UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE CIENCIAS MATEMÁTICAS

DEPARTAMENTO DE SISTEMAS INFORMÁTICOS Y COMPUTACIÓN

TRABAJO DE FIN DE GRADO

Verificación Formal de Redes Neuronales

Presentado por: David Periañez Rollán

Dirigido por: Rafael del Vado Vírseda

Grado en Matemáticas

Curso académico 2024-25

Agradecimientos

A mis padres, mi hermano y mis abuelos, que han estado a mi lado durante estos cuatro años y medio de carrera. Ha habido muchos momentos difíciles en los que yo no he podido estar para ellos, pero ellos siempre han estado ahí para mí.

A Sara, que debe de estar ya harta de oírme hablar sobre redes neuronales sin enterarse de nada, y aún así quiere venir a escuchar la defensa de este trabajo.

A María Gaspar, que tiene la culpa de que yo haya elegido empezar esta carrera. No me arrepiento de nada. Y a Rafa, que será con quien la termine. Fue él quien me propuso el tema, motivó y ayudó a estructurar y completar esta memoria.

A todos ellos, mi más sincero agradecimiento.

Resumen

La verificación formal de redes neuronales es un área de creciente interés dentro de la inteligencia artificial, motivada por la necesidad de garantizar propiedades críticas como la robustez en modelos de aprendizaje profundo. Este trabajo aborda el problema de la verificación desde una perspectiva teórica, estableciendo su formulación matemática y analizando los desafíos inherentes a su resolución.

Se estudian en profundidad dos enfoques principales: *alcanzabilidad* y *optimización*, explicando en detalle sus fundamentos, técnicas y algoritmos asociados. Además, se comparan distintos algoritmos dentro de cada enfoque en términos de completitud y complejidad algorítmica, identificando sus similitudes y limitaciones. Finalmente, se presentan herramientas de código abierto relevantes en el área, destacando su utilidad y desempeño.

Palabras clave: *red neuronal, problema de verificación, robustez, alcanzabilidad, optimización, completitud, algoritmo, aprendizaje profundo, función de activación, propagación, satisfactibilidad, inteligencia artifical.*

Abstract

Formal verification of neural networks is a field of growing interest within artificial intelligence, driven by the need to guarantee critical properties such like robustness in deep learning models. This work approaches the verification problem from a theoretical perspective, establishing its mathematical formulation and analyzing the challenges inherent to its resolution.

Two main approaches, *reachability* and *optimization*, are studied in depth, explaining in detail their foundations, techniques, and associated algorithms. In addition, different algorithms within each approach are compared in terms of completeness and complexity, identifying their similarities and limitations. Finally, relevant open-source tools in the area are presented, highlighting their usefulness and performance.

Keywords*: neural network, verification problem, robustness, reachability, optimization, completness, algorithm, deep learning, activation function, propagation, satisfiability, artificial intelligence.*

Índice general

1. Introducción **6**

1.1. Objetivos del trabajo 6

1.2. Contexto de la investigación 7

1.3. Estructura de la memoria 9

2. Preliminares matemáticos **10**

2.1. Redes Neuronales 10

2.2. Aprendizaje profundo 15

2.3. Funciones de activación 16

2.4. La norma lp 18

3. El problema de verificación **19**

3.1. Propiedades de corrección 19

3.2. Formulación del problema 21

3.3. Enfoques algorítmicos 22

3.4. Propiedades de los algoritmos 23

4. Algoritmos de alcanzabilidad **24**

4.1. Introducción 24

4.2. Hiperrectángulos 25

4.3. Politopos 28

4.4. ExactReach 31

4.5. *Split and Join* 34

5. Algoritmos de optimización **35**

5.1. Introducción 35

5.2. Codificación de la red neuronal 36

5.3. Reluplex 37

5.4. NSVerify y MIPVerify 39

6. Herramientas de verificación **42**

6.1. El algoritmo CROWN 42

6.2. Librería auto_LiRPA 45

6.3. Verificador alpha-beta-CROWN 47

7. Conclusiones y Trabajo Futuro **49**

7.1. Conclusiones 49

7.2. Trabajos relacionados 50

7.3. Trabajo futuro 50

Capítulo 1

Introducción

En los últimos años, el aprendizaje profundo y las redes neuronales han experimentado un crecimiento significativo en sus aplicaciones, haciéndose presentes en campos como la medicina y la ciberseguridad, donde desempeñan tareas críticas que requieren la garantía de llevarse a cabo de forma segura y correcta. Dada su creciente adopción en entornos cada vez más sensibles, surge la necesidad de desarrollar métodos que permitan verificar formalmente su funcionamiento. Este trabajo aborda el tema de la verificación formal de redes neuronales, proporcionando una visión estructurada de los métodos, algoritmos y desafíos existentes para llevarla a cabo.

En este capítulo introductorio, se comenzarán describiendo los objetivos generales y específicos de la investigación que ha dado lugar a este trabajo (1.1). A continuación, se situará el problema en el contexto de la inteligencia artificial y el aprendizaje profundo, y se plantearán las preguntas y desafíos que motivan la verificación formal de redes neuronales (1.2). Finalmente, se presentará la estructura de la memoria y el contenido de cada capítulo (1.3).

1.1. Objetivos del trabajo

En términos generales, este trabajo busca ofrecer una base teórica sólida sobre la verificación de redes neuronales, abarcando tanto su formulación matemática como las técnicas y algoritmos más relevantes para su análisis.

La verificación formal es un campo de investigación emergente que enfrenta retos significativos a causa de la complejidad y opacidad de los modelos de aprendizaje profundo. En este contexto, este trabajo se propone estructurar el conocimiento existente, proporcionando una comprensión clara de los enfoques actuales. Por tanto, se plantean los siguientes objetivos específicos:

- Justificar la necesidad de verificar formalmente redes neuronales a través de sus principales motivaciones, y analizar los desafíos teóricos y computacionales que surgen en su planteamiento y proceso.
- Definir formalmente las principales propiedades deseables en redes neuronales, y for-

mular matemáticamente el problema de verificación, estableciendo los fundamentos teóricos y las condiciones para abordarlo con rigor.

- Desarrollar en profundidad las técnicas y principios matemáticos en las que se basan los métodos de *alcanzabilidad* y *optimización*, detallando sus formulaciones matemáticas, y sus enfoques para el diseño de algoritmos.
- Explicar distintos algoritmos asociados a cada uno de los métodos, detallando su funcionamiento, y comparándolos en términos de eficiencia y complejidad algorítmica para proporcionar un análisis de sus ventajas y limitaciones.
- Presentar una visión general de herramientas *estado-del-arte* para la verificación de redes neuronales, destacando su relevancia y desempeño.

Para llevar a cabo esta investigación, se han tomado como referencias principales los libros [1] y [2], además de los artículos específicos de cada uno de los algoritmos explicados.

1.2. Contexto de la investigación

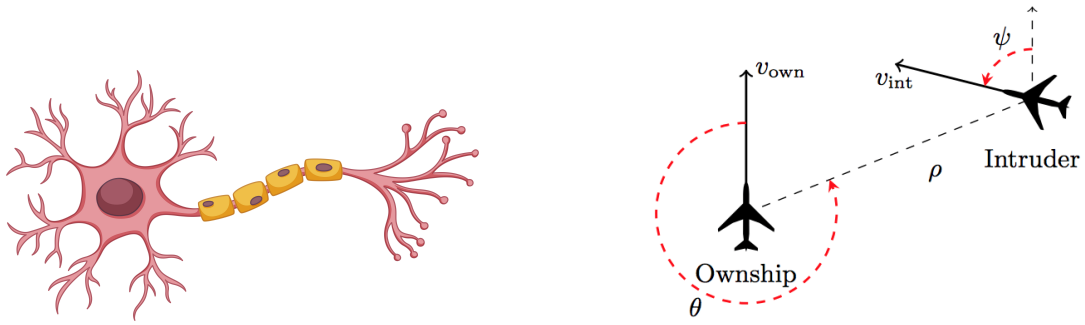
1.2.1. La inteligencia artificial y las redes neuronales

La **inteligencia artificial** (IA) es un concepto muy amplio que abarca modelos como las redes neuronales, los algoritmos genéticos y la optimización de enjambres de partículas [3]. Los primeros pasos de la IA estuvieron marcados por los algoritmos clásicos de ordenación y búsqueda, como QuickSort, MergeSort y A*, los cuales, en su momento, representaron grandes avances en la computación. Hoy en día, esos algoritmos ya no son considerados como inteligencia artificial, pues han sido reemplazados por los modelos actuales, que se caracterizan por su capacidad de aprender a partir de conjuntos de datos y adaptarse a nuevas situaciones.

Una de las ramas predominantes de la IA es el **aprendizaje profundo**, cuyos modelos se crean con el objetivo de aprender características complejas y patrones de comportamiento, que permitan posteriormente realizar predicciones, clasificaciones, e incluso tomar decisiones sobre datos nuevos en base al conocimiento adquirido. Las **redes neuronales profundas** son un tipo de modelos concreto dentro del aprendizaje profundo, que se inspiran en la biología del cerebro humano (véase Figura 1.1(a)) para imitar sus procesos de aprendizaje y tratamiento de la información. Su diseño consta de multitud de piezas básicas, las **neuronas artificiales**, conectadas entre sí en capas jerarquizadas.

Las redes neuronales tienen aplicaciones en un amplio abanico de campos, debido a su gran versatilidad y capacidad de adaptación a los problemas para los que se entrenan. Actualmente, con el incremento en la popularidad de la IA generativa, gracias a modelos como CHAT-GPT[1], Copilot[2] o Deepseek[3], el interés general por las redes neuronales va en aumento, y desde hace varios años, comienza a verse su uso en tareas como la atención médica, la detección de virus informáticos y el control de vehículos autónomos.

[1] *OpenAI.* (2025), `https://openai.com/index/chatgpt`
[2] *Microsoft Corporation.* (2025), `https://copilot.microsoft.com`
[3] *DeepSeek.* (2025), `https://www.deepseek.com`



(a) Neurona **(b)** Sistema ACAS Xu

Figura 1.1: Dibujo de una neurona biológica (a), tomada de *Freepik*[4]y diagrama del sistema basado en redes neuronales ACAS Xu (b), tomado de [4].

Todas estas son tareas críticas con mucho riesgo, que necesitan realizarse con seguridad y precisión pues, de lo contrario, las consecuencias podrían ser fatales. Surge, por tanto, la necesidad de garantizar formalmente, que las redes neuronales que se encargan de la toma de estas decisiones, cumplen realmente con su cometido, y que no existe la posibilidad de confundirlas con entradas que provoquen salidas no deseadas.

Un ejemplo clásico es el del sistema de alerta de tráfico y evasión de colisión para aeronaves no tripuladas, ACAS Xu (visto en [4]). Si detecta otro avión en una dirección, la red neuronal que lo controla deberá tomar la decisión de girar en dirección contraria para esquivarlo, y no deberá ser engañada por una nube o cualquier otra interferencia de menor criticidad, véase la Figura 1.1(b).

1.2.2. Los desafíos de la verificación formal

Desde Alan Turing y los inicios de la informática teórica, ha surgido la necesidad de comprobar que los algoritmos propuestos realmente resolvían los problemas para los que habían sido diseñados [1], naciendo así el campo de la verificación formal. Por tanto, es una consecuencia natural preguntarse de qué manera se puede verificar la corrección de los modelos actuales de inteligencia artificial.

En 1967, Robert Floyd introdujo métodos de verificación formal basados en invariantes de bucle [5] y, poco después, Tony Hoare estableció el esquema de precondición-postcondición para la verificación de la corrección de programas [6]. Estos avances sentaron las bases de los métodos empleados en la actualidad, los cuales consisten en formalizar, mediante lógica de Hoare, la especificación de un algoritmo, y demostrar, a través de una serie de pasos, condiciones e invariantes, que su implementación cumple dicha especificación.

Sin embargo, aquí surge el primer desafío de la verificación de redes neuronales: las tareas para las que han sido diseñadas, en muchos casos, no pueden expresarse explícitamente en forma matemática. Especificar un algoritmo que calcule factoriales es sencillo, pero ¿cómo se formaliza matemáticamente la acción de *reconocer patologías en una radiografía* o *conducir un vehículo de forma segura*? Las redes neuronales son tan útiles precisamente

[4] *Freepik* (2025), `https://www.freepik.es`

por este motivo: permiten resolver problemas que serían imposibles de abordar mediante algoritmos tradicionales [1].

Otro de los desafíos es la magnitud del espacio de entrada. Las redes neuronales reciben parámetros de entrada de espacios virtualmente infinitos, lo que hace inviable (y del todo inútil) comprobar cada una de las posibles configuraciones. Además, echar un vistazo al interior de una red entrenada tampoco arroja mucha luz; es lo que se conoce como el problema de la caja negra (o *black box*, en inglés), en referencia a la dificultad de interpretar y comprender el funcionamiento interno de modelos complejos [1].

Finalmente, se encuentran los desafíos técnicos y teóricos, como la capacidad de cómputo de los ordenadores actuales, y la búsqueda de la precisión en la verificación. Algunos algoritmos existentes son capaces de verificar ciertas propiedades de redes neuronales, sin embargo, su elevada complejidad algorítmica impide que sean escalables a redes con un gran número de neuronas [2]. Por otro lado, como veremos en el desarrollo de los métodos de verificación, la no linealidad representa el principal reto teórico para alcanzar la completitud en las verificaciones.

1.3. Estructura de la memoria

La presente memoria se estructura en los siguientes capítulos:

- El **Capítulo 2** introduce los conceptos matemáticos fundamentales sobre redes neuronales y aprendizaje profundo, estableciendo la notación matemática empleada en el resto de la memoria. Se estudia el funcionamiento de la neurona artificial y del perceptrón multicapa, y se describen las principales arquitecturas de redes neuronales profundas. También se define la normal vectorial, una herramienta que resultará de utilidad en los capítulos posteriores.
- En el **Capítulo 3** se exponen los aspectos formales del problema de la verificación. Se proponen definiciones de las propiedades deseables para las redes neuronales, y se formula a partir de ellas el esquema matemático del problema de verificación formal. Después, se indican los principales enfoques de resolución que existen, y se definen las propiedades de los algoritmos de verificación.
- Los Capítulos 4 y 5 se centran en desarrollar, en detalle, los principales métodos de resolución de problemas de verificación. En el **Capítulo 4** se trata el método *de alcanzabilidad*, y en el **Capítulo 5**, el de *optimización*. En cada uno de ellos se explican y comparan varios algoritmos, que servirán como ejemplo de las diferentes técnicas que aborda cada método.
- En el **Capítulo 6** se exploran herramientas de código diseñadas para la verificación de redes neuronales, las cuales implementan algoritmos basados en las ideas vistas en capítulos anteriores, y sirven como punto de referencia para conocer el *estado-del-arte* de la verificación formal de redes neuronales.
- Por último, en el **Capítulo 7** se presentan las conclusiones finales de la memoria, el trabajo futuro y algunos trabajos publicados relacionados con la investigación realizada.

Capítulo 2

Preliminares matemáticos

Este capítulo comenzará con una definición de red neuronal, seguido por un recorrido del marco teórico desde el perceptrón simple hasta el perceptrón multicapa, estableciendo la notación matemática utilizada en el resto del trabajo (2.1). A continuación, se ofrecerá una visión general del aprendizaje profundo y sus principales arquitecturas, como las redes convolucionales (CNN), recurrentes (RNN) y *Transformers* (2.2). Luego, se expondrán las funciones de activación más utilizadas en redes neuronales, con un enfoque que facilite su aplicación en algoritmos de verificación (2.3). Por último, se definirá la p-norma, herramienta que resultará de gran utilidad en los próximos capítulos (2.4).

La elaboración de este capítulo se ha basado principalmente en [7, 1, 2]. Se ha seguido la notación propuesta en [2].

2.1. Redes Neuronales

El concepto de **red neuronal** (NN, del inglés, *Neural Network*) abarca cualquier modelo computacional formado por nodos, o *neuronas*, interconectados que realizan cálculos matemáticos sencillos, y que comparten información entre ellos para aprender patrones complejos a partir de grandes volúmenes de datos. En esta sección nos centraremos en el perceptrón multicapa (MLP, del inglés, *Multi Layer Perceptron*) y en su definición y notación matemática. Para ello, se tomará como punto de partida el perceptrón simple, un algoritmo sencillo pero fundamental en el desarrollo de las redes neuronales, y se estudiará cómo se generaliza hasta alcanzar el MLP.

2.1.1. La neurona artifical

En 1958, Frank Rosenblatt introdujo el concepto de **perceptrón simple** [8], el modelo más básico de red neuronal artificial. Se trata de un clasificador binario basado en aprendizaje supervisado capaz de resolver problemas linealmente separables; es decir, dado un elemento puede decidir a cuál de entre dos grupos pertenece siempre que dichos grupos no estén *mezclados* entre sí.

La entrada del perceptrón simple es un vector x de n valores $(x_1, x_2, \ldots, x_n)$ que representa las diferentes características de un elemento como un punto en un espacio de n dimensiones. Su salida, $y \in \{0, 1\}$, será la predicción del grupo al que pertenece el elemento x, donde cada uno de los posibles valores, 0 o 1, representan la etiqueta que identifica al grupo al que ha sido clasificada la entrada x.

Para realizar la clasificación, el perceptrón realiza dos fases principales: una **combinación lineal** y una **activación**. En la primera fase, la combinación lineal consiste en una ponderación de las características x_i mediante valores de *peso* w_i, que representan la importancia relativa de cada característica. Después, se le añade un valor de *sesgo* b:

$$z = \sum_{i=1}^{n} w_i \cdot x_i + b \tag{2.1}$$

Para la segunda fase, la activación se realiza con la función escalonada (ecuación 2.2), que asigna a valores negativos la etiqueta 0, y a valores positivos la etiqueta 1, quedando así la entrada clasificada en uno de los dos grupos.

$$y = f(z) = \begin{cases} 1 & \text{si } z > 0 \\ 0 & \text{si } z \leqslant 0 \end{cases} \tag{2.2}$$

Una representación del perceptrón simple puede verse en la Figura 2.1.

