UACM

Universidad Autónoma de la Ciudad de México

NADA HUMANO ME ES AJENO

COLEGIO DE CIENCIA Y TECNOLOGÍA

LICENCIATURA EN MODELACIÓN MATEMÁTICA

Uso de una red neuronal para detectar autoría de textos

T E S I S

QUE PARA OPTAR POR EL TÍTULO DE

LICENCIADO EN MODELACIÓN MATEMÁTICA

P R E S E N T A

EDUARDO ABRAHAM DÍAZ PÉREZ

D I R E C T O R **MTRO. MARCO ANTONIO PRADO ZÁYAGO**

C O D I R E C T O R **DR. PABLO PADILLA LONGORIA**

Ciudad de México, junio de 2026.

UACM
Universidad Autónoma de la Ciudad de México
Nada humano me es ajeno

CA Coordinación Académica

SISTEMA BIBLIOTECARIO DE INFORMACIÓN Y DOCUMENTACIÓN

UNIVERSIDAD AUTÓNOMA DE LA CIUDAD DE MÉXICO
COORDINACIÓN ACADÉMICA

RESTRICCIONES DE USO PARA LAS TESIS DIGITALES

DERECHOS RESERVADOS©

Resumen

Resumen

En el ámbito de la filología, la atribución de autoría de textos antiguos o de autoría incierta representa un desafío que tradicionalmente requiere la intervención de especialistas. Este trabajo explora el uso de Redes Neuronales Artificiales (RNA), específicamente redes con compuertas Long Short-Term Memory (LSTM), como herramienta auxiliar para la detección de autoría de textos literarios en español. Se entrenó un modelo con 251 cuentos de tres autores latinoamericanos contemporáneos: Jorge Luis Borges, Mario Benedetti y Julio Cortázar. La metodología incluyó la extracción automatizada de textos, limpieza de datos (eliminación de encabezados, pies y palabras estructurales), tokenización, padding y generación de embeddings. El modelo se implementó en Python con TensorFlow y se ejecutó en entornos de cómputo de recursos limitados (Google Colaboratory). Los resultados alcanzaron una precisión máxima del 84% y un promedio del 75% en la clasificación de textos de prueba. Se identificaron limitaciones relacionadas con el tamaño reducido del corpus, la disparidad en las longitudes de los textos y la eliminación de palabras estructurales. No obstante, las pruebas con datos no etiquetados mostraron un desempeño congruente con la precisión obtenida durante el entrenamiento. Se concluye que las redes LSTM constituyen una herramienta viable y práctica para apoyar tareas filológicas, especialmente cuando se dispone de conjuntos de datos limitados y heterogéneos, aunque se requieren refinamientos futuros como el balanceo de clases, la ampliación del vocabulario y la implementación de umbrales de confianza.

Palabras clave: atribución de autoría, redes neuronales artificiales, LSTM, procesamiento de lenguaje natural, estilo literario, Borges, Benedetti, Cortázar.

Abstract

In the field of philology, authorship attribution of ancient or uncertain texts represents a challenge that traditionally requires the intervention of specialists. This work explores the use of Artificial Neural Networks (ANN), specifically Long Short-Term Memory (LSTM) networks, as an auxiliary tool for authorship detection of Spanish literary texts. A model was trained with 251 short stories by three contemporary Latin American authors: Jorge Luis Borges, Mario Benedetti, and Julio Cortázar. The methodology included automated text extraction, data cleaning (removal of headers, footers, and stopwords), tokenization, padding, and embedding generation. The model was implemented in Python using TensorFlow and executed on limited-resource computing environments (Google Colaboratory). Results achieved a maximum accuracy of 84% and an average of 75% on test set classification. Limitations related to the small corpus size, text length disparity, and stopword removal were identified. Nevertheless, tests with unlabeled data showed performance consistent with the training accuracy. It is concluded that LSTM networks constitute a viable and practical tool to support philological tasks, especially when dealing with limited and heterogeneous datasets, although future refinements such as class balancing, vocabulary expansion, and confidence thresholds are required.

Keywords: authorship attribution, artificial neural networks, LSTM, natural language processing, literary style, Borges, Benedetti, Cortázar.

Dedicatoria

A Yetlanezi Rodríguez,

Porque en los momentos de incertidumbre, tu presencia es certeza; cuando el cansancio merma mis sentidos, tu compañía es aliento. Este trabajo no habría sido posible sin tu paciencia infinita, tu escucha sincera y tu amor constante. Cada página lleva un poco de tu luz. Gracias por siempre estar.

Y también a quienes, desde la distancia o el silencio, me enseñaron que nunca he estado solo en el sendero.

Agradecimientos

Con profunda estima y sincero reconocimiento, deseo expresar mi más sentido agradecimiento a mis tutores, el Mtro. Marco Antonio Prado Záyago y el Dr. Pablo Padilla Longoria. Su invaluable orientación, su paciencia y su apoyo incondicional fueron el faro que guió este trabajo hasta su término. Sin su entrega y sabiduría, estas páginas no habrían visto la luz.

Extiendo mi gratitud a los doctores Carlos Islas Moreno y Felipe Humberto Contreras Alcalá, cuyas agudas observaciones y comentarios siempre atinados enriquecieron tanto la forma como el fondo de esta investigación. Su rigor académico y su calidez humana dejaron una huella profunda en este trabajo.

Quiero agradecer de manera especial al Mtro. David Fernando Vizuet Morales y al Dr. Enrique Espinoza Loyola, quienes me acompañaron durante toda mi formación. Su paciencia, su dedicación y la claridad de sus enseñanzas fueron la base sobre la que construí no solo esta tesis, sino también mi identidad como modelador matemático.

