
    ↪ dtype=torch.long)
544
545         if (punto < 400):
546               label_wrong =
    ↪ torch.tensor(y_data_train_zeros[(punto+200):(punto+200+1)],
    ↪ dtype=torch.long)
547         else:
548             label_wrong =
    ↪ torch.tensor(y_data_train_zeros[(punto-400):(punto-400+1)],
    ↪ dtype=torch.long)
549
550
551         x_pred = x_data_train_zeros[(punto):(punto+1)]+delta
552         x_pred = x_pred.detach().numpy()
553         outputs_def[t] = x_pred
554
555         # Distancia primer desplazamiento
556         if (t==0):
557           dif = outputs_def[t] - coord_in
558           dist_total = dist_total + np.sqrt(pow(dif[0,0],2) +
    ↪ pow(dif[0,1],2) + pow(dif[0,2],2))
559
560         # Distancia desplazamientos intermedios
561         else:
562           dif = outputs_def[t] - outputs_def[t-1]
563           dist_total = dist_total + np.sqrt(pow(dif[0],2) + pow(dif[1],2)
    ↪ + pow(dif[2],2))
564
565         loss = -nn.CrossEntropyLoss()(outputs_pred, label_correct) +
    ↪ nn.CrossEntropyLoss()(outputs_pred, label_wrong)
566
567         opt.zero_grad()
568         loss.backward()
569         opt.step()
570         delta.data.clamp_(-epsilon, epsilon)
571
572         max_class = outputs_pred.argmax(dim=1)[0].item()
573         t = t+1
574
575      dif = outputs_def[t-1] - coord_in
576      dist_final = np.sqrt(pow(dif[0,0],2) + pow(dif[0,1],2) +
    ↪ pow(dif[0,2],2) )
577
578      dist_total_media = dist_total_media + dist_total
579      dist_final_media = dist_final_media + dist_final
580
581      print("Repetición",rep)
582      print("Punto:", punto)
583      print("Clase:", clase)
584      print("Total distancia recorrida: ", dist_total)
585      print("Desplazamiento final: ", dist_final)
586      print("\n")
```

```
587
588 distancia_total_media = dist_total_media/15
589 distancia_final_media = dist_final_media/15
590
591 print("Distancia total recorrida media por 15 puntos escogidos al azar:
    ↪ ", distancia_total_media)
592 print("Desplazamiento final medio recorrido por 15 puntos escogidos al
    ↪ azar: ", distancia_final_media)
593
594
595 # In[ ]:
```

Código A.4: Definitivo 3D

POLITÉCNICA

ESCUELA TÉCNICA SUPERIOR DE INGENIEROS INDUSTRIALES
UNIVERSIDAD POLITÉCNICA DE MADRID

José Gutiérrez Abascal, 2. 28006 Madrid
Tel.: 91 336 3060
info.industriales@upm.es

www.industriales.upm.es