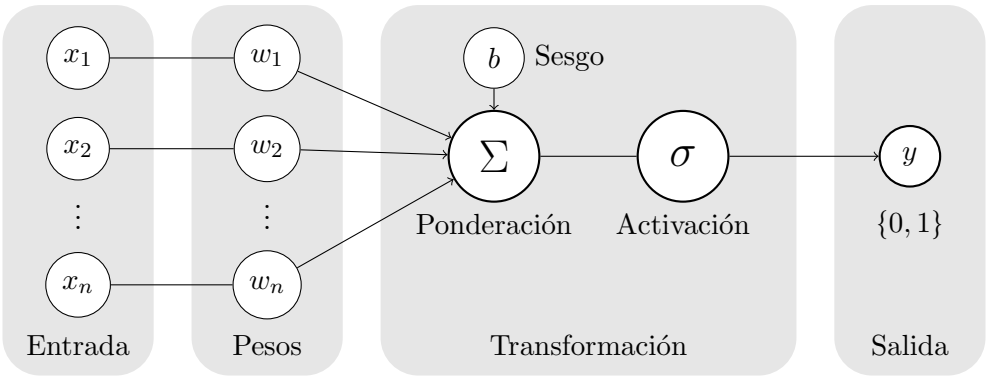


Figura 2.1: Diagrama extensivo de la neurona artificial. En futuras representaciones, cada neurona aparecerá como un único nodo por simplicidad. Elaboración propia.

Lo que hace el perceptrón, en realidad, es trazar una línea (o, en altas dimensiones, un hiperplano) para dividir el espacio $\mathbb{R}^n$ en dos subespacios, asignando valores 0 o 1 a las entradas, dependiendo del lado del hiperplano en el que se encuentren. Para construir este hiperplano correctamente, el perceptrón debe ser entrenado con un conjunto de datos (*dataset*) cuya clasificación sea conocida, de ahí el término *aprendizaje supervisado*.

Durante el entrenamiento, el perceptrón ajusta los pesos w_i y el sesgo b en pequeñas cantidades, en función de si las predicciones realizadas sobre las entradas de prueba son correctas o no, modificando gradualmente la posición del hiperplano separador. Cuando el error de las predicciones está por debajo de un umbral predefinido, o se alcanza un límite establecido de iteraciones, el proceso termina y el perceptrón se considera entrenado.

Las limitaciones del perceptrón simple son evidentes: solo es capaz de clasificar en dos conjuntos, y solo si estos son linealmente separables, véase la Figura 2.2(a).

(a)
Conjunto linealmente separable con hiperplano separador

(b)
Conjunto no linealmente separable, similar a la puerta XOR.

Figura 2.2: Ejemplos de dos conjuntos de puntos. Elaboración propia.

Ejemplo 2.1.1. *Un ejemplo clásico de problema sencillo que excede las capacidades del perceptrón simple es la puerta XOR, donde es imposible separar correctamente los dos conjuntos con una sola línea recta. Una representación del problema se encuentra en la Figura 2.2(b).*

Un primer paso para resolver estos problemas y ampliar las capacidades del perceptrón consiste en generalizarlo, obteniendo así mayor control sobre su salida mediante la definición de distintas **funciones de activación**. Esta versión, más general, se conoce como **neurona artificial**, y será el bloque básico de construcción de las redes neuronales de cualquier tipo. Las funciones de activación se verán en detalle en la Sección 2.3.

2.1.2. El perceptrón multicapa (MLP)

Las funciones de activación son fundamentales, pero es al conectar unas neuronas con otras cuando se alcanza el máximo potencial de ellas.

Definición 2.1.2. *Un **perceptrón multicapa** (MLP) es un modelo computacional formado por neuronas artificiales conectadas entre sí, organizadas en capas jerárquicas que forman una red. Los MLP se caracterizan por tener al menos dos capas: una de entrada y otra de salida, a las que se pueden añadir una o más capas intermedias, llamadas capas ocultas.*

Las redes neuronales que se utilizarán en este trabajo serán MLP **totalmente conectados** (*fully connected networks*) y de **propagación directa** (*feedforward neural networks*). Esto significa que todas las neuronas de una capa i reciben el mismo vector de entrada z_{i-1}, y sus salidas forman el vector de salida z_i, que será enviado a su vez a la siguiente capa, y así sucesivamente. Además, la información fluye siempre en una única dirección: desde la capa de entrada hasta la capa de salida, sin formar ciclos ni retroalimentaciones.

Observación 2.1.3. *Los MLP son un tipo de red neuronal, concepto más amplio que abarca todo tipo de arquitecturas (configuraciones de neuronas) como CNN, RNN o Transformers. Se profundizará en ello en la Sección 2.2.*

La combinación lineal y la activación que realiza cada neurona son operaciones que pueden generalizarse al nivel de una capa completa; veamos cómo. La i-ésima capa de una red neuronal, que cuenta con k_i neuronas, recibe un vector de entrada z_{i-1} de tamaño k_{i-1}, proporcionado por la capa anterior. Cada neurona $j \in \{1, 2, \ldots, k_i\}$ realiza, por separado, una combinación lineal con los pesos $\{w_{i,j,1}, w_{i,j,2}, \ldots, w_{i,j,k_{i-1}}\}$ y el sesgo $b_{i,j}$ que, de aplicarse todas juntas, pueden interpretarse como una única *transformación afín*; expresando todos los pesos en una matriz $W_i \in \mathbb{R}^{k_i \times k_{i-1}}$ y los sesgos en un vector $b_i \in \mathbb{R}^{k_i \times 1}$, podemos representarla como

$$\begin{pmatrix} \hat{z}_{i,1} \\ \hat{z}_{i,2} \\ \vdots \\ \hat{z}_{i,k_i} \end{pmatrix} = \begin{pmatrix} w_{i,1,1} & w_{i,1,2} & \ldots & w_{i,1,k_{i-1}} \\ w_{i,2,1} & w_{i,2,2} & \ldots & w_{i,2,k_{i-1}} \\ \vdots & \vdots & \ddots & \vdots \\ w_{i,k_i,1} & w_{i,k_i,2} & \ldots & w_{i,k_i,k_{i-1}} \end{pmatrix} \cdot \begin{pmatrix} z_{i-1,1} \\ z_{i-1,2} \\ \vdots \\ z_{i-1,k_{i-1}} \end{pmatrix} + \begin{pmatrix} b_{i,1} \\ b_{i,2} \\ \vdots \\ b_{i,k_i} \end{pmatrix} \tag{2.3}$$

A continuación, la activación dada por la función σ_i se realiza coordenada a coordenada sobre el vector $\hat{z}_i$. Cada capa puede representarse, por tanto, mediante la función f_i, que abreviamos como:

$$z_i = f_i(z_{i-1}) = \sigma_i(\hat{z}_i) = \sigma_i(W_i z_{i-1} + b_i) \tag{2.4}$$

El vector de entrada de una red neuronal de n capas se denota como $x = z_0$, y el de salida como $y = z_n$. El vector resultado de la capa oculta i previo a la activación se denota como $\hat{z}_i$ y, tras la activación, como z_i. La letra k_i se reserva para referirse al número de neuronas de la i-ésima capa.

Definición 2.1.4. *Una **red neuronal** es una función $f : \mathbb{R}^{k_0} \longrightarrow \mathbb{R}^{k_n}$, en general no lineal y no convexa, formada por la composición de las funciones de cada una de las n capas que la forman:*

$$f = f_n \circ f_{n-1} \circ \cdots \circ f_1 \tag{2.5}$$

donde cada capa f_i es la transformación dada por la ecuación 2.4. Un diagrama de red neuronal, con la notación empleada en esta memoria, se encuentra en la Figura 2.3.

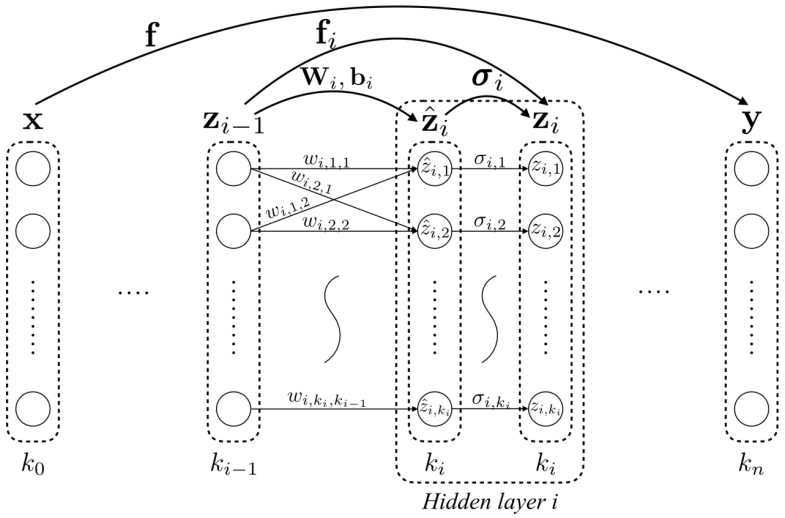


Figura 2.3: Representación de una red neuronal con la notación empleada. Tomada de [2].

2.1.3. Aprendizaje y retropropagación de errores

En 1958, Frank Rossenblatt propuso el algoritmo de entrenamiento del perceptrón simple [8]. Sin embargo, debido a sus limitaciones (señaladas en el libro *Perceptrons* [9] de Minsky y Papert en 1969), el interés por las redes neuronales disminuyó considerablemente. No fue hasta 1986 cuando Rumelhart, Hinton y Williams desarrollaron el algoritmo de **retropropagación de errores** (*backpropagation*) [10], gracias al cual se pueden entrenar redes con capas ocultas.

Para entrenar una red neuronal $f : \mathbb{R}^{k_0} \longrightarrow \mathbb{R}^{k_n}$, se necesitan los siguientes elementos:

- Un conjunto de **datos de entrenamiento** $E = \{(x_i, y_i) \mid x_i \in X,\ y_i \in Y\}$, donde $X \subset \mathbb{R}^{k_0}$ es un conjunto de entradas y $Y \subset \mathbb{R}^{k_n}$ sus correspondientes salidas correctas.
- Una **función de coste** C que mide el error entre la salida predicha $\tilde{y}$ de la red y la salida esperada y.
- Una **tasa de aprendizaje** $\eta \in \mathbb{R}$, que determinará la magnitud de los ajustes en los pesos. Una tasa demasiado baja hará que la red tarde mucho en aprender; y una demasiado alta hará que pierda precisión en el ajuste y pueda no llegar a converger.
- Un **algoritmo de optimización**, o aprendizaje, basado en el descenso del gradiente y en la retropropagación de errores.

Las redes se inicializan con pesos w aleatorios, y el objetivo del entrenamiento es ajustarlos para minimizar la función de coste. Para ello, se realizan múltiples iteraciones, o *épocas* n_e, sobre el conjunto de entrenamiento E, ajustando los pesos en función del error cometido. El funcionamiento es el mismo que para el perceptrón, sin embargo, realizar los cálculos para una red profunda se vuelve mucho más complejo.

La función de coste C, en general, es *no convexa*, por lo que calcular su mínimo global no es un problema trivial. Para minimizar C se hace uso del *descenso de gradiente* [7], que actualiza los pesos W en la dirección del gradiente negativo:

$$W' = W - \eta \nabla C(W) \tag{2.6}$$

donde $\nabla C(W)$ es el gradiente de la función de coste respecto a los pesos de la red. El problema es que, el cálculo de las derivadas parciales de C respecto a cada peso w, es computacionalmente muy costoso.

Aquí es donde entra en juego la retropropagación de errores, que aplica la regla de la cadena para calcular eficientemente los gradientes de cada capa, desde la última hasta la primera, en un proceso recursivo que aprovecha los cálculos de etapas anteriores para reducir significativamente el coste computacional del entrenamiento.

Las cuestiones teóricas y técnicas del algoritmo de *backpropagation* quedan fuera del objetivo de esta memoria, pues para la verificación se partirá de redes ya entrenadas. El lector interesado puede encontrar más información en [7, 10]. En la práctica, existen optimizaciones y mejoras tanto teóricas como computacionales realizadas a este algoritmo que lo hacen más eficiente.

2.2. Aprendizaje profundo

La neurona artificial es una herramienta muy versátil y escalable gracias a su simplicidad y capacidad de conexión. Existen distintos tipos de arquitecturas desarrolladas para cumplir propósitos específicos y resolver ciertos problemas de forma asequible. En esta sección se dará una visión general de algunos de los más relevantes [7].

Definición 2.2.1. *Se llama **red neuronal profunda** (DNN, del inglés Deep Neural Network) a cualquier arquitectura de red neuronal que cuente con múltiples capas ocultas.*

Definición 2.2.2. ***Arquitecturas de redes neuronales profundas**:*

*a) Una **red neuronal convolucional** (CNN) es un modelo de NN diseñado para procesar datos espaciales de alta dimensionalidad con estructura de cuadrícula, como imágenes. Su arquitectura se compone de capas de muestreo (pooling) para reducir el tamaño de las entradas, capas convolucionales que aplican filtros y extraen características específicas, y capas densamente conectadas para la clasificación final [7]. Aplicaciones destacables de las CNN son el reconocimiento facial, la clasificación de imágenes o la detección de objetos en estas. DenseNet[1] y ResNeXt[2] son dos ejemplos concretos muy utilizados.*

*b) Una **red neuronal recurrente** (RNN) es una arquitectura de red neuronal diseñada para procesar datos secuenciales, como listas de palabras. Esto se consigue con bucles en su estructura. Sus principales aplicaciones son el procesado de lenguaje natural (NLP) o la predicción de datos a partir de series temporales.*

*c) El **Transformer** [11] es un tipo de arquitectura de DNN que utiliza mecanismos de atención para procesar secuencias de datos, mejorando así la capacidad de generar respuestas que requieren contexto sin necesidad de recurrencias. Sus estructuras principales son un encoder, que procesa las entradas reteniendo el contexto de cada palabra; y un decoder, que genera una salida palabra a palabra utilizando el contexto extraído por el encoder. Al igual que las RNN se emplean principalmente para NLP, y toman como entrada secuencias de datos (palabras). Cada palabra es representada internamente por el algoritmo mediante un vector de alta dimensionalidad, que permite codificarla dentro de un contexto. Modelos como ChatGPT o Google Gemini[3] están basados en Transformers.*

*d) Una **red de generación adversarial** (GAN) consiste en dos redes neuronales que compiten entre sí: una generadora de datos sintéticos y otra discriminadora que valora la autenticidad de los datos. Son utilizadas para crear datos o información artificial a partir de un conjunto de datos real. La creación de imágenes fotorealistas, deepfakes o pistas de sonido se hace con este tipo de redes.*

En este trabajo, por razones de simplicidad, los algoritmos de verificación estudiados se aplican únicamente a MLP, aunque muchos de ellos son generalizables a cualquier tipo de DNN [1, 2].

[1] `https://paperswithcode.com/method/densenet`
[2] `https://paperswithcode.com/method/resnext`
[3] *Google, INC.* (2025), `https://gemini.google.com`

2.3. Funciones de activación

En la Sección 2.1.1 se estableció la importancia de las funciones de activación. Su papel principal es el de introducir *no-linealidad* en los modelos, lo que resulta fundamental para que los algoritmos puedan aprender patrones complejos y diferenciarse de las regresiones lineales tradicionales.

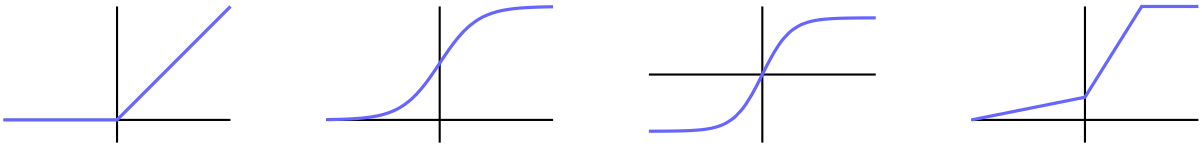
Sin la activación, cada capa de neuronas realizaría únicamente una transformación afín. Como vimos en la ecuación 2.5, una red neuronal es composición de la función de cada una de sus capas. Dado que la composición de funciones afines es también afín, una red neuronal sin activaciones se reduciría a una simple transformación afín, quedando su capacidad limitada a la aproximación de funciones lineales. Es decir, nada nuevo que permita clasificarlas como algoritmos de inteligencia artificial.

El propósito de la activación va más allá de introducir no-linealidad. Dependiendo de la función escogida, se puede restringir la salida a rangos específicos de valores que faciliten la interpretación y el entrenamiento. Algunas funciones incluso permiten activar o desactivar neuronas, influyendo en la capacidad de representación y eficacia del modelo.

Algunas de las funciones de activación más utilizadas son [7]:

- **ReLU** (*Rectified Linear Unit*), definida como $\text{relu}(z) = \text{máx}(0, z)$. Se trata de la función más utilizada por su sencillez y eficacia.
- ***Sigmoide***, cuya expresión es $\sigma(z) = \frac{1}{1+e^{-z}}$. Su rango es $(0, 1)$, lo que la hace útil para suavizar las salidas y facilitar su interpretación como función de probabilidad.
- ***Tangente hiperbólica***, definida como $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$. Es similar a la *sigmoide* pero su rango es $(-1, 1)$.
- ***Softmax***, dada por $\text{softMax}(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}}$. Su rango también es $(0, 1)$, pero cuenta con la particularidad de que la suma de todas las activaciones en la capa es igual a 1. Esto la hace especialmente útil en la última capa de redes de clasificación.