A la Universidad Autónoma de la Ciudad de México, mi más profundo reconocimiento. Esta casa de estudios ha sido un crisol de oportunidades, un espacio donde cientos de estudiantes —entre los que me incluyo— hemos recibido educación gratuita y de calidad. Gracias por creer en el proyecto de una universidad pública transformadora.

Agradezco también al Laboratorio de Matemáticas del plantel Cuautepec, donde encontré no solo los recursos técnicos para trabajar, sino un ambiente de colaboración y apoyo. El Mtro. Marco Prado, siempre atento y generoso, nos brindó a mis compañeros y a mí la orientación y el ánimo necesarios para culminar nuestra licenciatura.

Finalmente, a mis compañeros y compañeras de la Licenciatura en Modelación Matemática: gracias por compartir conmigo más que aulas y pizarras. Gracias por las largas jornadas de estudio, por las dudas compartidas, por los logros celebrados y por la amistad que hizo más llevadero este largo trayecto. Este logro también es de ustedes.

A todos, mi eterna gratitud.

Índice general

Introducción

Motivación

El presente trabajo surge de la confluencia de dos pasiones académicas que, en apariencia, pertenecen a mundos distintos: la literatura y las matemáticas. Por un lado, mi formación previa en Creación Literaria y en Letras Clásicas me ha permitido desarrollar una sensibilidad por el lenguaje, el estilo y las particularidades de la escritura de diversos autores. Por otro lado, mi reciente interés por la modelación matemática y el aprendizaje automático ha abierto la posibilidad de abordar problemas tradicionalmente humanísticos desde una perspectiva cuantitativa y computacional.

La inquietud inicial que dio origen a esta investigación fue una pregunta aparentemente simple: ¿puede una máquina aprender a reconocer el estilo literario de un autor de la misma manera que lo hace un filólogo experimentado? Durante mis estudios de licenciatura, al trabajar con textos de autores latinoamericanos, observé la dificultad que implica la clasificación manual de textos cuando no hay suficiente evidencia contextual o histórica. Esta experiencia me llevó a preguntarme si las herramientas de aprendizaje automático, particularmente las Redes Neuronales Artificiales, podrían ofrecer un soporte objetivo y sistemático para esta tarea.

Fue en este punto donde la guía de mis asesores y profesores resultó fundamental. En principio, estaba buscando adentrarme en el mundo de la ciencia y el análisis de datos con métodos computacionales. Sin embargo, a través de diversas conversaciones y revisiones bibliográficas, logramos identificar un nicho de oportunidad para conjuntar ambas áreas de conocimiento: la atribución de autoría mediante redes neuronales. Así, este trabajo se concibe como un puente entre las humanidades y las ciencias exactas, un intento de demostrar que el análisis literario y la modelación matemática no son disciplinas antagónicas, sino que pueden complementarse para ofrecer nuevas herramientas de análisis.

La presente investigación es, en esencia, un primer paso para explorar esta posibilidad, con la esperanza de que sirva como punto de partida para futuros trabajos interdisciplinarios.

Planteamiento del problema

En el área de la filología existe un problema parcialmente resuelto: cuando se encuentran textos antiguos o de autoría incierta, muchas veces no se conoce al autor. La atribución de autoría es una tarea que tradicionalmente recae en filólogos experimentados, quienes, a través de su vasto conocimiento sobre el estilo y las particularidades de cada escritor, pueden determinar si un fragmento o un texto completo puede ser atribuido a un autor específico. Sin embargo, este proceso es lento, subjetivo en cierta medida y requiere de un conocimiento especializado que no siempre está disponible.

En literatura y lingüística, el concepto de "estilo" no es algo fácil de definir. Se entiende por "estilo" la manera particular en la que un hablante utiliza la lengua para comunicarse, ya sea de forma oral, ya de forma escrita. Sin embargo, esta manera particular, que Ferdinand de Saussure denominó con el término "habla", está atravesada por muchos factores que la determinan: el contexto cultural del hablante, su disponibilidad léxica, sus referentes, su entorno, entre otras cosas. Cuando hablamos del lenguaje escrito, en particular en textos literarios, dicha "habla" también se verá atravesada por los temas estéticos particulares que mueven la sensibilidad del autor. En todos estos detalles debe poner atención un experto en literatura cuando estudia la obra particular de un autor, una época, una cultura o un movimiento. Una combinación de todos esos elementos es lo que lleva a determinar el "estilo" y a poder reconocer cuándo otro autor intenta imitar o emular su estilo. Sin embargo, un filólogo experimentado es capaz de aprender a reconocer incluso estas minucias para identificar con mucha precisión si un texto es de la autoría del escritor al que se dedica a estudiar, o si, por el contrario, pertenece al catálogo de sus obras espurias.

La labor filológica, como ha de esperarse, es un trabajo complejo que requiere conocer a cabalidad la obra de un autor, además de tener un dominio muy profundo de la lengua a nivel estructural e histórico para poder detectar minucias técnicas que son imperceptibles para el resto de los hablantes. Por lo anterior, constituye un desafío técnico considerable lograr que una computadora, a través de funciones matemáticas simples, realice aunque sea alguno de los escrutinios a los que un filólogo somete un texto para catalogarlo.

En las últimas décadas, han surgido enfoques computacionales para abordar este problema. Los métodos tradicionales de atribución de autoría se basaban en la extracción manual de características lingüísticas, como la elección de palabras, la frecuencia de términos, el uso de signos de puntuación o la longitud de las oraciones (Huertas-Tato et al., 2022). Estos enfoques, si bien lograban resultados aceptables en contextos controlados, presentan limitaciones importantes: son poco flexibles ante la aparición de nuevos autores y no logran capturar características estilísticas que no fueron consideradas inicialmente en el diseño del modelo (Huertas-Tato et al., 2022).

Con el desarrollo de las nuevas tecnologías, particularmente en el campo del procesa-

miento del lenguaje natural y el aprendizaje automático, surge la posibilidad de entrenar sistemas computacionales que asistan en esta tarea. En particular, las Redes Neuronales Artificiales (RNA), y más específicamente las redes con compuertas LSTM (Long Short-Term Memory), han demostrado ser herramientas útiles para el reconocimiento de patrones complejos en datos textuales secuenciales (Uquillas Trujillo & Moposita Lasso, 2025). Estas redes han sido aplicadas exitosamente en tareas como análisis de sentimiento en español (Uquillas Trujillo & Moposita Lasso, 2025), clasificación de discurso de odio (Kovács et al., 2021) y atribución de autoría en otros idiomas (Gagala, 2018).

Existen en la literatura algunos trabajos que abordan la predicción de autoría mediante redes neuronales, con resultados diversos como el trabajo de (Gagala, 2018), donde se compara el desempeño de una Red Neuronal Artificial con conjuntos de datos de cinco y veinte autores en inglés, francés, español y polaco. Otros estudios se centran en textos en otras lenguas, como la literatura china (Zhao et al., 2025) o la obra de Dante Alighieri en italiano (Bonomi, s/a), o bien emplean conjuntos de datos de gran tamaño. El presente trabajo se distingue por enfocarse en autores de habla hispana —específicamente, Jorge Luis Borges, Mario Benedetti y Julio Cortázar— con el objetivo de evaluar si la capacidad de predicción de la red es suficientemente adecuada para ser utilizada como herramienta auxiliar en la labor filológica, incluso cuando se dispone de un conjunto reducido de datos.

Importancia y resultados esperados

La ventaja de emplear una red neuronal radica en su capacidad para procesar una gran cantidad de datos en poco tiempo, lo que facilitaría la catalogación y el estudio de textos, especialmente de aquellos cuya autoría se desconoce. Aunque la red no sea capaz de determinar con total certeza el autor de un texto, sí puede ofrecer una guía que permita delimitar la búsqueda a unos pocos candidatos, agilizando así el trabajo del filólogo.

Este estudio constituye un primer intento de entrenar una red neuronal con un conjunto reducido de datos, evaluando su capacidad para predecir adecuadamente en condiciones que, en principio, podrían ser insuficientes para el entrenamiento de una red. Esta es una contribución relevante, pues como señalan Huertas-Tato et al. (2022), uno de los principales desafíos en atribución de autoría es la capacidad de generalización a nuevos dominios y conjuntos de datos, y el tamaño reducido de los corpus es una limitación frecuente en aplicaciones prácticas. Los resultados de este trabajo permitirán determinar si el enfoque propuesto es viable como herramienta auxiliar en contextos donde los datos son escasos, como suele ocurrir con textos históricos o de archivo.

Además, este trabajo busca corroborar si el preprocesamiento de los datos ayuda a mejorar el rendimiento de la Red Neuronal Artificial y reducir el costo computacional reflejado en el tiempo de compilación. Asimismo, se presentan ejemplos claros que muestran

paso a paso los procesos y cálculos que se realizan en una Red Neuronal Artificial, con el fin de romper con el mito de la "caja negra", que afirma que no sabemos qué ocurre en las capas ocultas de la red.

Estructura del documento

La estructura del documento es la siguiente: el Capítulo 1 introduce los conceptos teóricos y matemáticos que sustentan las Redes Neuronales Artificiales, incluyendo las funciones de activación, las funciones de pérdida y los algoritmos de optimización; el Capítulo 2 presenta la metodología de análisis de datos y la implementación del modelo computacional con compuertas LSTM; finalmente, el Capítulo 3 expone los resultados obtenidos y las conclusiones del estudio.

Objetivos

Objetivo general

Entrenar una Red Neuronal Artificial con cuentos de Jorge Luis Borges, Mario Benedetti y Julio Cortázar, de manera que sea capaz de predecir la autoría de otros textos de estos autores.

Objetivos particulares

- Recopilar un conjunto de datos sólido para entrenar la Red Neuronal Artificial.
- Preprocesar los textos mediante herramientas computacionales para limpiar los datos.
- Diseñar e implementar un modelo de Red Neuronal Artificial con compuertas LSTM utilizando el lenguaje de programación *Python*.
- Entrenar la red y evaluar su desempeño con datos no etiquetados (textos de los mismos autores no utilizados durante el entrenamiento).

1 Redes Neuronales Artificiales

1.1 La neurona biológica

Las neuronas biológicas son células especializadas del sistema nervioso, responsables del procesamiento y transmisión de información mediante señales electroquímicas. Su estructura y funcionamiento han servido como fuente de inspiración para el desarrollo de las Redes Neuronales Artificiales (RNA), modelos computacionales que buscan imitar capacidades cognitivas como el aprendizaje y la toma autónoma de decisiones. En esta sección exploraremos los principios fundamentales de las neuronas biológicas, desde su morfología hasta sus mecanismos de comunicación y plasticidad sináptica.

Las neuronas biológicas presentan una organización especializada que permite la recepción, integración y transmisión de señales (Saladin, 2012). Sus principales componentes son: a) soma (cuerpo celular): contiene el núcleo y los organelos encargados de la síntesis de proteínas. En su citoplasma se observan los cuerpos de Nissl (agregados de retículo endoplásmico rugoso) y neurofibrillas (filamentos del citoesqueleto); b) dendritas: prolongaciones ramificadas que reciben señales de otras neuronas. Su extensión y complejidad aumentan la capacidad de integración de información; y c) axón: una única prolongación que conduce impulsos nerviosos a largas distancias.

El axón cuenta con cuatro regiones clave: 1) zona de activación (segmento inicial y cresta axónica), donde se generan los potenciales de acción; 2) internódulos, segmentos cubiertos por mielina que aceleran la conducción; 3) nódulos de Ranvier, zonas desmielinizadas que permiten la regeneración de la señal; y 4) arborización terminal, donde se liberan neurotransmisores en los botones sinápticos.