Un diagrama de cada una de ellas se muestra en la Figura 2.4.



(a) ReLU **(b)** Sigmoide **(c)** tanh **(d)** PWL

Figura 2.4: Funciones de activación. Elaboración propia.

En este trabajo, prestaremos especial atención a la función ReLU, pues todos los algoritmos de verificación que se estudiarán en los Capítulos 4, 5 y 6 asumen su uso.

Esto es debido a que la ReLU pertenece a una familia de funciones conocidas como *lineales a trozos* (PWL, por sus siglas en inglés: *PieceWise Linear*) [1].

Definición 2.3.1. *Una **función lineal a trozos (PWL)** es aquella definida mediante una serie de intervalos disjuntos posiblemente abiertos o no acotados,* $[z_0, z_n]$*, que forman*

una partición de su domino de modo que la expresión de cada trozo es una función lineal o afín:

$$f(z) = \begin{cases} a_1 z + b_1 & si\ z \in [z_0, z_1] \\ a_2 z + b_2 & si\ z \in [z_1, z_2] \\ \vdots & \\ a_n z + b_n & si\ z \in [z_{n-1}, z_n] \end{cases} \tag{2.7}$$

donde a_i y b_i son los coeficientes de cada uno de los n trozos. Una representación de función PWL se encuentra en la Figura 2.4(d).

En la práctica, de cara a la verificación, es recomendable que las funciones de activación de la red sean PWL, ya que la representación fiel de funciones como la *sigmoide* o la *tanh* es computacionalmente inviable, demasiado costoso o, en algunos casos, incompatible con los algoritmos de verificación.

Como veremos en los Capítulos 4 y 5, la dificultad de verificar redes neuronales se debe en gran medida a la naturaleza de las funciones de activación. La no-linealidad es esencial para la capacidad expresiva de la red, pero resulta un desafío significativo en el momento de la verificación [1].

La forma en la que se tratarán las funciones no PWL será mediante una sobreaproximación. Sea $f : \mathbb{R} \longrightarrow \mathbb{R}$ una función no lineal, no PWL y monótona. Asumamos, sin pérdida de generalidad, que es creciente. Sean $c_1 < c_2 < \cdots < c_n$ una secuencia de valores arbitrarios del dominio de f, y l y u los límites inferior y superior de su rango, respectivamente. La sobreaproximación se define de la siguiente manera:

$$f^*(x) \equiv \begin{cases} l < f^*(x) \leqslant f(c_1) & \text{si } x \leqslant c_1 \\ f(c_1) < f^*(x) \leqslant f(c_2) & \text{si } c_1 < x \leqslant c_2 \\ \vdots & \\ f(c_n) < f^*(x) < u & \text{si } c_n < x \end{cases} \tag{2.8}$$

Lo que se consigue con esto es cubrir la función con rectángulos de modo que cada entrada $x \in [c_i, c_{i+1}]$ pueda adoptar cualquier valor de salida dentro de $[f(c_i), f(c_{i+1})]$. Esto es posible gracias a que la monotonía asegura que, si $c_i < c_{i+1}$ entonces $f(c_i) < f(c_{i+1})$, asegurando que no se queda ningún valor sin cubrir.

Ejemplo 2.3.2. *Sea f la función tangente hiperbólica (creciente, no PWL). Realicemos su sobreaproximación escogiendo, como puntos del dominio, $c_1 = -0{,}5$ y $c_2 = 0{,}5$. Tenemos que $tanh(-0{,}5) = -0{,}46$ y $tanh(0{,}5) = 0{,}46$. Además, sabemos que $l = -1$ y que $u = 1$.*

$$y = tanh^*(x) \equiv \begin{cases} -1 < y \leqslant -0{,}46 & si\ x \leqslant -0{,}5 \\ -0{,}46 < y \leqslant 0{,}46 & si\ -0{,}5 < x \leqslant 0{,}5 \\ 0{,}46 < y < 1 & si\ 0{,}5 < x \end{cases}$$

Puede verse una visualización en la Figura 2.5. Cuantos más puntos del dominio para hacer intervalos se tomen, más ajustada será la aproximación, pero más costoso computacionalmente será manejar la función.

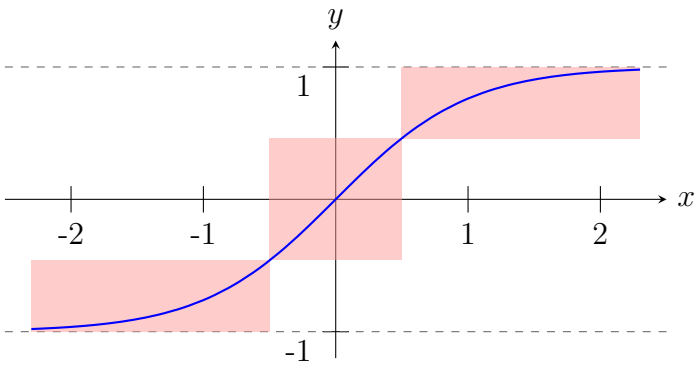


Figura 2.5: Función tangente hiperbólica aproximada por rectángulos con puntos escogidos en $c_1 = -0{,}5$ y c_2=0.5. Elaboración propia.

La técnica de sobreaproximación (ecuación 2.8) ha sido tomada de [1] y adaptada. El ejemplo con *tanh* es de elaboración propia, basado en los ejemplos proporcionados en el mismo artículo.

Al sobreaproximar funciones utilizando este método, se están teniendo en cuenta puntos que no pertenecen a la función original, añadiendo así comportamientos extra al análisis. Esto provocará que, cualquier algoritmo que haga estas aproximaciones, pierda la completitud. La aproximación, además, debe ser siempre *por exceso*, y cubrir por completo la función, o de lo contrario se perdería la seguridad. Veremos más sobre este aspecto en la Sección 3.4 y en las descripciones de cada algoritmo.

2.4. La norma l_p

Definición 2.4.1. *Sea L^p un espacio vectorial normado, es decir, un espacio vectorial Z sobre el que se define la función* ***norma*** $||\cdot||_p : Z \longrightarrow \mathbb{R}$ *como*

$$||z||_p = \left(\sum_i |z_i|^p \right)^{\frac{1}{p}} \tag{2.9}$$

Ejemplo 2.4.2. *Dependiendo de la norma utilizada, la distancia definida tiene un significado u otro [1]. Las más utilizadas son:*

- ***Norma euclídea*** l_2*:* $||z||_2 = \sqrt{\sum_i |z_i|^2}$*. Si dos imágenes distan* 1 *en esta norma, significa que el total de las diferencias entre ambas fotos no excede* 1*. Es decir, se permiten grandes variaciones en unos pocos píxeles o pequeñas en muchos.*
- ***Norma Manhattan*** l_∞*:* $||z||_\infty = \max_i |z_i|$*. Mide la discrepancia máxima entre dos puntos. Si dos imágenes distan* 0,1 *en esta norma, cada píxel varía, como máximo,* 0,1 *de su homólogo en la otra imagen. Es decir, solo se permiten pequeñas variaciones, pero en todos los píxeles.*

En esta memoria, se hará uso de la norma l_p para la definición de regiones en los espacios de entrada y salida de una red neuronal mediante p-bolas de centro x_0 y radio ε:

$$B_p(x_0, \varepsilon) = \{x \mid ||x - x_0||_p \leqslant \varepsilon\} \tag{2.10}$$

Capítulo 3

El problema de verificación

El objetivo de este capítulo es establecer un marco teórico para la formulación de problemas de verificación, que sirva como punto de partida para el desarrollo de las principales líneas de diseño de algoritmos destinados a resolverlos. Para ello, se comenzarán identificando algunas propiedades de corrección (3.1) deseables en una red, como la robustez o la invariancia, entre otras. Luego, se presentará una definición formal del problema de verificación (3.2), siguiendo el esquema clásico de precondición-postcondición. A partir de esa base teórica, se propondrán los dos principales enfoques de resolución (3.3): alcanzabilidad y optimización, que se verán en detalle en los Capítulos 4 y 5, respectivamente. Por último, se analizarán las propiedades clave (3.4) que determinan la efectividad de estos algoritmos: la seguridad y la completitud.

El contenido de este capítulo es una síntesis propia hecha a partir de la información encontrada en [1, 2, 12, 13]. En particular, la idea de la formulación del problema en la Sección 3.2 es una combinación del principio teórico propuesto en [1] con el enfoque práctico que se sigue en [2].

3.1. Propiedades de corrección

Las propiedades de corrección son aquellas condiciones que debe cumplir una red neuronal para asegurarnos de que realiza correctamente la función para la que ha sido diseñada y entrenada. Por ejemplo, en una red que clasifica imágenes de animales, pretendemos que no se etiquete como «*león*» la imagen de una cebra. El problema es que, definir en lenguaje formal estas propiedades, no es una tarea sencilla [1]: de igual manera que nos es imposible codificar matemáticamente la función de *clasificar imágenes de animales*, nos es imposible codificar la capacidad de *hacerlo bien*.

No obstante, es posible identificar ciertas propiedades deseables (o indeseables) que pueden resultar de utilidad para decidir si una red neuronal cumple o no su propósito. Es importante destacar que estas propiedades deben entenderse como conceptos generales, y no como definiciones estrictas, pues han sido ampliamente discutidas en la literatura y no existe un consenso absoluto sobre su formulación [1]. Las definiciones de esta sección son de elaboración propia.

3.1.1. Robustez

Definición 3.1.1. *Decimos que una red es **robusta** cuando pequeñas perturbaciones en los parámetros de entrada no dan lugar a grandes cambios en los de salida. Una red neuronal f es **robusta** alrededor de x_0 cuando, fijada x_0 una entrada de la red y dada una perturbación $\varepsilon > 0$, las entradas x en un entorno de x_0 y la p-bola $||x - x_0||_p < \varepsilon$, cumplen que $||f(x) - f(x_0)||_p < \delta$, siendo $\delta > 0$ la tolerancia permitida en la salida.*

Observación 3.1.2. *La comprobación de la robustez ha de hacerse sobre una entrada x_0 fijada, por lo que se trata de una propiedad local; además, no tendría sentido definirla a nivel global, pues estaríamos pidiendo que todas las entradas x de una red tuviesen la misma salida. Cuando decimos que una red es robusta, nos referimos a que, en general, resiste bien ataques adversariales, es decir, entradas que buscan engañar a la red con perturbaciones sutiles.*

Ejemplo 3.1.3. *Sea f una red de reconocimiento de imágenes médicas. Si sabemos que f ha clasificado la imagen de una herida, x_0, con la etiqueta «grave», y se aumenta levemente el contraste o se cambia el color de algunos píxeles, querremos que f continúe clasificando estas nuevas imágenes x también como «grave», y no como «leve».*

3.1.2. Invariancia

Definición 3.1.4. *Los criterios de **invariancia** establecen que la salida de una red debe permanecer inalterada bajo ciertas transformaciones permitidas. La red neuronal f será **invariante** por la transformación τ cuando $f(\tau(x)) = f(x)$, $\forall x$.*

Ejemplo 3.1.5. *En la red neuronal médica f del ejemplo anterior, rotar la fotografía de una herida, o trasladarla cierto vector, no debe dar lugar a interpretaciones diferentes.*

Observación 3.1.6. *En este caso, la invariancia, sí es una propiedad que buscamos a nivel global.*

3.1.3. Alcanzabilidad

Cuando se habla de alcanzabilidad para una red neuronal f, surgen dos conceptos: uno es la propiedad de una región del espacio de salida de ser alcanzada por f, mientras que el otro es la región del espacio de salida que se alcanza partiendo de determinadas entradas:

Definición 3.1.7. ***Alcanzabilidad***

*a) Una región $\mathcal{Y}$ del espacio de salida **es alcanzable** si $\forall y \in \mathcal{Y}, \exists x$ tal que $f(x) = y$.*

*b) Dado un conjunto de entradas $\mathcal{X}$, la **región alcanzable** $\mathcal{R}$ (del inglés, reachable) será el conjunto $\{y \mid y = f(x);\ x \in \mathcal{X}\}$.*

Ejemplo 3.1.8. *Sea f la red que controla un robot cirujano. Se quiere tener la certeza de que, sean cuales sean los parámetros que el robot reciba como entrada, este siempre podrá llegar a una solución segura. Esto lo podemos comprobar definiendo cuales son las salidas seguras, $\mathcal{Y}$, y viendo que son alcanzables desde cualquier configuración de parámetros; o probando que dados un conjunto de parámetros $\mathcal{X}$, la región alcanzable $\mathcal{R}$ está dentro de lo considerado seguro.*

3.1.4. Monotonía

Definición 3.1.9. *Una red es **monótona** cuando valores mayores de entrada producen valores mayores de salida, es decir, dadas dos entradas x y x', si $x > x'$ entonces $f(x) \geqslant f(x')$. La monotonía puede querer buscarse a nivel global, o solo en ciertas regiones del espacio de entrada.*

Ejemplo 3.1.10. *Un algoritmo f de predicción de complicaciones médicas debería devolver un mayor nivel de riesgo cuanto mayor sea la edad del paciente.*

Existen otras propiedades interesantes, como la **seguridad** o la **consistencia**, pero sus definiciones tienden a ser imprecisas debido a su naturaleza intuitiva y su alta dependencia del contexto [1]. Esto las hace difíciles de definir y encajar en un esquema común. Que un algoritmo sea **seguro** significa que no va a alcanzar un mal estado, y decimos que una red es **consistente** cuando su comportamiento es coherente y no se salta ciertos axiomas lógicos o experiencias cotidianas asumidas. Sin embargo, ¿cómo se codifica esto? En muchas ocasiones, la respuesta es haciendo uso de las propiedades vistas anteriormente: mediante una prueba de robustez o de monotonía, por ejemplo.

3.2. Formulación del problema

Todas las propiedades de corrección estudiadas en la sección anterior tienen algo en común, y es que pueden expresarse mediante la siguiente frase:

«*Para cierta entrada x que cumple una condición P, la red produce una salida $y = f(x)$ que cumple otra condición Q.*»

Esta idea es la clave para formular un esquema general de problema de verificación, adaptable a cada caso específico mediante la selección adecuada de las condiciones necesarias. Se trata, precisamente, de la forma tradicional de especificar y verificar algoritmos que parte de la lógica de Hoare y su esquema de precondición-postcondición [6, 1]. En este marco, se establece una precondición P sobre la entrada x, y una postcondición Q que debe cumplir la salida $y = f(x)$. Suele visualizarse de la siguiente manera:

$$\begin{gathered} \{P\} \\ y = f(x) \\ \{Q\} \end{gathered} \tag{3.1}$$

Ejemplo 3.2.1. *Sea f una red neuronal de reconocimiento de imágenes. Por simplicidad, supongamos que las imágenes son en escala de grises y están representadas mediante una matriz de valores enteros entre 0 y 255. Sea $class(y)$ la función que decide la etiqueta final a la que se asigna una salida y. Se busca verificar que, cualquier imagen x resultado de alterar levemente el brillo de la imagen x_0, es clasificada con la misma etiqueta que x_0. El problema puede codificarse como:*

$$\begin{gathered} \{P \equiv |x - x_0| \leqslant \varepsilon\} \\ y_0 = f(x_0) \\ y = f(x) \\ \{Q \equiv class(x) = class(x_0)\} \end{gathered}$$

Una observación que nos será útil para desarrollar métodos algorítmicos de resolución de problemas de verificación es ver que la precondición, en realidad, define una serie de restricciones sobre el espacio de entrada, que pueden interpretarse como una región $\mathcal{X}$; y la postcondición, a su vez, define otra región sobre el espacio de salida, $\mathcal{Y}$ [2]. En este sentido, un problema de verificación quedará resuelto cuando se logre demostrar o refutar que

$$\forall x \in \mathcal{X}, y = f(x) \Rightarrow y \in \mathcal{Y} \tag{3.2}$$

Esta idea proviene directamente de la definición de robustez, solo que al formularla de este modo, se introduce cierta flexibilidad adicional: la región $\mathcal{X}$ representa el entorno de x_0, pero no se establece ningún requisito sobre la forma que ha de tener, mientras que la región $\mathcal{Y}$ puede variar desde una única etiqueta hasta todo un rango de valores aceptables para la salida.

La idea de utilizar el marco teórico de la precondición-postcondición como en la ecuación 3.1 ha sido tomada de [1], mientras que la empleada en la ecuación 3.2 es algo más común en la literatura por su enfoque práctico, y fue estudiada en [2].

3.3. Enfoques algorítmicos

Los enfoques algorítmicos son líneas generales de diseño que se utilizan para la creación de algoritmos de verificación. En la literatura existen distintos sistemas para su clasificación, algunos de ellos basados en las propiedades de los verificadores [1], y otros en el enfoque o tipo de resultado que arroja cada verificador [2]. Para esta memoria se ha optado por una mezcla, resultando en una clasificación en dos grandes bloques, respaldados por las formulaciones del problema de verificación vistas en la Sección 3.2.

Partiendo de la formulación de la ecuación 3.2, se observa que el problema de verificación equivale a comprobar que la región en la que se transforma $\mathcal{X}$ mediante f, denotada como $\mathcal{R} = f(\mathcal{X})$, está completamente contenida en la región objetivo esperada, es decir, $\mathcal{R} \subseteq \mathcal{Y}$.

Este enfoque es conocido como *método de alcanzabilidad* [2], debido a su relación con la propiedad homónima 3.1.3, y se estudiará en detalle en el Capítulo 4. A estos algoritmos, también se les conoce como *basados en dominios abstractos* o *incompletos* [1], ya veremos por qué.