Existen diferentes tipos de neuronas, aunque todas presentan tres propiedades fundamentales que permiten la comunicación con otras células (Saladin, 2012): 1) excitabilidad, es decir, responden a estímulos ambientales; 2) conductividad, o sea, responden a estímulos mediante la producción de señales eléctricas; 3) secreción, cuando la señal eléctrica alcanza el final de una fibra nerviosa, la neurona secreta un neurotransmisor químico que cruza la separación y estimula la célula siguiente.

Según su rol en el procesamiento de información, las neuronas se clasifican en (Saladin,

2012): 1) neuronas sensoriales (aferentes) especializadas en detectar estímulos y transmitir información de estos al Sistema Nervioso Central (SNC); 2) interneuronas (neuronas de asociación) encontradas completamente dentro del SNC y encargadas de procesar, almacenar y recuperar información y tomar decisiones que determinan la forma en la que el cuerpo responde a los estímulos; 3) motoneuronas (eferentes) que envían señales a las células musculares y glandulares.

La comunicación neural tiene como base la electrofisiología: mecanismos celulares para producir potenciales y corrientes eléctricos. Las células vivas están polarizadas. Las neuronas mantienen un potencial de membrana en reposo de aproximadamente $-70mV$, generado por diferencias en la concentración iónica alta en K^+ intracelular y Na^+ extracelular y la acción de la bomba $Na^+/K^+ATPasa$ (Saladin, 2012).

Cuando un estímulo supera el umbral de excitación ($-55mV$), se produce un potencial de acción: 1) despolarización: apertura de canales de Na^+ regulados por voltaje, permitiendo la entrada masiva de iones positivos; 2) repolarización: cierre de canales de Na^+ y apertura de canales de k^+, restaurando la carga negativa interna; 3) período refractario: breve intervalo en el que la neurona no puede generar nuevos potenciales, limitando la frecuencia de disparo.

La velocidad de transmisión depende directamente de la estructura del axón. Para las fibras amielínicas, la conducción continua es más lenta (0.5 a 2 m/s), mientras que para las fibras mielínicas, la conducción saltatoria alcanza hasta los 120 m/s.

La comunicación entre neuronas ocurre en sinapsis, estructuras donde la neurona presináptica libera neurotransmisores que se difunden por la hendidura sináptica y son recibidos por la neurona postsináptica mediante receptores específicos.

A su vez, las neuronas procesan información mediante potenciales excitatorios repetidos desde una misma neurona presináptica o mediante potenciales excitatorios simultáneos desde múltiples fuentes.

La capacidad de aprendizaje en redes neuronales biológicas está basada en Potenciación a Largo Plazo (LTP). No existen neuronas específicas que almacenen un recuerdo o un aprendizaje, no hay células asignadas para la memoria; en cambio, la base física de la memoria es una ruta a través del cerebro a la que se denomina rastro de memoria (engrama) en la que se han formado nuevas sinapsis o se modificaron sinapsis existentes. Las sinapsis no están fijas de por vida, pueden crearse o eliminarse. A la capacidad sináptica de cambiar se le denomina plasticidad sináptica.

Existen diferentes modos de potenciación sináptica que pueden durar de unos cuantos segundos a toda la vida, a estos se les denomina: a) memoria inmediata: representa la capacidad de mantener algo en la mente durante unos cuantos segundos; b) memoria a corto plazo: puede durar de unos segundos a algunas horas pero la información puede perderse con facilidad si existe alguna distracción o deja de repetirse; y c) memoria a largo

plazo: puede durar toda la vida y se clasifica en memoria declarativa y procedimental. La primera es la encargada de retener acontecimientos y hechos, mientras la segunda retiene habilidades motoras.

Las neuronas biológicas operan mediante un sistema altamente organizado de señales eléctricas y químicas, con mecanismos de integración y plasticidad que permiten el aprendizaje adaptativo. Estos principios han inspirado el diseño de las redes neuronales artificiales (RNA), donde unidades computacionales denominadas "neuronas artificiales" emulan ciertas propiedades de las neuronas biológicas a través de una función matemática: suma ponderada de entradas, funciones de activación y ajuste sináptico (aprendizaje). En las siguientes secciones se explorará cómo estos modelos replican las capacidades de procesamiento de las células biológicas.

1.2 Neuronas artificiales

Una red neuronal artificial es una técnica común en el aprendizaje de máquina (*Machine Learning*) que emula los mecanismos de aprendizaje de los organismos biológicos. En los seres vivos, se les denomina neuronas a las células del sistema nervioso. En la sección anterior vimos cómo existe un pequeño espacio entre las dendritas de una neurona y el axón de otra. Este espacio, denominado sinapsis, es el encargado de conectar las neuronas entre sí. La fuerza de conexión entre sinapsis cambia a menudo por estímulos externos. Estos cambios son los que originan el aprendizaje en los organismos vivientes (Aggarwal, 2018b).

Martín del Brío y Sanz Molina (2001) afirman que "la neurona en realidad es un pequeño procesador, sencillo, lento y poco fiable (a diferencia de nuestros potentes microprocesadores), sin embargo, en el cerebro cohabitan unos cien mil millones de neuronas operando en paralelo", añaden además que "por otra parte, las neuronas no deben ser *programadas*, éstas aprenden a partir de las señales que reciben del entorno, y operan siguiendo un esquema también muy diferente de los computadores convencionales". Los mismos autores aseguran que en una red de neuronas no existe una unidad central de procesamiento, sino que las neuronas influyen una en otra, lo que provoca una dinámica compleja de activaciones y desactivaciones. Así, las neuronas son un sistema auto organizado.

Este mecanismo biológico es simulado en las redes neuronales artificiales, que contienen unidades computacionales conocidas como neuronas artificiales. Una neurona artificial intenta emular, mediante una función matemática, el comportamiento de la neurona biológica. En este caso, en lugar de dendritas tendremos un conjunto de datos numéricos, en lugar de soma tendremos una función matemática que procesará dichos datos, y en lugar de axón contaremos con los denominados pesos sinápticos y el resultado de la función

matemática. De manera análoga a las neuronas biológicas, las neuronas artificiales se interconectan entre sí para formar una red neuronal. Así, la salida de una neurona artificial, a su vez, se convierte en la entrada de otra neurona contigua.

Martín del Brío y Sanz Molina (2001) definen a la neurona como: "un dispositivo simple de cálculo que, a partir de un vector de entrada procedente del exterior o de otras neuronas, proporciona una única respuesta o salida"; por otra parte, Silaparasetty (2020) la define como "una función matemática que replica las neuronas en el cerebro humano".

Una neurona artificial se compone básicamente de un vector de entradas, un vector de pesos sinápticos (a cada entrada le corresponderá un peso), un sesgo, una función de activación y una salida. A grandes rasgos, una red neuronal artificial es un conjunto de neuronas artificiales conectadas entre sí. Tomando en cuenta lo anterior, proponemos la siguiente definición:

Definición 1.2.1. Sean $\phi : \mathbb{R} \to \mathbb{R}$ una función y $W \in \mathbb{R}^n$, se define la neurona artificial sin sesgo asociada a ϕ y a W, denotada por $\hat{y}_{\phi,W}$, donde

$$\hat{y}_{\phi,W} : \mathbb{R}^n \to \mathbb{R}$$

como

$$\hat{y}_{\phi,W}(X) = \phi(W \cdot X)$$

El vector $X = [x_1, x_2, ..., x_n] \in \mathbb{R}^n$ representa las entradas de la neurona. Por ejemplo, supongamos que queremos que la neurona clasifique el nivel socioeconómico de una persona basado en los siguientes datos: 1) material del piso de la vivienda; 2) cantidad de focos en la vivienda; 3) número de habitantes en la vivienda; y 4) número de vehículos de la familia. En ese sentido, nuestro vector de entradas $X \in \mathbb{R}^4$ sería de la forma $X = [x_1, x_2, x_3, x_4]$ donde x_1 representa el material del piso, x_2 la cantidad de focos y así sucesivamente. Notemos que en este ejemplo particular, x_2, x_3 y x_4 son datos numéricos enteros, pero x_1 no lo es, así que es necesario establecer una equivalencia numérica para cada material considerado en el estudio (como 0 para tierra, 1 para cemento, 2 para loseta, etcétera). Así, si tenemos un vector $Y = [y_1, y_2, y_3] \in \mathbb{R}^3$ porque nos falta algún dato, o un vector $Z = [z_1, z_2, z_3, z_4, z_5] \in \mathbb{R}^5$ porque también se consideró el número de ventanas de la vivienda como parámetro, entonces Y y Z no podrán utilizarse en la neurona, porque esta neurona está definida sobre $\mathbb{R}^4$.

$W = [w_1, w_2, ..., w_n] \in \mathbb{R}^n$ es el vector de pesos sinápticos. El valor de w_i cuantifica la importancia de la entrada x_i, o, en otras palabras, representa la fuerza de la conexión entre la entrada x_i y la neurona. Más adelante veremos que los valores w_i se ajustan para establecer relaciones entre las entradas y la salida de la neurona.

La función $\phi : \mathbb{R} \to \mathbb{R}$ se conoce como función de activación. Ésta nos permite controlar la salida de la neurona dentro de ciertos rangos adecuados según la necesidad del problema. En algunos modelos de Redes Neuronales Artificiales, se pide que la función de activación sea monótona creciente y continua. Existen diferentes funciones de activación, aunque las más comunes, según Martín del Brío y Sanz Molina (2001) son: la identidad ($\phi(z) = z$), las funciones escalón (como $\phi(z) = \text{sgn}(z)$), o las sigmoides ($\phi(z) = \sigma(z) = \dfrac{1}{1+e^{-z}}$, $\phi(z) = \tanh(z) = \dfrac{e^z - e^{-z}}{e^z + e^{-z}}$). La función de activación determina el valor de salida de la neurona. Dependiendo de la función elegida, la salida de la neurona puede quedar restringida a ciertos intervalos, por ejemplo, la tangente hiperbólica produce valores en el intervalo $(-1, 1)$ o la función sigmoide en el intervalo $(0, 1)$. En algunas ocasiones, los algoritmos requieren que la función de activación sea derivable.

Finalmente, el operador $W \cdot X$ denota el producto interno euclidiano en $\mathbb{R}^n$. Bibliografía como (Aggarwal, 2018b) suelen denotar dicho producto interno como $W^T X$. También suelen representar dicho producto interno como:

$$z = W \cdot X = \sum_{i=1}^{n} w_i x_i$$

Esta última representación resulta conveniente porque permite mostrar claramente cómo se obtiene el valor de salida de una neurona como en el siguiente ejemplo:

Ejemplo 1.2.1. Sean $X = [5, 15, 4, 3] \in \mathbb{R}^4$ el vector de entradas, donde x_1 representa el material del piso, x_2 el número de focos, x_3 la cantidad de cohabitantes y x_4 la cantidad de autos; $W = [0.4, 0.8, 0.3, -0.2]$ el vector de pesos sinápticos; y $\phi(z) = z$ la función de activación.

Entonces, la neurona artificial sin sesgo $\hat{y}_{\phi,W}(X) = \phi(W \cdot X)$ es:

$$\begin{aligned}
\hat{y}_{\phi,W}(X) &= \phi(W \cdot X) \\
&= \phi(\sum_{i=1}^{4} w_i x_i) \\
&= \phi(5 \times 0.4 + 15 \times 0.8 + 4 \times 0.3 + 3 \times -0.2) \\
&= \phi(2 + 12 + 1.2 - 0.6) \\
&= \phi(14.6) \\
&= 14.6
\end{aligned}$$

También es importante mencionar que la bibliografía habla de un sesgo b, que determinará si la neurona se activa o no.