Por otro lado, tomando como punto de partida la formulación del problema hecha en la ecuación 3.1, resolver el problema de verificación significa demostrar

$$P \wedge y = f(x) \Rightarrow Q \tag{3.3}$$

El segundo de los dos enfoques, que se estudiará en el Capítulo 5, trata de probar esta implicación desde el punto de vista de la lógica, traduciendo la red neuronal a fórmulas y resolviendo un problema de satisfactibilidad (SAT). Debido a las técnicas que se emplearán para su resolución, suele referirse a este enfoque como *método de optimización* [2], y sus algoritmos son también conocidos como *basados en restricciones* o *completos* [1].

Dependiendo del enfoque escogido y las decisiones de diseño que se tomen, los algoritmos devolverán resultados de un tipo determinado [2]. Por norma general, los algoritmos de

alcanzabilidad devolverán la región alcanzable, $\mathcal{R}$, o una sobreaproximación de esta. La propiedad se cumple cuando está contenida en la región objetivo, $\mathcal{R} \subseteq \mathcal{Y}$. Los algoritmos de optimización, en cambio, devuelven un contraejemplo en el caso en que la propiedad a verificar no se cumpla (UNSAT), y un modelo en el caso en que sí (SAT).

Existen otros tipos de resultados, como los *adversariales*, que muestran la tolerancia admisible alrededor de una entrada prefijada para que la región alcanzable quede dentro de la región objetivo [2, 14, 15], sin embargo, una discusión exhaustiva de estos resultados y los métodos que los arrojan quedan fuera de los objetivos de este trabajo.

3.4. Propiedades de los algoritmos

Algo que hay que tener en cuenta en el momento de diseñar un algoritmo de verificación es la corrección del propio algoritmo. No solo tenemos que ver qué propiedades de la red son deseables, sino qué propiedades de los algoritmos en sí necesitamos saber que se cumplen.

Las más importantes son la seguridad (*safety* en la literatura en inglés) y la completitud (*completeness*), aunque hay otras, como la terminación, que no son tan estudiadas [1, 2].

Definición 3.4.1. ***Propiedades de los algoritmos***

a) ***Seguridad****. Decimos que un algoritmo es* ***seguro*** *cuando, al arrojar un resultado positivo, es decir, que la propiedad a verificar se cumple, efectivamente la propiedad se cumple. Significa que el algoritmo no fallará arrojando falsos positivos.*

b) ***Completitud****. Un algoritmo es* ***completo*** *cuando es seguro y, además, al devolver un resultado negativo (la propiedad no se cumple), la propiedad verdaderamente no se cumple. Es decir, el algoritmo no devolverá falsos negativos.*

Todos los algoritmos que se diseñen deben ser seguros, pues, de lo contrario, no podríamos fiarnos de ellos. Sin embargo, no todos tienen por qué ser completos.

Un algoritmo que es completo garantiza que el resultado que devuelve es cierto, tanto si dice que la propiedad se verifica como si no. Sin embargo, un algoritmo que no sea completo puede decir que la propiedad no se cumple, y que en realidad sí se cumpla. De manera intuitiva, podemos pensar que el algoritmo «*no quiere pillarse los dedos*» asegurando que la propiedad se cumple, y por eso, ante la duda, dice que no. Pero no hay problema, porque si es seguro, cuando devuelva que sí, sabremos a ciencia cierta que es verdad.

Idealmente, querríamos que todos los algoritmos fuesen completos y, sobre el papel, actualmente existen algoritmos completos y seguros para todos los enfoques. Sin embargo, en ocasiones estos no resultan eficientes. Por cuestiones de capacidad de cómputo, complejidad algorítmica o escalabilidad, muchas veces resulta beneficioso *sacrificar* la completitud para obtener un algoritmo de verificación que sea capaz de resolver los problemas en costes asumibles.

Discutiremos todos estos problemas en los próximos capítulos, conforme se desarrollan cada uno de los enfoques.

Capítulo 4

Algoritmos de alcanzabilidad

En este capítulo se estudiarán en detalle los algoritmos de alcanzabilidad, partiendo del más básico e introduciendo diferentes técnicas para mejorar su precisión y eficiencia. Se comenzará con la propagación de hiperrectángulos (4.2), un método sencillo pero eficiente que posteriormente se refinará mediante el uso de politopos (4.3). Ambos algoritmos, que operan neurona a neurona, han sido extraídos de [1]. Luego, se explorará un enfoque capa a capa con los algoritmos ExactReach [12] (4.4), que garantiza la completitud de la verificación, y Ai2 [13], que mejora su eficiencia, haciendo uso de la técnica (4.5) para obtener una versión más rápida aunque incompleta. La explicación de los dos últimos ha sido tomada de [2].

4.1. Introducción

Como se ha visto en la Sección 3.3, podemos pensar en la precondición y la postcondición como restricciones que definen regiones en los espacios de entrada y salida de la red.

Sea $f : \mathbb{R}^{k_0} \longrightarrow \mathbb{R}^{k_n}$ la función que representa la transformación realizada por una red neuronal de n capas, y $\mathcal{X} \subseteq \mathbb{R}^{k_0}$ e $\mathcal{Y} \subseteq \mathbb{R}^{k_n}$ las regiones en los espacios de entrada y salida, respectivamente, definidas por las restricción dadas por la precondición y postcondición. El problema de verificación consiste en probar la ecuación 3.2. Para ello, se codifica $\mathcal{X}$ como *dominio abstracto*, y se propaga hacia adelante por la red hasta obtener $f(\mathcal{X}) = \mathcal{R} \subseteq \mathbb{R}^{k_n}$, la región alcanzable. La propiedad quedará verificada si $\mathcal{R} \subseteq \mathcal{Y}$. Puede verse una representación gráfica en la Figura 4.1.

Definición 4.1.1. [1] *Un **dominio abstracto** es una representación matemática de un conjunto de valores dentro de un espacio determinado, para el que se definen operaciones como la suma, producto por escalares, unión o intersección. Los hiperrectángulos, zonotopos o politopos son ejemplos de dominios abstractos.*

En la práctica, existen algunas dificultades para llevar esto a cabo. La primera de ellas es la codificación de las regiones: dependiendo del dominio abstracto que se utilice para representarlas, se obtendrá un modelado más o menos preciso y expresivo. La segunda es

[1] Elaboración propia basada en [1].

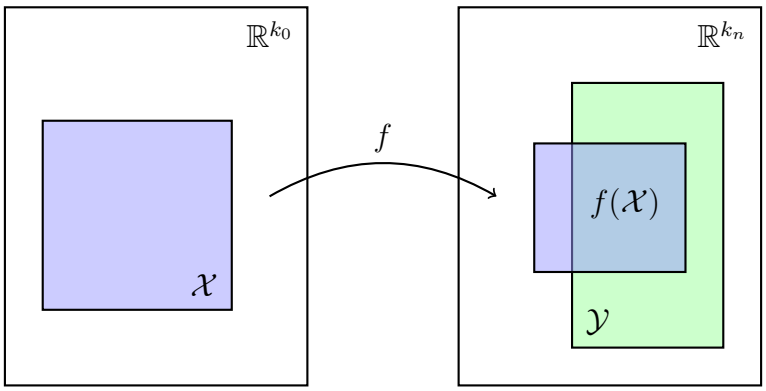


Figura 4.1: Ilustración del objetivo del enfoque de alcanzabilidad. En este ejemplo, la propiedad no se cumple, pues $\mathcal{R}$ no está contenido en $\mathcal{Y}$. Elaboración propia.

la forma de tratar la parte no lineal de las transformaciones de las neuronas: el método escogido determinará si se comete o no un error. Por último, encontramos las limitaciones en la capacidad de cómputo y los tiempos intratables resultado de una complejidad algorítmica elevada.

Por todo ello, salvo excepciones, los métodos de alcanzabilidad lo que buscan en realidad (o lo mejor que pueden conseguir), es obtener una región $\widetilde{\mathcal{R}} \supseteq \mathcal{R}$ que sobreaproxima la región alcanzable real. Esta sobreaproximación será la que provoque la pérdida de la completitud de los algoritmos (véase la Figura 4.2), motivo por el cual esta familia suele denominarse en la literatura como *algoritmos de verificación incompletos*, aunque algunos como ExactReach (Sección 4.4) sí consiguen la completitud.

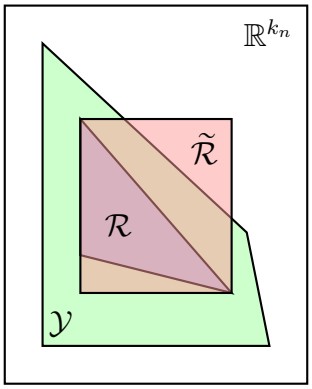


Figura 4.2: La región $\widetilde{\mathcal{R}}$ sobreaproxima $\mathcal{R}$. Aunque la propiedad se cumple en este ejemplo ($\mathcal{R} \subseteq \mathcal{Y}$), el algoritmo devolverá que no se verifica, pues la región calculada $\widetilde{\mathcal{R}} \not\subseteq \mathcal{Y}$. El algoritmo no es completo. Elaboración propia.

Observación 4.1.2. *Al diseñar un algoritmo que aproxime $\mathcal{R}$, dicha aproximación debe ser siempre por encima, es decir, ha de cumplir $\mathcal{R} \subseteq \widetilde{\mathcal{R}}$. De lo contrario, se perdería la propiedad de seguridad.*

4.2. Hiperrectángulos

La forma más sencilla en la que podemos codificar una región es mediante intervalos. Para cada neurona en la capa de entrada, se establece el límite inferior l y superior u de los

valores que puede admitir. Esto define un *hiperrectángulo* en $\mathbb{R}^{k_0}$ que modeliza la región $\mathcal{X}$.

Definición 4.2.1. *Llamamos* ***hiperrectángulo*** *en $\mathbb{R}^n$ a la región n-dimensional $\{x \in \mathbb{R}^n \mid l_i \leqslant x_i \leqslant u_i\}$ con límites dados por $([l_1, u_1], \ldots, [l_n, u_n])$.*

Definición 4.2.2. *Sea $g : \mathbb{R}^n \to \mathbb{R}$ una función que actúa sobre los elementos de un conjunto $X \subset \mathbb{R}^n$. Un* ***transformador abstracto*** *[1] para g será una función $g^a : D \to \mathbb{R}$ que cumpla $g(X) = \{g(x) \mid x \in X\} \subseteq g^a(X)$, y que actúa sobre un objeto D, representación de un dominio abstracto determinado.*

Ejemplo 4.2.3. *Sea $g(x) = x + 5$, $X = ([0, 1])$ y D el hiperrectángulo de una dimensión (intervalo). El transformador abstracto para g es la función $g^a([l, u]) = ([l + 5, u + 5])$. En este caso, para X, $g^a(X) = g^a([0, 1]) = ([5, 6])$.*

Los transformadores abstractos permiten definir funciones sobre dominios abstractos. De esta manera, es posible propagar la región $\mathcal{X}$ por las neuronas y capas de la red para obtener la región alcanzable. Ya vimos que cada neurona realiza una transformación lineal (afín) y otra no lineal (activación). Veamos cómo se definen transformadores para cada una de ellas.

Proposición 4.2.4. *[1] El transformador abstracto para la función afín*

$$g(x_1, \ldots, x_n) = c_0 + \sum_{i=1}^{n} c_i x_i \tag{4.1}$$

con $c_i \in \mathbb{R}$, empleando como dominio el hiperrectángulo, se define como:

$$g^a([l_1, u_1], \ldots, [l_n, u_n]) = \left[c_0 + \sum_{i=1}^{n} \alpha_i,\ c_0 + \sum_{i=1}^{n} \beta_i \right] \tag{4.2}$$

siendo $\alpha_i = \text{mín}(c_i l_i, c_i u_i)$ y $\beta_i = \text{máx}(c_i l_i, c_i u_i)$.

Proposición 4.2.5. *[1] El transformador abstracto para cualquier función g monótona creciente usando el dominio de intervalos es $g^a([l, u]) = [g(l), g(u)]$. En particular, las funciones de activación estudiadas en 2.3 como ReLU o sigmoide, y cualquier función PWL, lo son. Por tanto*

$$\text{relu}([l, u]) = [\text{relu(l)},\ \text{relu}(u)] \tag{4.3}$$

Con esto tenemos todo lo necesario para construir el primer algoritmo de verificación. Veremos su funcionamiento mediante el siguiente ejemplo, que compararemos posteriormente con los demás algoritmos.

Ejemplo 4.2.6. *Sea $f : \mathbb{R}^2 \to \mathbb{R}^2$ la red neuronal con una capa de entrada y otra de salida, ambas con dos neuronas, representada en el diagrama de la Figura 4.3, cuya matriz de pesos W y vector de sesgos b es*

$$W = \begin{pmatrix} 2 & -1 \\ 1 & 3 \end{pmatrix} \qquad b = \begin{pmatrix} 0 \\ -2 \end{pmatrix}$$

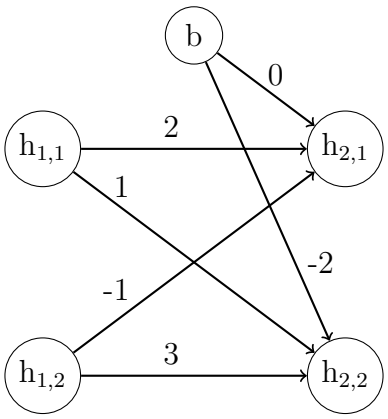


Figura 4.3: Grafo de la red neuronal f del ejemplo transversal a todo el capítulo. Elaboración propia.

Sea $\mathcal{X} = ([0,1],[0,1]) \subseteq \mathbb{R}^2$ *la región de entrada dada por la precondición, modelizada mediante h-rectángulos. Propaguémosla utilizando los transformadores abstractos definidos.*

La transformación lineal que realiza la neurona $h_{2,1}$ *viene dada por la función* $g_1(x,y) = 2x - y$*. Por tanto,* $g_1([0,1],[0,1]) = [2 \cdot 0 - 1 \cdot 1, 2 \cdot 1 - 1 \cdot 0] = [-1,2]$*. De igual modo, para la neurona* $h_{2,2}$ *con* $g_2(x,y) = x + 3y - 2$*, se tiene que* $g_2([0,1],[0,1]) = [1 \cdot 0 + 3 \cdot 0 - 2, 1 \cdot 1 + 3 \cdot 1 - 2] = [-2,2]$.

Por último, aplicamos la activación: $relu([-1,2]) = [0,2]$ *y* $relu([-2,2]) = [0,2]$*. De este modo, la región alcanzable calculada,* $\widetilde{\mathcal{R}}$*, es el rectángulo* $([0,2],[0,2]) \subseteq \mathbb{R}^2$*. Puede verse un diagrama de su representación en la Figura 4.4.*

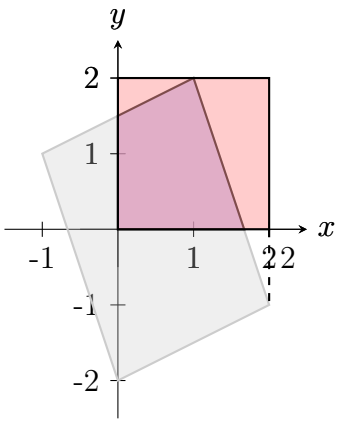


Figura 4.4: Resultado de la propagación de $([0,1],[0,1])$ por hiperrectángulos. En rojo, la región $\widetilde{\mathcal{R}}$ y, por debajo, en azul, la región alcanzable real $\mathcal{R}$. En gris, la transformación afín de $([0,1],[0,1])$, previa a la activación. Elaboración propia.

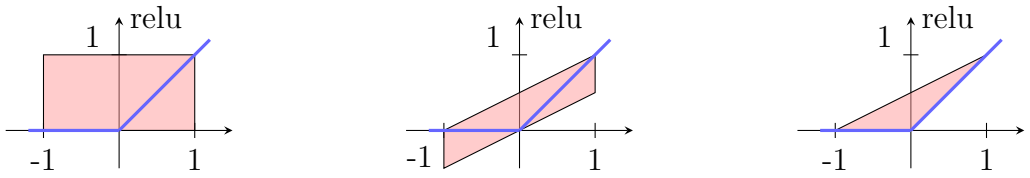
Como puede observarse a través del diagrama 4.4, este método de verificación, al que llamaremos *propagación de hiperrectángulos*, no es completo [1], pues hay sobreaproximación. En general, los algoritmos que propagan regiones neurona a neurona como este, perderán la completitud por dos motivos.

El primero de ellos es la **expresividad**: en el caso de los h-rectángulos, la capacidad que

tienen de modelizar regiones es bastante limitada, en el sentido de que es imposible ligar las variables de una dimensión con las de otra (no se puede modelar la recta $y = x$, por ejemplo), y tampoco se pueden codificar regiones que no tengan sus lados paralelos a los ejes. Esto provoca que, después de la transformación afín de cada capa, los algoritmos tengan que *encajar* la región real resultante en una que sea del tipo de dominio abstracto que trabajan.

En la Figura 4.4, puede verse que el límite superior de $\widetilde{\mathcal{R}}$ para la coordenada x es 2 y, sin embargo, no parece que la región real llegue a alcanzarlo. Puede verse una explicación visual en la misma figura: el rectángulo gris es el resultado directo de aplicar a $\mathcal{X}$ la transformación afín dada por los pesos y el sesgo. Al realizar la activación ReLU, los puntos que se encuentran en el tercer cuadrante son proyectados en el eje x, pues la coordenada y es negativa y se transforma en 0, dando lugar así al intervalo $[0, 2]$, que, efectivamente, forma parte de $\mathcal{R}$.

El otro motivo de pérdida de completitud es la forma en que se trata la transformación no lineal que introduce la **activación**. Para encajar la función ReLU (y cualquier función de activación) en un dominio abstracto, es inevitable cometer un error de sobreaproximación, véase la Figura 4.5. Cualquier algoritmo de verificación que modelice funciones de activación por dominios abstractos será incompleto [1].



(a) Hiperrectángulos **(b)** Zonotopos **(c)** Politopos

Figura 4.5: Aproximación de la función ReLU por distintos dominios abstractos. Elaboración propia.

4.3. Politopos

Para tratar de solucionar el problema de la expresividad, existen dominios abstractos que permiten mayor libertad que el h-rectángulo, como los *zonotopos* y los *politopos*, y que además, por ser más precisos, mejoran notablemente las aproximaciones que se realizan. A cambio, requieren una codificación más compleja y mayor número de operaciones, pudiendo dar lugar a algoritmos demasiado costosos.

Definición 4.3.1. *Se conoce como* ***politopo*** *a la generalización de los polígonos y poliedros a cualquier número de dimensiones.*

Definición 4.3.2. *Un* ***politopo convexo*** P *de n dimensiones se define como la envoltura convexa de un conjunto finito de puntos en $\mathbb{R}^n$. Existen distintas definiciones alternativas equivalentes:*

a) ***V-Politopo*** *[2]: Sea $v_1, v_2, \ldots, v_k \in \mathbb{R}$ un conjunto finito de puntos llamados vérti-*

ces. El politopo convexo generado por ellos es:

$$P = \text{conv}(\{v_1, v_2, \ldots, v_k\}) = \left\{ \sum_{i=1}^{k} \lambda_i v_i \;\middle|\; \lambda_i \geqslant 0, \sum_{i=1}^{k} \lambda_i = 1 \right\} \tag{4.4}$$

donde conv *es la envoltura convexa.*

b) ***H-Politopo*** *[2]: Un politopo convexo es la intersección de un número finito de semiespacios (cerrados) en* $\mathbb{R}^n$*:*

$$P = \{x \in \mathbb{R}^n \mid Ax \leqslant c\} \tag{4.5}$$

donde A *es una matriz* $m \times n$*,* y c *es un vector en* $\mathbb{R}^m$ *que representan las* m *inecuaciones que delimitan la región.*

c) *Mediante* ***generadores*** *[1]: Dado un conjunto* F *de inecuaciones lineales sobre* m *generadores* $\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_m$*, definimos un politopo como el conjunto*

$$P = \left\{ \left(c_{10} + \sum_{i=1}^{m} c_{1i} \cdot \varepsilon_i, \ldots, c_{n0} + \sum_{i=1}^{m} c_{ni} \cdot \varepsilon_i) \right) \;\middle|\; F(\varepsilon_1, \varepsilon_2, \ldots, \varepsilon_m) \right\}, \tag{4.6}$$

que abreviaremos con la notación $(\langle c_{1i} \rangle_i, \ldots, \langle c_{ni} \rangle_i, F)$*.*

En esta sección se trabajará con politopos convexos definidos por generadores (ecuación 4.6) y su notación abreviada. Se seguirá el mismo método que con los h-rectángulos: propagar la región codificada con politopos neurona a neurona a través de transformadores abstractos hasta la capa de salida.

Proposición 4.3.3. *[1] El transformador abstracto para la función afín*

$$g(x_1, \ldots, x_n) = a_0 + \sum_{i=1}^{n} a_i x_i \tag{4.7}$$

con $c_i \in \mathbb{R}$*, empleando como dominio el politopo convexo, se define como:*

$$g^a(\langle c_{1i} \rangle, \ldots, \langle c_{ni} \rangle, F) = \left(\left\langle a_0 + \sum_{j=1}^{n} a_j c_{j0}, \sum_{j=1}^{n} a_j c_{j1}, \ldots, \sum_{j=1}^{n} a_j c_{jm} \right\rangle, F \right) \tag{4.8}$$

donde n *es el número de dimensiones y* m *el número de generadores. Observación: las variables auxiliares* i *y* j *cuentan generadores y dimensiones, respectivamente. El conjunto de restricciones sobre los generadores,* F*, se mantiene inalterado.*

Proposición 4.3.4. *[1] El transformador abstracto para la función ReLU empleando como dominio el politopo convexo se define como sigue:*

$$relu^a(\langle c_i \rangle_i, F) = (\langle 0, \underbrace{0, \ldots, 0}_{m}, 1 \rangle, F') \tag{4.9}$$

$$F' \equiv F \wedge \text{máx}(0, \langle c_i \rangle) \leqslant \varepsilon_{m+1} \leqslant \frac{u(\langle c_i \rangle - l)}{u - l} \tag{4.10}$$

donde l *y* u *son los límites superiores e inferiores, y se calculan mediante optimización lineal.*

Observación 4.3.5. *Este transformador añade un generador adicional, ε_{m+1}, sobre el que se definen las restricciones necesarias para obtener el nuevo politopo. Los coeficientes de los demás generadores y el término independiente se* resetean*, poniendo su valor a cero.*

Ejemplo 4.3.6. *Utilicemos estos dos transformadores para calcular la región de alcanzabilidad de la red definida en 4.3, partiendo igualmente de $\mathcal{X} = [0,1] \times [0,1]$, que podemos codificar como politopo de dos dimensiones con dos generadores de la siguiente manera:*

$$\mathcal{X} = \{(\varepsilon_1, \varepsilon_2) \mid \varepsilon_1, \varepsilon_2 \in [0,1]\} = (\langle 0,1,0\rangle, \langle 0,0,1\rangle, F)$$

siendo $F \equiv 0 \leqslant \varepsilon_1 \leqslant 1 \wedge 0 \leqslant \varepsilon_2 \leqslant 1$.

Empecemos por las transformaciones afines:

- *Para la neurona $h_{2,1}$, $g_1(x,y) = 2x - y$:*

$$\begin{aligned} g_1^a(\langle 0,1,0\rangle, \langle 0,0,1\rangle, F) &= (\langle 0 + 2\cdot 0 - 1\cdot 0, 2\cdot 1 - 1\cdot 0, 2\cdot 0 - 1\cdot 1\rangle, F) \\ &= (\langle 0, 2, -1\rangle, F) \end{aligned}$$

- *Para $h_{2,2}$, $g_2(x,y) = x + 3y - 2$:*

$$\begin{aligned} g_2^a(\langle 0,1,0\rangle, \langle 0,0,1\rangle, F) &= (\langle -2 + 1\cdot 0 + 3\cdot 0, 1\cdot 1 + 3\cdot 0, 1\cdot 0 + 3\cdot 1\rangle, F) \\ &= (\langle -2, 1, 3\rangle, F) \end{aligned}$$

Puede verse que el politopo $(\langle 0,2,-1\rangle, \langle -2,1,3\rangle, F)$ se corresponde con el paralelogramo gris de la Figura 4.4. Falta realizar la activación ReLU:

- *Para $h_{2,1}$:*

$$\begin{aligned} \text{relu}(\langle 0,2,-1\rangle, F) &= (\langle 0,0,0,1,0\rangle, F') \\ F' \equiv F \wedge \text{máx}(0, 2\varepsilon_1 - \varepsilon_2) \leqslant \varepsilon_3 &\leqslant \frac{4\varepsilon_1 - 2\varepsilon_2 + 2}{3} \end{aligned}$$

- *Para $h_{2,2}$:*

$$\begin{aligned} \text{relu}(\langle -2,1,3\rangle, F) &= (\langle 0,0,0,0,1\rangle, F'') \\ F'' \equiv F \wedge \text{máx}(0, -2 + \varepsilon_1 + 3\varepsilon_2) \leqslant \varepsilon_4 &\leqslant \frac{\varepsilon_1 + 3\varepsilon_2}{2} \end{aligned}$$

Observamos que, al hacer la función ReLU en la primera neurona se ha añadido el generador ε_3, y en la segunda, el ε_4. El resultado final es

$$\widetilde{\mathcal{R}} = (\langle 0,0,0,1,0\rangle, \langle 0,0,0,0,1\rangle, F' \wedge F'')$$

Puede verse una representación gráfica en la Figura 4.6.

Esta vez, la aproximación ha sido mucho más ajustada. Aún así, sigue habiendo un error considerable. En parte, la culpa es de los dominios abstractos. La región real es *imposible* de modelizar mediante un politopo convexo (pues el camino entre los puntos $(1,2)$ y $(2,0)$ no está contenido en su interior). Además, como ya vimos, la sobreaproximación de la ReLU hace que sea imposible lograr la completitud.

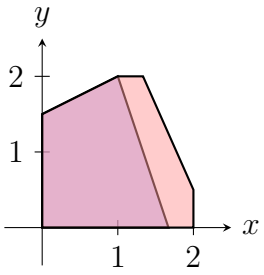


Figura 4.6: Resultado de la propagación de $[0,1]\times[0,1]$ por politopos. Elaboración propia.

4.4. ExactReach

Para conseguir la completitud será necesario un cambio de enfoque. Hasta ahora, se ha trabajado aplicando los transformadores abstractos neurona a neurona. En esta sección se presenta el algoritmo ExactReach [12, 2], que utiliza politopos convexos, pero los propaga capa a capa. Utilizaremos las representaciones como V-politopos (ecuación 4.4) y H-politopos (ecuación 4.5).

Veamos cómo se propaga cada capa i. Lo primero que hay que realizar es la transformación lineal. Para ello, el algoritmo toma un V-politopo y transforma cada vértice con los pesos W_i y sesgos b_i. Los siguientes resultados aseguran que el V-politopo resultante es el correcto, y que no se realiza ninguna sobreaproximación.

Proposición 4.4.1. *Sea $f : \mathbb{R}^n \longrightarrow \mathbb{R}^m$ una aplicación afín. Entonces, para todo conjunto de puntos $X \subset \mathbb{R}^n$, se tiene que*

$$f(\text{conv}(X)) = \text{conv}(f(X)) \tag{4.11}$$

Demostración. Debe probarse la doble inclusión.

Empecemos por $f(\text{conv}(X)) \subset \text{conv}(f(X))$. Sea un punto arbitrario $y \in f(\text{conv}(X))$, veamos que $y \in \text{conv}(f(X))$. Sabemos que $y = f(x)$ para algún $x \in \text{conv}(X)$, y que, por definición de combinación convexa, $x = \sum_k \lambda_i x_i$, con $x_i \in X$, $\lambda_i \geqslant 0$ y $\sum_k \lambda_i = 1$. Por ser f afín, preserva combinaciones convexas, por lo que

$$y = f(x) = f\left(\sum_{i=1}^{k} \lambda_i x_i\right) = \sum_{i=1}^{k} \lambda_i f(x_i)$$

con $f(x_i) \in f(X)$. Ya que y es una combinación convexa en $f(X)$, $y \in \text{conv}(f(X))$. El sentido contrario es análogo, ya que la preimagen por una transformación afín de una combinación convexa también es convexa[2]. □

Corolario 4.4.2. *Sea $P \subset \mathbb{R}^n$ un politopo convexo. Entonces, $f(P) \subset \mathbb{R}^m$ es también un politopo convexo.*

Demostración. Por definición, $P = \text{conv}(\{v_1, v_2, \ldots, v_k\})$. Entonces, por la Proposición 4.4.1, $f(P) = \text{conv}(\{f(v_1), f(v_2), \ldots, f(v_k)\})$, que es otro politopo convexo[2]. □

[2]Elaboración propia.

Ejemplo 4.4.3. *La región* $[0,1] \times [0,1]$ *puede codificarse como H-politopo como*

$$\mathcal{X} = \{(x, y) \in \mathbb{R}^2 \mid 0 \leqslant x \leqslant 1;\ 0 \leqslant y \leqslant 1\}$$

Y como V-politopo mediante sus vértices:

$$\mathcal{X} = \left\{ \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \begin{pmatrix} 1 \\ 1 \end{pmatrix} \right\}$$

Aplicando a cada vértice la operación $\tilde{x}_i = Wx_i + b$*, siendo* W *y* b *las del Ejemplo 4.3, obtenemos*

$$\hat{\mathcal{Z}} = \left\{ \begin{pmatrix} 0 \\ -2 \end{pmatrix}, \begin{pmatrix} -1 \\ 1 \end{pmatrix}, \begin{pmatrix} 2 \\ -1 \end{pmatrix}, \begin{pmatrix} 1 \\ 2 \end{pmatrix} \right\}$$

Puede verse que estos son los vértices del paralelogramo sombreado en gris de la Figura 4.4, que se corresponde con la transformación antes de la activación. Su representación como H-politopo es

$$\hat{\mathcal{Z}} = \left\{ (\hat{z}_1, \hat{z}_2) \in \mathbb{R}^2 \;\middle|\; \begin{pmatrix} 1 & -2 \\ 3 & 1 \\ -1 & 2 \\ -3 & -1 \end{pmatrix} \begin{pmatrix} \hat{z}_1 \\ \hat{z}_2 \end{pmatrix} \leqslant \begin{pmatrix} 4 \\ 5 \\ 3 \\ 2 \end{pmatrix} \right\}$$

El siguiente paso es realizar la activación, solo que ahora no es posible hacerlo coordenada a coordenada. La idea clave es ver que la función ReLU, por definición, divide el espacio en dos regiones disjuntas con distinto patrón de activación. Decimos que una entrada está *activada* si es mayor o igual a cero, y que está *inactiva* si no lo es.

Generalizando esta idea para k dimensiones, vemos que la ReLU divide el espacio $\mathbb{R}^k$ en 2^k regiones disjuntas, pues cada dimensión puede ser marcada con un 0 o un 1, dependiendo de si ha sido activada o no.

Cada patrón de activación $h \in \{1, 2, \ldots, 2^k\}$ puede expresarse de forma inequívoca como un vector binario $\{0,1\}^k$, que será la diagonal de la matriz de activación P_h. Añadiendo el conjunto de restricciones $(I - 2P_h)x \leqslant 0$ al politopo resultado de la transformación afín (ecuación 4.12), $\hat{\mathcal{Z}}$, se consigue «*recortarlo*» restringiéndolo a la subregión sobre la que actúa el patrón de activación h [2].

$$\hat{\mathcal{Z}}_h = \left\{ \hat{z} \in \mathbb{R}^k \;\middle|\; \begin{pmatrix} \hat{A} \\ I - 2P_h \end{pmatrix} \hat{z} \leqslant \begin{pmatrix} c \\ 0 \end{pmatrix} \right\} \tag{4.12}$$

Una vez hecho el recorte, se puede aplicar a cada una de las h subregiones la activación correspondiente:

$$\mathcal{Z}_h = \{z = P_h \hat{z} \mid \hat{z} \in \hat{\mathcal{Z}}_h\} \tag{4.13}$$

La salida de la capa será la unión de todos estos conjuntos activados:

$$\mathcal{Z} = \bigcup_{h=1}^{2^k} \mathcal{Z}_h \tag{4.14}$$

El conjunto de salida $\mathcal{Z}$ es ajustado [2], en el sentido de que no comete ningún error de sobreaproximación. Gracias a esto, el algoritmo ExactReach es seguro y completo: la

codificación por politopos es expresiva, la transformación afín se realiza de forma precisa en todos los casos, y la activación consigue capturar el comportamiento de la función ReLU sin sobreaproximar.

Ejemplo 4.4.4. *Continuando con el ejemplo anterior, al tener dos neuronas y ser, por tanto, el espacio* $\mathbb{R}^2$*, hay* $2^2 = 4$ *patrones de activación diferentes, que se corresponden con los cuatro cuadrantes coordenados. En el primer cuadrante,* $P_1 = \left(\begin{smallmatrix} 1 & 0 \\ 0 & 1 \end{smallmatrix}\right)$*, por lo que deben añadirse a* $\hat{\mathcal{Z}}$ *las restricciones*

$$\begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} \hat{z}_1 \\ \hat{z}_2 \end{pmatrix} \leqslant \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

Realmente, lo único que se ha hecho ha sido añadir las inecuaciones $\hat{z}_1 \geqslant 0$ *y* $\hat{z}_2 \geqslant 0$*, que son las que caracterizan el primer cuadrante. El resto de cuadrantes son análogos, con sus correspondientes matrices,* $P_2 = \left(\begin{smallmatrix} 0 & 0 \\ 0 & 1 \end{smallmatrix}\right)$*,* $P_3 = \left(\begin{smallmatrix} 0 & 0 \\ 0 & 0 \end{smallmatrix}\right)$ *y* $P_4 = \left(\begin{smallmatrix} 1 & 0 \\ 0 & 0 \end{smallmatrix}\right)$*.*

Por último, se aplica la activación correspondiente a cada subregión $\hat{\mathcal{Z}}_i$*. En el caso del primer cuadrante, por ser* $P_1 = I$*, no hay ningún cambio. En el tercer cuadrante,* $P_3 = O$*, por lo que el único punto que queda es el origen:* $\hat{\mathcal{Z}}_3 = \{(0, 0)\}$*. Para el segundo y cuarto cuadrantes, la ReLU resulta en una proyección del politopo recortado en el eje* OX *y* OY*, respectivamente, dando lugar a:*

$$\hat{\mathcal{Z}}_2 = \left\{(0, \hat{z}_2) \in \mathbb{R}^2 \;\middle|\; \hat{z}_2 \in [0, 3/2]\right\}$$
$$\hat{\mathcal{Z}}_4 = \left\{(\hat{z}_1, 0) \in \mathbb{R}^2 \;\middle|\; \hat{z}_1 \in [0, 2]\right\}$$

La región alcanzable final será, por tanto,

$$\mathcal{R} = \hat{\mathcal{Z}}_1 \cup \hat{\mathcal{Z}}_2 \cup \hat{\mathcal{Z}}_3 \cup \hat{\mathcal{Z}}_4 = \hat{\mathcal{Z}}_1 \cup \hat{\mathcal{Z}}_4$$

puesto que los cuadrantes 2 *y* 3 *están contenidos en el primero. Véase que no se ha escrito* $\widetilde{\mathcal{R}}$*, sino directamente* $\mathcal{R}$*, pues el método es completo y la región alcanzable calculada es la real. Una representación gráfica puede verse en la Figura 4.7(a).*

(a) ExactReach **(b)** Ai2 con politopos

Figura 4.7: Región alcanzable resultado de la propagación de $[0, 1] \times [0, 1]$. En (a) aparece la región alcanzable real. Elaboración propia.