Definición 1.2.2. Sean $\phi : \mathbb{R} \to \mathbb{R}$ una función, $W \in \mathbb{R}^n$ y $b \in \mathbb{R}$, se define la neurona artificial con sesgo explícito b asociada a ϕ, W y b, denotada por $\hat{y}_{\phi,W,b}$, donde

$$\hat{y}_{\phi,W,b} : \mathbb{R}^n \to \mathbb{R}$$

como

$$\hat{y}_{\phi,W,b}(X) = \phi(W \cdot X + b)$$

Algunas veces se considera el sesgo como un valor w_i dentro del vector de pesos al que se le asigna una entrada $x_i \equiv 1$, y normalmente se considera $i = 0$ para indicar que el sesgo será siempre la primera entrada, o $i = n$ para que sea la última entrada.

Así tenemos la siguiente:

Definición 1.2.3. Sean $\phi : \mathbb{R} \to \mathbb{R}$ una función, $\tilde{X} = [1, x_1, x_2, ..., x_n] \in \mathbb{R}^{n+1}$ y $\tilde{W} = [b, w_1, w_2, ..., w_n] \in \mathbb{R}^{n+1}$. Se define la neurona artificial con sesgo implícito b asociada a ϕ y $\tilde{W}$, denotada por $\hat{y}_{\phi,\tilde{W}}$, donde

$$\hat{y}_{\phi,\tilde{W}} : \mathbb{R}^{n+1} \to \mathbb{R}$$

como

$$\hat{y}_{\phi,\tilde{W}}(\tilde{X}) = \phi(\tilde{W} \cdot \tilde{X})$$

Con base en las dos definiciones anteriores, notemos que:

$$\begin{aligned}
\tilde{W} \cdot \tilde{X} &= \sum_{i=0}^{n} w_i x_i \\
&= w_0 x_0 + w_1 x_1 + \cdots + w_n x_n \\
&= b \times 1 + w_1 x_1 + \cdots + w_n x_n \\
&= b + \sum_{i=1}^{n} w_i x_i \\
&= \sum_{i=1}^{n} w_i x_i + b \\
&= X \cdot W + b
\end{aligned}$$

Con esto podemos observar que las definiciones de neurona artificial con sesgo explícito y neurona artificial con sesgo implícito son equivalentes.

Veamos ahora un pequeño ejemplo de cómo influye el sesgo b en la salida de la neurona $\hat{y}_{\phi,W}(X)$:

Ejemplo 1.2.2. Sean

$$X = [5, 15, 4, 3] \in \mathbb{R}^4$$

el vector de entradas;

$$W = [0.4, 0.8, 0.3, -0.2] \in \mathbb{R}^4$$

el vector de pesos sinápticos;

$$b = -15$$

el sesgo; y

$$\}\phi(z) = z$$

la función de activación.

Entonces, la neurona artificial con sesgo explícito b, $\hat{y}_{\phi,W,b}(X) = \phi(W \cdot X + b)$ es:

$$\begin{aligned}
\hat{y}_{\phi,W}(X) &= \phi(W \cdot X + b) \\
&= \phi(\sum_{i=1}^{4} w_i x_i + b) \\
&= \phi(5 \times 0.4 + 15 \times 0.8 + 4 \times 0.3 + 3 \times -0.2 - 15) \\
&= \phi(2 + 12 + 1.2 - 0.6 - 15) \\
&= \phi(14.6 - 15) \\
&= \phi(-0.4) \\
&= -0.4
\end{aligned}$$

Para efectos de este trabajo, de ahora en adelante vamos a denotar la neurona artificial como $\hat{y}(X)$ a sabiendas de que es una función parametrizada por el vector de pesos sinápticos W, el sesgo b (cuando exista) y por la función de activación ϕ.

En las implementaciones computacionales, los valores de los pesos y del sesgo se inician comúnmente de manera aleatoria y se van ajustando durante el entrenamiento de la red neuronal; pero, al concluir el entrenamiento, estos valores quedan fijos y serán éstos los que operen cuando se le dé a la red neuronal un nuevo vector de entrada para predecir o catalogar.

Antes de continuar, es necesario establecer las siguientes definiciones:

Definición 1.2.4. Se define una Red Neuronal Artificial Monocapa (*single-layer artificial neural network*) como una estructura compuesta por un vector $X \in \mathbb{R}^n$ y una colección de neuronas artificiales $O = [\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m]$.

Denotaremos dicha red como

$$M_\Theta(X) = [\hat{y}_1(X), \hat{y}_2(X), \ldots, \hat{y}_m(X)],$$

donde Θ representa el conjunto de parámetros compuesto por los pesos sinápticos y los sesgos de las neuronas $\hat{y}_i \in O$.

El vector $X \in \mathbb{R}^n$ se denomina capa de entrada, mientras que O se conoce como capa de salida.

En la literatura suelen utilizar la letra o para referirse a las neuronas de la capa de salida, debido a que en inglés se denomina *output layer*. Por ello, utilizaremos la letra O para denotar la capa de salida, y o_i nos ayudará a distinguir una neurona artificial que se encuentra específicamente en la capa de salida O. Así, representaremos la capa de salida como $O = [o_1, o_2, \ldots, o_m]$ donde $o_i := \hat{y}_i$ es una neurona artificial.

Definición 1.2.5. Se define una Red Neuronal Artificial Multicapa (*multilayer artificial neural network*) como una estructura compuesta por:

- una capa de entrada $X \in \mathbb{R}^n$;
- una familia de capas $\{H_i\}_{i=1}^p$, donde cada capa tiene la forma

  $$H_i = [h_{i1}, h_{i2}, \ldots, h_{iq_i}],$$

  y cada h_{ij} es neurona artificial;
- una capa de salida

  $$O = [o_1, o_2, \ldots, o_m].$$

Definimos recursivamente los vectores de activación por:

$$A_0 = X,$$

y

$$A_i = H_i(A_{i-1}), \qquad \text{con } i = \overline{1, p}.$$

A cada colección H_i se le denomina capa oculta y corresponde con una capa intermedia que no es el vector de entradas, ni la capa de salida.

Finalmente, la salida de la red está dada por

$$N_\Theta(X) = O(A_p),$$

donde Θ representa el conjunto de parámetros compuesto por los pesos sinápticos y sesgos de todas las neuronas de la red.

Suele utilizarse la letra h para referirse a las neuronas de las capas ocultas por el término inglés *hidden layer*. En adelante usaremos H_i para denotar la i-ésima capa oculta.

A manera de ejemplo simple, veremos cómo opera una Red Neuronal Artificial compuesta por dos neuronas en la capa oculta y una neurona en la capa de salida.

Ejemplo 1.2.3. Tomemos como base la idea anterior: una red neuronal busca catalogar el nivel socioeconómico de un individuo basado en cuatro características de la vivienda, y demos algunos valores numéricos únicamente para ver el funcionamiento de la neurona. Sólo tengamos en cuenta que, como estos valores no son tomados de datos reales, la salida de la neurona no nos dará ninguna información relevante.

Sean

$$X = [3, 7, 4, 2]$$

donde x_1 representa el material del piso de la vivienda, x_2 el número de focos, x_3 el número de habitantes y x_4 los vehículos que tiene la familia;

$$W_{h1} = [0.3, 2.5, 1.7, 0.8] \text{ y } W_{h2} = [0.4, 0.8, 0.3, -0.2]$$

los pesos sinápticos correspondientes a las neuronas h_1 y h_2 de la capa oculta;

$$b_{h1} = -3, \quad b_{h2} = -5 \text{ y } b_{o1} = -0.3$$

los sesgos correspondientes a las neuronas de la capa oculta y a la neurona de la capa de salida respectivamente;

$$\phi_{h1}(z) = z \text{ y } \phi_{h2}(z) = \operatorname{sgn}(z)$$

las funciones de activación correspondientes a las neuronas de la capa oculta;

$$W_{o1} = [0.1, -0.5] \text{ y } \phi_{o1}(z) = \sigma(z)$$

el vector de pesos sinápticos y la función de activación de la capa de salida que sólo cuenta con la neurona $O = [o_1]$.

Veamos, en primera instancia, lo que sucede en h_1:

$$\begin{aligned} h_1(X) &= \phi_{h1}(W_{h1} \cdot X + b_{h1}) \\ &= \phi_{h1}(\sum_{i=1}^{4} w_{h1,i} x_i + b_{h1}) \\ &= \phi_{h1}(0.3 \times 3 + 2.5 \times 7 + 1.7 \times 4 + 0.8 \times 2 - 3) \\ &= \phi_{h1}(0.9 + 17.5 + 6.8 + 1.6 - 3) \\ &= \phi_{h1}(23.8) \\ &= 23.8 \end{aligned}$$

Ahora veamos lo que sucede con $\hat{y}_{h2}$:

$$\begin{aligned} h_2(X) &= \phi_{h2}(W_{h2} \cdot X + b_{h2}) \\ &= \phi_{h2}(\sum_{i=1}^{4} w_{h2,i} x_i + b_{h2}) \\ &= \phi_{h2}(0.4 \times 3 + 0.8 \times 7 + 0.3 \times 4 - 0.2 \times 2 - 5) \\ &= \phi_{h2}(1.2 + 5.6 + 1.2 - 0.4 - 5) \\ &= \phi_{h2}(2.6) \\ &= 1 \end{aligned}$$

Obtenemos $H(X) = A_1 = [h_1(X), h_2(X)] = [23.8, 1]$ que será la entrada para nuestra capa de salida $O = o_1$. Así:

$$
\begin{aligned}
o_1(A_1) &= \phi_{o1}(W_{o1} \cdot A_1 + b_{o1}) \\
&= \phi_{o1}(\sum_{i=1}^{2} w_{o1,i} h_i + b_{o1}) \\
&= \phi_{o1}(0.1 \times 23.8 - 0.5 \times 1 - 0.3) \\
&= \phi_{o1}(2.38 - 0.5 - 0.3) \\
&= \phi_{o1}(1.58) \\
&= \frac{1}{1 + e^{-1.58}} \\
&= 0.829204
\end{aligned}
$$

El valor de salida de nuestra pequeña red neuronal es 0.829204.

De ahora en adelante, utilizaremos el término Red Neuronal Artificial para referirnos tanto a la red monocapa como a la red multicapa, y las distinguiremos por la cantidad de capas ocultas que tenga el modelo.

En las neuronas artificiales, el aprendizaje no se originará a través de los cambios en la fuerza de conexión de la sinapsis (como en las neuronas biológicas), sino a partir de los datos de entrenamiento.

Definición 1.2.6. Sea $\Theta_t = (W_t, b_t)$ el conjunto de parámetros de una Red Neuronal Artificial en la iteración t.

Se define el entrenamiento como el proceso de construcción de una sucesión de redes neuronales $\{N_{\Theta_t}\}_{t=0}^{T}$, donde cada red neuronal está determinada por los parámetros Θ_t, obtenidos iterativamente con el propósito de minimizar una función $J(\Theta)$ a la que se llamará función objetivo y que representa el error de predicción de la Red Neuronal Artificial.