Este algoritmo es completo, pero, a cambio, tiene un alto coste computacional. Una capa de k neuronas divide el espacio en 2^k subespacios que deben ser procesados, y habrán de propagarse en la siguiente capa, dando lugar a un coste exponencial en el número n de neuronas, $\mathcal{O}(2^n)$ [2]. Esto hace que el algoritmo sea muy poco escalable, e intratable para redes neuronales complejas que tengan un alto número de capas y neuronas.

Es cierto que existen algunas mejoras que pueden realizarse, como optimizar el número de desigualdades en los politopos tras la transformación lineal y el recorte, o ignorar los recortes que estén contenidos en otros, como se ha tenido en cuenta en el ejemplo. Aún así, en el peor de los casos, el coste de la completitud sigue siendo demasiado elevado [2, 12].

4.5. *Split and Join*

Hasta ahora se han propagando hiperrectángulos neurona a neurona, dando lugar a grandes errores de sobreaproximación, que se han reducido gracias a la expresividad de los politopos. Por último, se ha refinado la representación algorítmica de la ReLU utilizando ExactReach y la técnica de la propagación capa a capa. Todo esto con el objetivo de desarrollar un algoritmo más preciso hasta alcanzar la completitud. Sin embargo, el resultado ha sido un método computacionalmente muy costoso, y lo que se busca ahora es dar un paso atrás, sacrificando la completitud en favor de un método que logre un equilibrio entre fiabilidad y eficiencia.

El algoritmo Ai2 [13, 2], implementa sobre las ideas de ExactReach la técnica *Split and Join* (separar y juntar), para evitar la ramificación exponencial en las subregiones producidas al aplicar la ReLU. Los pasos son iguales que ExactReach: se realiza la transformación afín, la separación (*split*) por patrones de activación, y la activación de cada una de las subregiones. Sin embargo, en lugar de alimentar la siguiente capa con una lista con los politopos resultantes de cada subregión, unifica (*join*) todas ellas en un único politopo, utilizando algún algoritmo de *Convex Hull* (como el implementado en `SciPy.spatial` [16]) sobre los vértices de cada una.

Al hacer la envoltura convexa se comete un error de sobreaproximación, pero es el mínimo que se puede cometer para conseguir un único politopo después de las transformaciones de cada capa [2].

Ejemplo 4.5.1. *Continuando el ejemplo anterior donde lo dejamos, los vértices de cada región son:* $V(\hat{\mathcal{Z}}_1) = \left\{\binom{0}{0}, \binom{5/3}{0}, \binom{1}{2}, \binom{0}{3/2}\right\}$ *y* $V(\hat{\mathcal{Z}}_4) = \left\{\binom{0}{0}, \binom{2}{0}\right\}$*, y su envoltura convexa es* $\widetilde{\mathcal{R}} = \text{conv}(V(\hat{\mathcal{Z}}_1), V(\hat{\mathcal{Z}}_4))$*, cuyos vértices son*

$$V(\widetilde{\mathcal{R}}) = \left\{\begin{pmatrix}0\\0\end{pmatrix}, \begin{pmatrix}2\\0\end{pmatrix}, \begin{pmatrix}1\\2\end{pmatrix}, \begin{pmatrix}0\\3/2\end{pmatrix}\right\}$$

Una representación visual puede verse en la Figura 4.7(b).

Observación 4.5.2. *Originalmente, Ai2 fue implementado utilizando zonotopos en lugar de politopos convexos [13]. Se trata de un dominio abstracto con expresividad intermedia entre el h-rectángulo y el politopo.*

Capítulo 5

Algoritmos de optimización

En este capítulo se examinarán en detalle los algoritmos de optimización, que buscan decidir la satisfactibilidad del problema de verificación haciendo uso de técnicas de programación lineal. Se comenzará explicando cómo se pueden codificar las redes neuronales (5.2) y qué desafíos surgen durante el proceso. A continuación, se explorarán tres métodos clave para resolverlos: el primero de ellos es el algoritmo Reluplex (5.3), que combina ideas de lógica con búsqueda binaria en un enfoque híbrido; el segundo es NSVerify (5.4), una implementación del problema de optimización directa pero ingenua; y el último, MIPVerify (5.4), que refina las ideas del anterior para lograr una mayor eficacia.

Los tres algoritmos se describen en [2], destacando especialmente su versión conjuntista de Reluplex [4], que será la que se explique en este capítulo. Las ideas sobre la codificación y los retos a resolver que guían el desarrollo de este capítulo provienen, una vez más, de la combinación de [1] y [2].

5.1. Introducción

El segundo gran enfoque de verificación es el de optimización. Partiendo esta vez de la ecuación 3.1, el problema de verificación consiste en demostrar

$$P \wedge y = f(x) \Longrightarrow Q \tag{5.1}$$

o, lo que es lo mismo, probar que no puede cumplirse

$$P \wedge y = f(x) \wedge \neg Q \tag{5.2}$$

Si existiese una asignación de las variables x e y para la fórmula 5.2 que la hiciese cierta, la propiedad no se cumpliría. El siguiente paso es, por tanto, codificar la precondición P, la postcondición Q, y la red $y = f(x)$ para comprobar el estado lógico de dicha fórmula.

Para realizar esta codificación se utiliza Lógica de Primer Orden (en inglés, *FOL*, First Order Logic) bajo la teoría del álgebra lineal real (en inglés, *LRA*, Linear Real Algebra) [1]. Supongamos que P y Q definen sendos conjuntos de restricciones $\mathcal{X}$ e $\mathcal{Y}$ sobre los espacios de entrada y salida, respectivamente. Probar si la expresión 5.2 es verdadera o falsa puede

formularse, entonces, como la resolución del siguiente **problema de optimización** (de ahí el nombre del enfoque) [2]:

$$\begin{aligned} \min_{x,y} \quad & \text{obj}(x, y, \mathcal{X}, \mathcal{Y}) \\ \text{t.q.} \quad & x \in \mathcal{X} \\ & y = f(x) \\ & y \notin \mathcal{Y} \end{aligned} \tag{5.3}$$

Si el problema 5.3 tiene solución factible significa que la propiedad no se cumple, pues la asignación de las variables x e y encontradas forman un *contraejemplo* de dicha propiedad, es decir, un caso en el que se cumple la precondición pero no la postcondición. Si el problema de optimización no tiene solución, entonces la propiedad se cumple.

La función objetivo, $\text{obj}(x, y, \mathcal{X}, \mathcal{Y})$, juega un papel secundario en el análisis, pues no es necesaria para la verificación: solo necesitamos saber si existen soluciones factibles. Los algoritmos Reluplex y NSVerify, de hecho, no establecen ninguna función objetivo y son completos [2]. En cambio, MIPVerify sí que lo hace, obteniendo resultados de tipo *adversarial*.

5.2. Codificación de la red neuronal

La precondición y postcondición vendrán dadas como un conjunto de restricciones sobre las entradas $x \in \mathbb{R}^{k_0}$ y las salidas $y \in \mathbb{R}^{k_n}$. De igual manera que en el enfoque de alcanzabilidad, estas pueden interpretarse como regiones de k_i dimensiones, que podrán ser hiperrectángulos, politopos, o incluso subconjuntos no acotados.

Por lo general, por comodidad en el diseño del algoritmo, se prefiere que $\mathcal{X}$ sea un H-politopo, pues su descripción forma las inecuaciones que codifican la condición $x \in \mathcal{X}$ directamente en el lenguaje LRA [2]. Por otro lado, es común que $\mathcal{Y}$ sea dado por un *complemento de politopo* CP (si P es un politopo en el espacio ambiente X, su complemento CP es $X \backslash P$), para que añadir al problema las restricciones correspondientes a $y \notin \mathcal{Y} = y \notin CP$ sea equivalente a añadir $y \in P$, o lo que es lo mismo, las inecuaciones que definen P como politopo ya en LRA [2].

Como ya se ha discutido en varias ocasiones, codificar la red $y = f(x)$ consta de dos partes: una lineal (afín) y otra no lineal (activación). Podemos codificar la parte afín neurona a neurona [1, 2] de forma inmediata mediante las ecuaciones

$$\hat{z}_{i,j} = w_{i,j} z_{i-1} + b_{i,j} \quad \forall i \in \{1, \ldots, n\}, \forall j \in \{1, \ldots, k_i\} \tag{5.4}$$

donde, como se vio en la notación del Capítulo 2, $\hat{z}_{i,j}$ es la salida de la j-ésima neurona de la capa i antes de ser activada.

La codificación de la función de activación es, de nuevo, un desafío, cuyas posibles soluciones son las que dan lugar a los distintos algoritmos que se estudiarán en las próximas secciones. En el caso de Reluplex (Sección 5.3), la función ReLU se codifica como una disyunción de fórmulas, pues se aborda el problema 5.3 como un problema SMT, mientras que en NSVerify y MIPVerify (Sección 5.4), la activación será un conjunto de restriccio-

nes LRA, pues el problema 5.3 se trata como un problema MILP (*Mixed Integer Linear Programming*).

Definición 5.2.1. *Un **problema SAT** (problema de SATisfiabilidad booleana) es un problema que consiste en decidir si, dada una fórmula en lógica proposicional ϕ, existe una asignación de valores $A : \{x_1, \ldots, x_n\} \to \{\top, \bot\}$ para sus variables que hace que la expresión sea satisfactible, es decir, evalúe como verdadero: $\exists A$ tal que $\phi(A) = \top$. Si el resultado es favorable, decimos que el problema es* SAT*, y si no,* UNSAT.

Observación 5.2.2. *El problema SAT es el problema NP-completo clásico, pues fue el primero identificado como perteneciente a esta clase. Esto significa que (según el estado de la teoría de la computación actual), no puede resolverse en tiempo polinomial $\mathcal{O}(n^k)$. El problema SAT, en el peor de los casos, tiene complejidad algorítmica $\mathcal{O}(2^n)$, exponencial para el número de variables.*

Definición 5.2.3. *Un **problema SMT** (Satisfiability Modulo Theory) es una generalización del problema SAT que amplía el ámbito de la fórmula ϕ: en lugar de trabajar únicamente con lógica proposicional, se permiten teorías específicas como la aritmética, el álgebra lineal (LRA), las estructuras algebraicas, ciertas estructuras de datos, etc.*

Los algoritmos de esta sección tratan la resolución del problema 5.3 como un problema SMT cuya teoría es LRA. Este es un problema decidible NP-completo, cuyo tiempo algorítmico en el peor de los casos es exponencial. Si se tratase con álgebra no lineal, el tiempo pasaría a ser doble exponencial, y si se tratase con funciones trascendentes (como *tanh* o e^x), sería directamente un problema indecidible [1].

Tanto Reluplex como NSVerify y MIPVerify son completos siempre que las activaciones sean ReLU o PWL [1, 2]. Si sus activaciones no son PWL, será necesario sobreaproximarlas, tal y como se estudió en la Sección 2.3, perdiendo por tanto la completitud.

5.3. Reluplex

La función ReLU puede codificarse directamente como fórmula en FOL de la siguiente manera [1]:

$$(\hat{z}_{i,j} \leqslant 0 \implies z_{i,j} = 0) \wedge (\hat{z}_{i,j} > 0 \implies z_{i,j} = \hat{z}_{i,j}) \tag{5.5}$$

la cual, manipulando para eliminar los condicionales, se transforma en

$$(\hat{z}_{i,j} \leqslant 0 \wedge z_{i,j} = 0) \vee (\hat{z}_{i,j} > 0 \wedge z_{i,j} = \hat{z}_{i,j}) \tag{5.6}$$

Sin embargo, esta fórmula no puede ser añadida directamente al problema 5.3, pues contiene una disyunción, y los algoritmos de optimización solo admiten conjunciones de ecuaciones o inecuaciones lineales.

El problema consiste, por tanto, en encontrar una manera de eliminar la disyunción o, dicho en otras palabras, decidir en qué estado de activación se encuentra la ReLU para deshechar el otro. De esto se encarga el algoritmo Reluplex [4, 1, 2]. Su nombre es una combinación de las palabras *ReLU* y *Simplex*, e implementa una estrategia de búsqueda binaria con *backtracking* para decidir el estado de activación de cada neurona de la red.

La idea clave es que Reluplex llama al algoritmo del Simplex sobre el conjunto de restricciones que codifican el problema 5.3 para que compruebe su satisfactibilidad. Después, modifica los estados de activación de las ReLU en busca de una asignación que satisfaga todas las restricciones y genere, por tanto, un contraejemplo. En el algoritmo 1 se encuentra un pseudocódigo para Reluplex, de elaboración propia, basado en la explicación de [2]. Veámos como funciona.

Sea f una red neuronal y F el conjunto de igualdades definidas en 5.4 que codifican la parte lineal de la red, junto con las desigualdades que codifican la precondición $x \in \mathcal{X}$ y la negación de la postcondición $y \notin \mathcal{Y}$. Sea $\delta_{i,j} \in \{0, 1\}$ el estado de activación de la j-ésima neurona de la capa i, donde 0 significa que la neurona está *inactiva* ($\hat{z}_{i,j} \leqslant 0$) y 1 que está *activa* ($\hat{z}_{i,j} > 0$). Y, por último, sean los conjuntos $\mathcal{U}$ (*undetermined*), $\mathcal{A}$ (*activated*) y $\mathcal{N}$ (*not-activated*), los cuales contienen los índices (i, j) de las neuronas cuyo estado de activación es desconocido, activo, e inactivo, respectivamente.

El conjunto B de restricciones añadidas al problema estará formado por las restricciones básicas F junto con las de las ReLU determinadas en $\mathcal{A}$ y $\mathcal{N}$ [2]:

$$
\begin{aligned}
B = \{ & z_i, \hat{z}_i \mid z_0 \in \mathcal{X},\ z_n \notin \mathcal{Y}; \\
& \hat{z}_{i,j} = W_{i,j}\hat{z}_{i,j} + b_{i,j} \quad \forall i \in \{1, \dots, n\},\ \forall j \in \{1, \dots, k_i\}; \\
& z_{i,j} = \hat{z}_{i,j},\ \hat{z}_{i,j} > 0 \quad \forall (i,j) \in \mathcal{A}; \\
& z_{i,j} = 0, \quad \hat{z}_{i,j} \leqslant 0 \quad \forall (i,j) \in \mathcal{N}; \\
& z_{i,j} \geqslant 0, \quad z_{i,j} \geqslant \hat{z}_{i,j}\}
\end{aligned}
\tag{5.7}
$$

Observación 5.3.1. *Las variables de decisión sobre las que trabajará Simplex en 5.7 son $z_{i,j}$ y $\hat{z}_{i,j}$, y no hay función objetivo. Lo que se busca con la optimización es únicamente comprobar la factibilidad.*

En cada iteración, Reluplex trata de optimizar el modelo llamando al algoritmo del Simplex sobre B (línea 6). Si no existe solución factible (UNSAT), esa rama se cierra y se hace *backtracking* (línea 7). Si sí que se encuentra solución (SAT), puede haber ocurrido una de entre estas dos situaciones:

- Que todas las ReLU se satisfagan, esto es: $\forall (i, j) \in \mathcal{U},\ z_{i,j} = \max\{0, \hat{z}_{i,j}\}$. Entonces, el problema efectivamente tiene solución factible, y esta representa un contraejemplo para el problema de la verificación: la propiedad no se satisface (línea 10).
- Que exista al menos una ReLU no satisfecha, esto es: $\exists (i', j') \in \mathcal{U} \ : \ z_{i',j'} \neq \max\{0, \hat{z}_{i',j'}\}$. Reluplex intentará arreglar este nodo de dos formas: activándolo o desactivándolo. Esto ramifica el árbol de búsqueda en dos casos: en el primero, añade (i', j') a $\mathcal{A}$ (línea 15) y, en el segundo, lo añade a $\mathcal{N}$ (línea 16). En ambos lo elimina de $\mathcal{U}$, pues ya está determinado (línea 14).

El algoritmo termina cuando se encuentra un contraejemplo o cuando se agotan todas las ramas del árbol de búsqueda, devolviendo UNSAT, en cuyo caso la propiedad se verifica.

Observación 5.3.2. *El algoritmo Reluplex está diseñado para funcionar únicamente con activaciones ReLU; sin embargo, sus autores proponen que es posible adaptarlo para trabajar con cualquier función PWL [4]. Observamos que esto provocaría un aumento en el número de casos de la ramificación y, por tanto, su complejidad algorítmica.*

Algorithm 1 Reluplex

Entrada: conjuntos F, $\mathcal{U}$, $\mathcal{A}$ y $\mathcal{N}$
Salida: contraejemplo o verificación

Inicialización del conjunto B
while true **do**
    $r \leftarrow \text{Simplex}(B)$
    if r es UNSAT **then return** UNSAT → rama cerrada

    if $\forall (i,j) \in \mathcal{U},\ z_{i,j} = \max\{0, \hat{z}_{i,j}\}$ **then**
        return SAT → contraejemplo encontrado

    else
        Sea $(i',j') \in \mathcal{U}$ un nodo tal que $z_{i',j'} \neq \max\{0, \hat{z}_{i',j'}\}$
        $\mathcal{U} \leftarrow \mathcal{U} \setminus \{(i',j')\}$
        $r_1 \leftarrow \text{Reluplex}(F, \mathcal{U}, \mathcal{A} \cup \{(i',j')\}, \mathcal{N})$
        $r_2 \leftarrow \text{Reluplex}(F, \mathcal{U}, \mathcal{A}, \mathcal{N} \cup \{(i',j')\})$

        if $r_1 = r_2 =$ UNSAT **then return** UNSAT
        else return SAT

Reluplex realiza la búsqueda priorizando profundidad (*depth-first search*) y, en el peor de los casos, tendrá que realizar $2^{|\mathcal{U}|}$ iteraciones, siendo $|\mathcal{U}|$ el número de funciones ReLU en la red. Esto se puede mejorar si se emplea un algoritmo como el que se estudió en la Sección 4.2 para calcular los límites superior e inferior $\hat{u}_{i,j}$ y $\hat{l}_{i,j}$ de cada neurona antes de la activación. Si $0 \leqslant l_{i,j} \leqslant u_{i,j}$ entonces se tiene que $\delta_{i,j} = 1$, y si $l_{i,j} \leqslant u_{i,j} \leqslant 0$, $\delta_{i,j} = 0$. De esta manera, se puede realizar una asignación previa de algunas neuronas a los conjuntos $\mathcal{A}$ y $\mathcal{N}$, reduciendo así el número de ReLU a tener en cuenta en la fase de decisión [2].

5.4. NSVerify y MIPVerify

En esta sección, se estudian dos algoritmos que, al diferencia de Reluplex, sortean el problema de la disyunción en la ecuación 5.6 codificando las funciones ReLU directamente como conjuntos de inecuaciones, que darán lugar a un problema MILP.

Definición 5.4.1. *Un problema SMT, cuya teoría es LRA, es lo que se conoce como un* ***problema MILP*** *(Mixed Integer Linear Programming) [1]. Los problemas MILP involucran la búsqueda de valores para un conjunto de variables sujeto a restricciones lineales de las cuales algunas pueden ser forzadas a ser enteras.*

NSVerify [17] codifica la ReLU de cada una de las neuronas (i,j) de una red f de forma ingenua (*naive*) con el siguiente conjunto de inecuaciones [2]:

$$\begin{aligned} R_{i,j} = \{ & z_{i,j} \geqslant \hat{z}_{i,j}, \\ & z_{i,j} \geqslant 0, \\ & z_{i,j} \leqslant \hat{z}_{i,j} + M(1 - \delta_{i,j}), \\ & z_{i,j} \leqslant M\delta_{i,j} \} \end{aligned} \tag{5.8}$$

Observamos que se hace uso de las variables binarias de decisión $\delta_{i,j} \in \{0, 1\}$ introducidas en la sección anterior. Esto es porque las variables de decisión del optimizador en NSVerify serán $z_{i,j}$ y $\delta_{i,j}$, donde esta última está restringida a ser entera, pues es binaria.

Cuando el optimizador decide que $\delta_{i,j} = 0$, las ecuaciones en 5.8 son equivalentes a $z_{i,j} = 0$, y cuando $\delta_{i,j} = 1$, a $0 \leqslant \hat{z}_{i,j} = z_{i,j} \leqslant M$. Este comportamiento será exactamente el mismo que el de la función ReLU, si M es lo suficientemente grande [17]. De hecho, NSVerify será seguro y completo solo si M satisface [2]:

$$M \geqslant \max_{i,j}\{|\hat{u}_{i,j}|, |\hat{l}_{i,j}|\} \tag{5.9}$$

La variable M debe proporcionársele a NSVerify en su inicialización como una constante. Puede calcularse de forma previa utilizando algoritmos como el de propagación de hiperrectángulos, visto en la Sección 4.2.

El problema 5.3 a resolver está formado por las restricciones

$$\begin{aligned} B = \{&z_{i,j}, \delta_{i,j} \mid z_{0,j} \in \mathcal{X},\ z_{n,j} \notin \mathcal{Y}; \\ &\hat{z}_{i,j} = W_{i,j}\hat{z}_{i,j} + b_{i,j}, \quad R_{i,j}, \quad \forall i \in \{1, \ldots, n\},\ \forall j \in \{1, \ldots, k_i\}\} \end{aligned} \tag{5.10}$$

NSVerify utilizará un *solver* o *resolutor* MILP (como *Gurobi*[1]) para obtener la satisfactibilidad de B.

Veamos ahora de qué manera se puede refinar NSVerify para obtener un mejor rendimiento. Estas mejoras darán lugar a MIPVerify [14], otro algoritmo completo de la categoría de optimización pura.

MIPVerify, al igual que NSVerify, también resolverá un problema MILP de restricciones 5.10, solo que la forma de codificar las ReLU será ligeramente distinta a la dada por la ecuación 5.8.

$$\begin{aligned} R_{i,j} = \{&z_{i,j} \geqslant \hat{z}_{i,j}, \\ &z_{i,j} \geqslant 0, \\ &z_{i,j} \leqslant \hat{z}_{i,j} - \hat{l}_{i,j}(1 - \delta_{i,j}), \\ &z_{i,j} \leqslant \hat{u}_{i,j}\delta_{i,j}\} \end{aligned} \tag{5.11}$$

En esta ocasión, se ha sustituido la constante M por instancias de los límites para cada neurona antes de la activación $\hat{l}_{i,j}$ y $\hat{u}_{i,j}$. Este algoritmo requiere calcularlos de antemano.

Esta variante evita tener que correr el riesgo de perder la seguridad por culpa de una M que no sea lo suficientemente grande, y abre la puerta al diseño de algoritmos que utilicen una táctica basada en la relajación continua de la variable binaria $\delta_{i,j}$ [2]. Cuando esto se hace, la codificación en 5.11 es equivalente a la llamada relajación por triángulo vista en la Figura 4.5(c).

La otra mejora que introduce MIPVerify es el uso de una función objetivo. Tanto Reluplex como NSVerify utilizan optimizadores para comprobar la satisfactibilidad de 5.3, pero ninguno de ellos se aprovecha de su capacidad de minimizar o maximizar un objetivo. MIPVerify utiliza la función *perturbación máxima*:

$$\text{obj}(x, x_0) = ||x - x_0||_\infty \tag{5.12}$$

[1] *Gurobi Optimization LLC.* (2025), `https://www.gurobi.com`

Dada una entrada x_0, minimizar esta función significa hallar la máxima perturbación que puede sufrir x_0 para que $y \notin \mathcal{Y}$. La satisfactibilidad de la propiedad a verificar será determinada comparando la región de perturbación máxima permitida obtenida con $\mathcal{X}$. Se trata de un resultado de tipo *adversarial*, mencionados en la Sección 3.3.

Capítulo 6

Herramientas de verificación

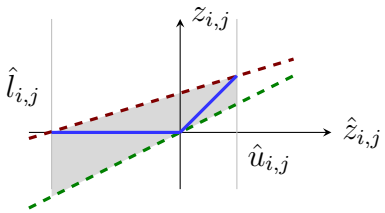
Este capítulo se enfoca en las implementaciones *estado-del-arte* de algoritmos de verificación de redes neuronales en librerías de código abierto, desarrolladas a partir del trabajo de los profesores Huan Zhang, Kaidi Xu y su equipo [18, 15, 19, 20]. Se comenzará con una introducción al algoritmo CROWN (6.1), el cual combina características de los enfoques de alcanzabilidad y optimización, y se verán sus principales variantes. A continuación, se explorará la librería auto_LiRPA (6.2), que implementa una automatización de CROWN para una amplia gama de arquitecturas de redes neuronales, y se destacarán sus principales contribuciones y casos de uso. Finalmente, se presentará el verificador α, β-CROWN (6.3), una extensión basada en auto_LiRPA que ha sido galardonada como la mejor herramienta de verificación de redes neuronales en competiciones internacionales recientes.

6.1. El algoritmo CROWN

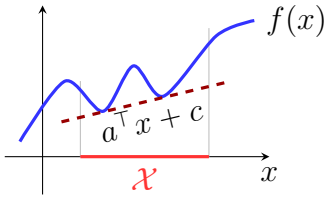
El algoritmo de verificación en el que se basan las implementaciones de las herramientas que veremos en este capítulo es CROWN [18], y fue propuesto por Huan Zhang y Tsui-Wei Weng en 2018. La notación ha sido adaptada a la empleada en esta memoria.

Se trata de un algoritmo de alcanzabilidad, pues el resultado que se obtiene de él son las cotas superiores e inferiores de los valores de cada neurona de salida, tal y como se obtenían de la propagación de h-rectángulos 4.2. La diferencia es que, lo que CROWN propaga, son límites (*bound propagation*), y lo hace hacia atrás: desde la capa de salida hasta la capa de entrada. Como añadido, algunas de las extensiones de CROWN implementan rutinas de optimización y esquemas de ramificación y corte, similares a los estudiados en los algoritmos de optimización del Capítulo 5.

Veamos cómo funciona [21, 18]. Sea $f : \mathbb{R}^{k_0} \longrightarrow \mathbb{R}^{k_n}$ una red neuronal de propagación directa, y sea la región de entrada $\mathcal{X}$. El objetivo de CROWN es encontrar la cota inferior y superior, f_L^* y f_U^*, de la función $f(x)$, con $x \in \mathcal{X}$. Se asumirá, sin pérdida de generalidad, que la capa de salida tiene una única neurona ($k_n = 1$), pues para varias salidas se aplica el algoritmo a cada una de ellas por separado; y que solo se busca la cota inferior, pues el problema para la cota superior es análogo. Además, por simplicidad, se ignorarán los cálculos a realizar con los sesgos.



(a)
Relajación de la ReLU mediante una pareja de cotas lineales.



(b)
Visualización en una dimensión de la cota inferior final de la red.

Figura 6.1: Adaptación de los diagramas en [21] a la notación utilizada en esta memoria.

La verificación será completa si se encuentra la cota exacta f_L^*. En su versión original, CROWN solo puede asegurar una estimación inferior f_L de esta (ecuación 6.1), por lo que es incompleto.

$$f_L \leqslant f_L^* = \min_{x \in \mathcal{X}} f(x) \tag{6.1}$$

Para hallar la cota, se comienza con la capa de salida $f(x) = y = z_n$, y se sustituye capa a capa hacia atrás, aplicando $z_i = \sigma_i(W_i z_{i-1} + b_i)$ (ecuación 2.4) hasta que se llega a la capa de entrada, consiguiendo que $f(x)$ dependa solo de x.

Ejemplo 6.1.1. *Sea f una red con dos capas de pesos W_1 y W_2, ambas con función de activación ReLU, y una capa final de pesos w_3 (vector fila) sin activación. Se comienza con $f(x) = z_3$, y se sustituye la variable z_i paso a paso hacia atrás. En la siguiente etapa, $f(x) = w_3^\top z_2$, y en la siguiente, $f(x) = w_3^\top \cdot \text{relu}(\hat{z}_2)$.*

El problema vuelve a ser las funciones de activación. Si la red solo contase con las transformaciones lineales dadas por los pesos W_i y los sesgos b_i, el cálculo se reduciría a una función lineal:

$$f(x) = W_n W_{n-1} \cdots W_1 x + c = a^\top x + c \tag{6.2}$$

donde c es el sesgo resultante final, y el vector $a^\top$ puede calcularse mediante una multiplicación de matrices. Esta idea es, precisamente, la que motiva la introducción de activaciones no-lineales, como se explicó en la Sección 2.3.

La idea clave de CROWN consiste en la relajación lineal de las funciones de activación en cada neurona mediante dos cotas lineales (ecuación 6.3), como se puede ver en la Figura 6.1(a). De este modo, podrán escribirse como matrices D_i para cada capa, que puedan multiplicarse con las matrices de pesos W_i en la ecuación 6.2. A cambio, por culpa de las desigualdades, se pierde la completitud.

$$\underline{a}_{i,j}\hat{z}_{i,j} + \underline{c}_{i,j} \leqslant \text{relu}(\hat{z}_{i,j}) = z_{i,j} \leqslant \overline{a}_{i,j}\hat{z}_{i,j} + \overline{c}_{i,j} \tag{6.3}$$

La matriz D_i será una matriz diagonal cuyos términos serán $\underline{a}_{i,j}$ o $\overline{a}_{i,j}$, dependiendo del signo de los elementos del vector $w^\top$ a su izquierda. Iterando este proceso se consigue avanzar hacia atrás acotando f y, al finalizar el proceso, se obtiene una cota lineal general para la red (ecuación 6.4) que solo depende de x. Puede verse un diagrama en 6.1(b).

$$f(x) \geqslant a^\top x + c \tag{6.4}$$

Ejemplo 6.1.2. *Siguiendo el ejemplo anterior, la cota lineal quedaría como*

$$f(x) \geqslant w_3 \top D_2 W_2 D_1 W_1 x + c = a^\top x + c$$

Por último, para hallar el valor de la cota inferior, se resuelve el problema de optimización lineal dado por:

$$f_L = \min_{x \in \mathcal{X}} a^\top x + c \leqslant \min_{x \in \mathcal{X}} f(x) \tag{6.5}$$

Ejemplo 6.1.3. *En un clasificador binario queremos que un conjunto de imágenes se clasifiquen con la etiqueta* $\{1\}$*, que se consigue si la salida de* f *es mayor que cero. Si* $f_L > 0$ *entonces* $f_L^* > 0$*, y sabremos que, aún en el peor de los casos, las imágenes serán etiquetadas con* $\{1\}$*.*

6.1.1. Alpha-CROWN

Alpha-CROWN [19] es una extensión del algoritmo CROWN que mejora las cotas lineales de la ecuación 6.3, haciéndolas más ajustadas.

Si la función a tratar es la ReLU, dados $\hat{l}_{i,j}$ y $\hat{u}_{i,j}$, la cota superior más ajustada posible para $z_{i,j}$ será, de forma inequívoca, la recta que pasa por los puntos $(\hat{l}_{i,j}, 0)$ y $(\hat{u}_{i,j}, \hat{u}_{i,j})$. Sin embargo, la cota inferior admite cierta flexibilidad: sirve cualquier recta que pase por el origen y tenga pendiente $\alpha_{i,j} \in [0, 1]$; véase el diagrama 6.1(a).

Lo que busca α-CROWN es optimizar la elección del parámetro α para encontrar la mejor cota final para la red. Para ello, se resuelve el problema de optimización:

$$\max_{0 \leqslant \alpha \leqslant 1} \min_{x \in \mathcal{X}} f_L(x, \alpha) \leqslant \min_{x \in \mathcal{X}} f(x) \tag{6.6}$$

donde, antes, $f_L(x) = \min_{x \in \mathcal{X}}(a^\top x + c)$ era la cota lineal calculada por CROWN, que ahora depende tanto de x como de α.

Cada función ReLU inestable permite la elección de su $\alpha_{i,j}$. En sus implementaciones, α-CROWN emplea el descenso de gradiente para escoger el óptimo en cada una de ellas, asegurando las cotas lineales más ajustadas posibles, y acercando la cota final f_L a f_L^* [19], aunque aún sin asegurar completitud.

6.1.2. Beta-CROWN

Beta-CROWN [20] es otra extensión para CROWN que utiliza un esquema de ramificación y poda (*branch and bound*) para conseguir la convergencia de f_L a f_L^*, y alcanzar así la completitud.

El primer paso que realiza β-CROWN es la ramificación (*branching*). Cada ReLU inestable se separa en dos subproblemas: uno con la ReLU activada, y otro con la ReLU sin activar (similar a Reluplex 5.3), y se añade en cada caso la restricción adicional $\hat{z} > 0$ o $\hat{z} \leqslant 0$ que corresponda.

El segundo paso consiste en calcular los límites para cada subproblema (*bounding*), para lo que se utiliza un *solver* de problemas lineales. Si la cota inferior conseguida para la red

no entra dentro de lo esperado (en el ejemplo del clasificador lineal, $f_L < 0$), entonces, de nuevo se separa en dos casos, ramificando para refinar la cota hasta que esta sea óptima. Las ramas que sí cumplen la condición son cerradas, *podadas*. Este proceso continúa, mejorando con cada iteración la cota inferior f_L calculada, que acabará convergiendo a f_L^*, consiguiendo así la completitud [20].

El problema es que, el algoritmo base, CROWN, no puede manejar las restricciones de ramificación $\hat{z} > 0$ y $\hat{z} \leqslant 0$, pues no son matrices que pueda multiplicar con las W_i. Por ello, de forma similar a como se hacía en α-CROWN, se añaden las matrices diagonales S_i de términos $s_{j,j} \in \{0, 1, -1\}$ y los parámetros optimizables β, provenientes de la propagación de multiplicadores Lagrangianos [20, 21].

La cota final sobre la salida de la red f obtenida será

$$f_L = \max_{\beta \geqslant 0} \min_{x \in \mathcal{X}} (a + P\beta)^\top x + q^\top \beta + c \leqslant \min_{x \in \mathcal{X}} f(x) \tag{6.7}$$

La optimización depende ahora del parámetro β también. Puede utilizarse el descenso de gradiente para resolverlo, igual que en α-CROWN.

6.2. Librería auto_LiRPA

Auto_LiRPA[1] [15] es una herramienta para el cálculo automático de cotas de redes neuronales, basada en el análisis de perturbaciones mediante relajación lineal (LiRPA, *Linear Relaxation based Perturbation Analysis*). Se trata de una librería de `Python` en desarrollo, diseñada para integrarse con la biblioteca de aprendizaje automático `PyTorch`, y facilitar la certificación de modelos y la verificación de su robustez de manera eficiente.

Se basa en la extensión y generalización de algoritmos LiRPA existentes (como CROWN [18] o IBP [22]), para operar sobre cualquier grafo computacional, en particular, redes neuronales. Auto_LiRPA se encarga de computar cotas garantizadas para los nodos de salida del grafo, en base a las restricciones (perturbaciones) dadas por el usuario sobre los nodos de entrada.

El algoritmo principal que implementa auto_LiRPA es CROWN, con su extensión α-CROWN, en sus versiones *backward* (como vimos en la Sección 6.1), y *forward*, aunque también incluye otros métodos como IBP (*Interval Bound Propagation*).

Las principales contribuciones de auto_LiRPA son:

- **Facilidad de uso**: el objetivo de Huan Zhang y su equipo es el de desarrollar una herramienta de verificación de redes neuronales que fuese útil y accesible, incluso para usuarios sin experiencia en los aspectos técnicos de LiRPA y su implementación eficiente.
- **Flexibilidad y generalidad**: antes de este trabajo, las implementaciones de LiRPA eran específicas para cada arquitectura y poco escalables [15]. Con Auto_LiRPA se pueden conseguir verificaciones de robustez de vanguardia para redes avanzadas, como DenseNet, ResNeXt y *Transformers*.

[1] `https://github.com/Verified-Intelligence/auto_LiRPA`

- **Compatibilidad con perturbaciones generales**: los enfoques anteriores, tradicionalmente, se limitaban a implementaciones con restricciones $\mathcal{X}$ sobre la entrada en forma de p-bolas.
- **Introducción de la técnica *loss fusion*** [15]: optimiza el cálculo de las cotas mediante LiRPA *durante* el entrenamiento, lo que permite realizar entrenamiento certificado (*certified training*, Sección 7.3) de redes grandes en costes de tiempo y memoria asumibles.
- **Aplicación en optimización de paisajes planos**: facilita el análisis de la variación de la función de pérdida según los parámetros de entrada, y apoya el entrenamiento de redes con paisajes planos para que generalicen mejor, es decir, que hagan predicciones más robustas sobre datos no conocidos (Seccion 7.3).

En esta memoria se ha optado por explorar la potencia de una herramienta profesional como auto_LiRPA, en lugar de implementar métodos desde cero. De este modo, es posible ilustrar la verificación de redes neuronales *estado-del-arte*, y estudiar modelos avanzados, desarrollados y optimizados por equipos de investigación de vanguardia.

6.2.1. Código, ejemplos y *benchmark*

Para hacer uso de las herramientas implementadas en auto_LiRPA, primero debe cargarse un modelo de red neuronal definido con la clase `torch.nn.Module`, y proporcionar un conjunto de datos de entrada, que puede ser, por ejemplo, una imagen de la que queremos comprobar robustez frente a perturbaciones:

```
modelo  = inicializarModelo()  #cargar un modelo de torch.nn
entrada = cargar_datos_entrada()
```

A continuación, se encapsula el modelo en un objeto de la clase `BoundedModule`, que envuelve un modelo de `PyTorch` para habilitar el cálculo automático de las cotas; se define la región de entrada permitida $\mathcal{X}$ mediante un objeto de la clase `Perturbation`, por ejemplo, `PerturbationLpNorm`; y se aplica la perturbación a la entrada para convertirla en un objeto de clase `Tensor`:

```
modelo       = BoundedModule(modelo, entrada)
perturbacion = PerturbationLpNorm(norm=np.inf, eps=0.1)
entrada      = BoundedTensor(entrada, perturbaciones)
```

Por último, se calculan las cotas inferior `lb` $= f_L$ y superior `ub` $= f_U$ con el método `compute_bounds`, que permite, gracias a su parámetro `method`, seleccionar distintos verificadores para obtener los límites garantizados. Algunos de ellos son `'IBP'`, `'CROWN'`, `'CROWN-IBP'`, `'Forward'` y `'alpha-CROWN'`:

```
lb, ub = modelo.compute_bounds(x=(entrada,), method="CROWN")
```

En el repositorio de *GitHub* 1 de la herramienta se encuentran varios ejemplos prácticos de uso, donde se muestra la aplicación de auto_LiRPA a distintos tipos de redes.

Auto_LiRPA se ha probado en *datasets* como CIFAR-10[2] o Tiny-ImageNet[3], que son puntos de referencia en el entrenamiento de redes neuronales, y se ha implementado sobre arquitecturas complejas como DenseNet o ResNeXt [15]. Hasta la publicación de la herramienta, aplicar LiRPA en ese tipo de redes para *datasets* tan extensos resultaba intratable por tiempo y memoria. En el Cuadro 6.1 puede verse cómo, gracias a *loss fusion*, se consiguen tiempos tan solo unas pocas veces mayores que el de entrenar la red sin verificación.

Dataset	Entrenamiento	Natural	IBP	LiRPA w/LF
CIFAR-10	DenseNet	22.07	54.40 (2.46×)	90.79 (4.11×)
	ResNeXt	17.48	32.44 (2.20×)	55.84 (3.78×)
Tiny-ImageNet	DenseNet	135.17	318.77 (2.36×)	513.96 (3.80×)
	ResNeXt	92.63	191.34 (2.07×)	337.83 (3.65×)

Cuadro 6.1: Tiempos de reloj (en segundos) de entrenamiento por época de dos arquitecturas distintas para los *datasets* CIFAR-10 y Tiny-ImageNet, comparando el entrenamiento sin verificación (natural) con los algoritmos de verificación certificada IBP y LiRPA con *Loss Fusion*. Extracto de [15].

Además, se ha comparado su desempeño con otras implementaciones *estado-del-arte* como IBP, obteniendo resultados similares, y en algunos casos, incluso mejorándolos [15]. En el Cuadro 6.2 puede verse una comparativa.

Dataset	Error	DenseNet		ResNeXt	
		IBP	auto_LiRPA	IBP	auto_LiRPA
CIFAR-10	Estándar	57.21 %	56.03 %	56.32 %	53.85 %
	Verificada	69.59 %	67.57 %	70.41 %	68.25 %
Tiny-ImageNet	Estándar	78.40 %	77.96 %	78.58 %	78.58 %
	Verificada	86.87 %	85.44 %	86.95 %	86.95 %

Cuadro 6.2: Tasas de error de los modelos IBP y auto_LiRPA en entrenamiento certificado de las arquitecturas complejas DenseNet y ResNeXt en distintos *datasets*. Se comparan errores sin ataques adversariales (Estándar) y bajo evaluación certificada (Verificada). Extracto de [15].

6.3. Verificador alpha-beta-CROWN

Mientras que auto_LiRPA es un conjunto de herramientas para el cálculo de cotas y el entrenamiento certificado de grafos computacionales, α, β-CROWN[4] es un verificador de redes neuronales completo y eficiente, que utiliza auto_LiRPA como base para verificar redes neuronales de diversas arquitecturas. De hecho, ambos proyectos son actualmente liderados por Huan Zhang.

[2] `https://www.cs.toronto.edu/~kriz/cifar.html`
[3] `https://paperswithcode.com/dataset/tiny-imagenet`
[4] `https://github.com/Verified-Intelligence/alpha-beta-CROWN`

El verificador α, β-CROWN logra la completitud gracias a la combinación de las mejoras en las cotas que proporciona α-CROWN (6.1.1) con la convergencia a las cotas exactas alcanzada mediante *branch and bound* con β-CROWN (6.1.2). Su implementación optimizada es eficiente, y permite acelerar los cálculos haciendo uso de GPUs (*Graphics Processing Units*).

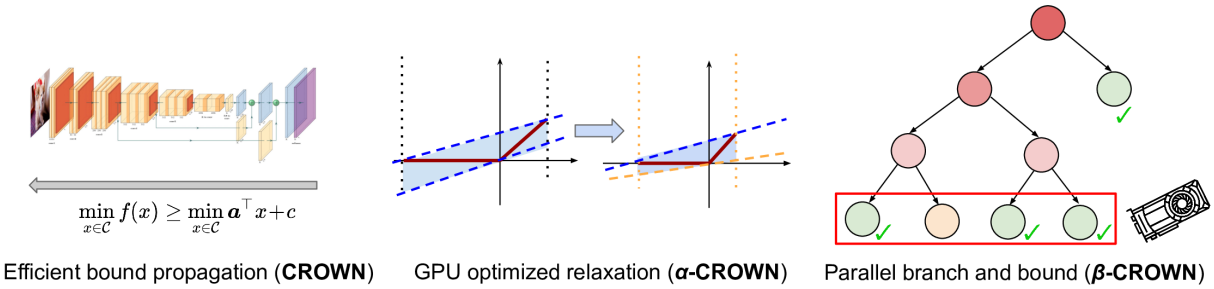


Figura 6.2: Algoritmo α, β-CROWN. Tomada de [21].

Actualmente, α, β-CROWN es el campeón de la Competición Internacional de Verificación de Redes Neuronales (VNN-COMP[5]), premio que ganó en 2024 por cuarto año consecutivo, obteniendo el primer puesto en todos los *benchmarks* propuestos. Superó a sus rivales tanto en precisión de las verificaciones como en tiempo de procesamiento; en la Figura 6.3 puede verse que α, β-CROWN es varios órdenes de magnitud más rápido que las demás implementaciones, aumentando así su escalabilidad a redes más grandes.

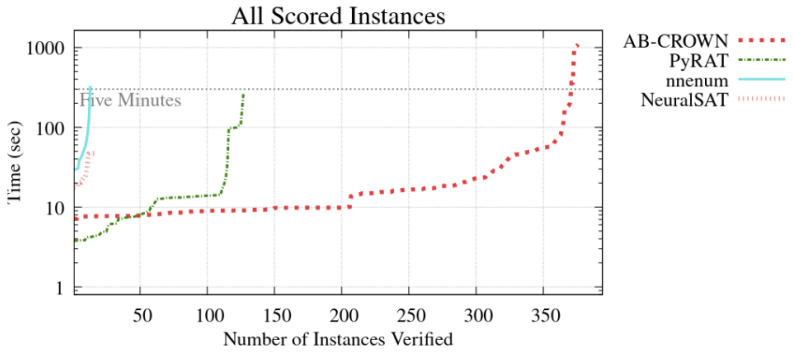


Figura 6.3: Tiempo en segundos por número de instancias verificadas. Tomado de los resultados de la VNN-COMP 24[5].

[5] https://sites.google.com/view/vnn2024

Capítulo 7

Conclusiones y Trabajo Futuro

En este capítulo, se pretende dar una visión global de las conclusiones a las que se ha llegado durante el proceso de investigación que ha llevado a escribir esta memoria. Esto servirá para descubrir las líneas de investigación y trabajo futuro que quedan abiertas, y que podrían ser la base para próximos proyectos.

7.1. Conclusiones

La definición formal del problema de verificación resulta crucial para el diseño de algoritmos de verificación, ya que no solo establece el punto de partida, sino que actúa como guía para el desarrollo, precisando las condiciones de las que se parte y el resultado que se espera obtener. Un pequeño cambio en la formulación del problema, tal y como se ha visto en las ecuaciones 3.1 y 3.2, ha dado lugar a dos enfoques totalmente diferentes, cada uno con sus propios desafíos técnicos y estrategias de resolución.

En particular, uno de los mayores obstáculos en la verificación de redes neuronales radica en la no linealidad introducida por las funciones de activación, tema que se ha abordado desde distintos puntos de vista a lo largo de este trabajo. Tal y como se ha visto en los Capítulos 4 y 5, gran parte del diseño de algoritmos de verificación gira en torno a la manera de hacer más eficiente el tratamiento y representación de las funciones de activación.

Desde un punto de vista computacional, se concluye que la verificación de redes neuronales es un problema *NP-completo*, al menos si se busca la completitud. Esto se ha visto reflejado en el recorte de las regiones en *ExactReach*, la ramificación en *Reluplex* y α, β-CROWN, o la resolución de problemas SMT en los algoritmos del Capítulo 5. Todos estos algoritmos tienen complejidad $\mathcal{O}(2^n)$, siendo n el número total de neuronas.

Si se quiere mejorar la eficiencia y reducir la complejidad algorítmica, es necesario sacrificar la completitud realizando sobreaproximaciones, relajaciones continuas, o tomando límites poco ajustados.

Por otro lado, las herramientas actuales de verificación de redes neuronales, como α, β-CROWN, han demostrado su eficiencia en redes con miles de neuronas [15], y probado

su superioridad frente a otras alternativas previas. Sin embargo, aún no cuentan con la escalabilidad necesaria como para aplicarse a la verificación de redes neuronales de última generación, como *GPT-3*, que cuenta con 175 mil millones de parámetros[1].

7.2. Trabajos relacionados

A continuación, se presentan algunos trabajos clave que han servido como referencia para el desarrollo de esta memoria, ya sea como base conceptual, punto de comparación o fuente de algoritmos específicos.

- **Introduction to Neural Network Verification**, Aws Albarghouthi [1]. Su enfoque divulgativo y tono didáctico han sido útiles para comprender las líneas de pensamiento que dan lugar a los algoritmos de verificación.
- **Algorithms for Verifying Deep Neural Networks**, Changliu Liu et al. [2]. Este libro presenta un análisis detallado de 18 algoritmos de verificación de redes neuronales, entre los que se encuentran la mayoría de los estudiados en esta memoria. Su estructura sistemática ha facilitado la comparación entre distintos métodos y su entendimiento en profundidad.
- **Automatic perturbation analysis for scalable certified robustness and beyond**, Kaidi Xu et al. [15]. Se trata del artículo al que va asociada la herramienta auto_LiRPA.

7.3. Trabajo futuro

En cuanto a las líneas de investigación que quedan abiertas, sobre las que se podrían desarrollar nuevos trabajos en el futuro, destacan las siguientes:

- **Entrenamiento adversarial**: se trata de un enfoque basado en la integración de los algoritmos de verificación directamente en el entrenamiento de la red, para conseguir redes neuronales más robustas y fáciles de verificar posteriormente, pues han sido entrenada mediante ataques adversariales [1, 15].
- **Optimización heurística**: se plantea el uso de algoritmos de enjambres de partículas (PSO, del inglés, *Particle Swarm Optimization*) y de algoritmos genéticos para realizar un ajuste de restricciones e hiperparámetros eficiente, o para la búsqueda efectiva de contraejemplos en forma de entradas que maximicen la función de coste de la red [23].

[1]Cifra proporcionada por el propio ChatGPT citando `https://arxiv.org/pdf/2005.14165`.

Bibliografía

[1] Aws Albarghouthi. *Introduction to Neural Network Verification*. En: *CoRR* abs/2109.10317 (2021).

[2] Changliu Liu et al. *Algorithms for Verifying Deep Neural Networks*. En: *Foundations and Trends in Optimization* 4 (2021), págs. 244-404. DOI: 10.1561/2400000035.

[3] Rishal Hurbans. *Grokking Artificial intelligence algorithms*. Manning Publications, 2020.

[4] Guy Katz et al. *Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks*. En: *CoRR* abs/1702.01135 (2017).

[5] Robert W. Floyd. *Assigning meanings to programs*. En: *Proceedings of the American Mathematical Society* (1967).

[6] C. A. R. Hoare. *An axiomatic basis for computer programming*. En: *Communications of the ACM* (1969).

[7] Ian Goodfellow, Yoshua Bengio y Aaron Courville. *Deep Learning*. MIT Press, 2016.

[8] Frank Rosenblatt. *The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain*. En: *Psychological Review* (1958).

[9] Marvin Minsky y Seymour A. Papert. *Perceptrons*. Cambridge, MA: MIT Press, 1969.

[10] David E Rumelhart, Geoffrey E Hinton y Ronald J Williams. *Learning Representations by Back-Propagating Errors*. En: *Nature* (1986).

[11] Ashish Vaswani et al. *Attention Is All You Need*. En: *CoRR* abs/1706.03762 (2017).

[12] Weiming Xiang, Hoang-Dung Tran y Taylor T. Johnson. *Reachable Set Computation and Safety Verification for Neural Networks with ReLU Activations*. En: *CoRR* abs/1712.08163 (2017).

[13] Timon Gehr et al. *AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation*. En: *IEEE Symposium on Security and Privacy (SP)*. 2018.

[14] Vincent Tjeng y Russ Tedrake. *Evaluating robustness of neural networks with mixed integer programming*. En: *CoRR* abs/1711.07356 (2017).

[15] Kaidi Xu et al. *Automatic perturbation analysis for scalable certified robustness and beyond*. En: *Advances in Neural Information Processing Systems* 33 (2020).

[16] Pauli Virtanen et al. *SciPy 1.0: Fundamental algorithms for scientific computing in Python*. En: *Nature Methods* 17.3 (2020), págs. 261-272.

[17] Alessio Lomuscio y Lalit Maganti. *An approach to reachability analysis for feed-forward ReLU neural networks*. En: *CoRR* abs/1706.07351 (2017).

[18] Huan Zhang et al. *Efficient Neural Network Robustness Certification with General Activation Functions*. En: *Advances in Neural Information Processing Systems* 31 (2018), págs. 4939-4948.

[19] Kaidi Xu et al. *Fast and Complete: Enabling Complete Neural Network Verification with Rapid and Massively Parallel Incomplete Verifiers*. En: *International Conference on Learning Representations*. 2021.

[20] Shiqi Wang et al. *Beta-CROWN: Efficient bound propagation with per-neuron split constraints for complete and incomplete neural network verification*. En: *Advances in Neural Information Processing Systems* 34 (2021).

[21] Huan Zhang et al. *Formal Verification of Deep Neural Networks: Theory and Practice*. 2022. URL: `https://www.youtube.com/watch?v=-EKQhkMHWVU`.

[22] Yuhao Mao et al. *Understanding Certified Training with Interval Bound Propagation*. 2024.

[23] T. Liu y X. Wang. *A Particle Swarm Optimization Algorithm for Finding Counterexamples in Neural Networks*. En: *Proceedings of the International Conference on Neural Information Processing* (2020).