Notemos que, la definición de entrenamiento de una Red Neuronal Artificial se ajusta a la definición de una neurona como $\hat{y}_{\phi,W}(X) = \phi(W \cdot X)$ y que por lo tanto

$$W_1 \neq W_2 \Longrightarrow \hat{y}_{\phi,W_1} \neq \hat{y}_{\phi,W_2}$$

Sin embargo, en la bibliografía que define una Red Neuronal Artificial desde un punto de vista computacional, suelen describir el proceso de entrenamiento como un ajuste de parámetros más que como una sucesión de redes neuronales.

Durante la etapa de entrenamiento, se le proveen a la red neuronal artificial una serie de datos de entrada cuyo valor de salida se conoce previamente; estos datos se denominan

datos etiquetados y sirven para determinar qué tan adecuados son los valores de los pesos sinápticos dependiendo de qué tan cercana sea la salida de la red con respecto al valor real. Si el valor real dista mucho del valor de salida, entonces se ajustarán los pesos para que esa distancia sea mínima. Así, los pesos de las neuronas se ajustan en respuesta a los errores de predicción. El objetivo de cambiar los pesos es modificar la función calculada para hacer predicciones cada vez más precisas.

Al ajustar sucesivamente los pesos de las neuronas artificiales a partir de muchos datos etiquetados, la función calculada por la red neuronal se refina con el tiempo y puede hacer predicciones más precisas, hasta que, eventualmente, es capaz de reconocer datos no etiquetados. La principal utilidad de los modelos de aprendizaje automático se obtiene de su capacidad para generalizar su aprendizaje de los datos etiquetados durante su entrenamiento a datos sin etiquetar posteriormente (Aggarwal, 2018b).

Antes de estudiar a fondo cómo opera una red neuronal artificial, revisaremos la estructura y el funcionamiento de una neurona artificial básica: el perceptrón.

1.2.1 El perceptrón

El primer modelo de Red Neuronal Artificial (RNA) fue desarrollado en 1958 por Rosenblatt (Hilera González & Martínez Hernando, 1995), aunque este primer modelo no pretendía ser un algoritmo computacional, sino una maquinaria compuesta de varios circuitos que semejaban a las neuronas biológicas (Aggarwal, 2018b). Posteriormente, se implementó el algoritmo computacional que sigue estas mismas ideas. En un inicio, la función de activación usada por el perceptrón fue la función signo debido a que esta arquitectura de red neuronal se utiliza principalmente para catalogar elementos en dos categorías.

El perceptrón simple está compuesto por una capa de entrada y una capa de salida que contiene una única neurona artificial. Por otro lado, el perceptrón multicapa incorpora una o más capas ocultas entre la capa de entrada y la capa de salida.

Consideremos el caso en el que cada dato para el entrenamiento es de la forma (X_i, y_i), donde cada $X_i = [x_{i1}, \ldots, x_{in}]$ contiene n características variables, y $y_i \in \{-1, 1\}$ contiene la variable de clase binaria del valor etiquetado (Aggarwal, 2018b). Tomemos el ejemplo anterior, pero imaginemos que la clasificación sólo puede tomar dos valores: $y_i = -1$, que representa una clase social baja; y $y_i = 1$, que representa una clase social alta.

La capa de entrada consta de $n = 4$ nodos que contienen las cuatro características del vector X_i, relacionadas con cuatro pesos del vector W. En la capa de entrada no se realiza ningún cálculo. La función

$$X_i \mapsto W \cdot X_i$$

define una función sobre $\mathbb{R}^n$, cuyo valor se calcula en el nodo de salida. Finalmente, se utiliza la función de activación $\phi(z) = \text{sgn}(z)$ para hacer la predicción. Así, la predicción $\hat{y}_i(X_i)$ se calcula como:

$$\hat{y}_i(X_i) = \text{sgn}(W \cdot X_i + b).$$

La función signo mapea un valor real hacia -1 o 1, lo que es apropiado para hacer una clasificación binaria. Nótese que $\hat{y}_i(X_i)$ denota un valor predicho, mientras que y_i se utiliza para denotar un valor observado o etiquetado. El error de predicción será

$$E(X_i) = y_i - \hat{y}_i(X_i).$$

En este caso particular, $E(X_i) \in \{-2, 0, 2\}$ debido a que tanto y_i como $\hat{y}_i(X)$ sólo pueden tomar los valores $\{-1, 1\}$. Si $E(X_i) \neq 0$, los valores de W deberán actualizarse.

La función signo sirve como función de activación en un perceptrón simple. Aunque, como veremos más adelante, existen diferentes funciones de activación que pueden utilizarse dependiendo de qué se busque modelar. Cabe destacar que, como el perceptrón simple sólo tiene un nodo de salida, que es donde se realizan los cálculos, se considera que es una Red Neuronal Artificial Monocapa.

El objetivo de una Red Neuronal Artificial consiste en minimizar el error de predicción (Aggarwal, 2018b). Para ello, es necesario distinguir entre función de pérdida y función objetivo. Sea

$$\mathcal{D} = \{(X_i, y_i)\}_{i=1}^n$$

un conjunto de entrenamiento, donde X_i representa el vector de entrada correspondiente a la observación i, y y_i el valor observado asociado a dicha observación.

La función de pérdida, sobre la que profundizaremos más adelante, cuantifica el error asociado a una observación particular del conjunto de entrenamiento. Una elección común consiste en utilizar el error cuadrático:

$$L_i = (y_i - \hat{y}_i(X_i))^2.$$

La expresión anterior corresponde al error cuadrático entre el valor observado y_i y la predicción $\hat{y}_i(X)$.

Por otro lado, la función objetivo $J(\Theta)$, que también abordaremos en la siguiente sección, representa el error de todo el conjunto de entrenamiento. Una elección común consiste en utilizar el promedio de las pérdidas:

$$J(\Theta) = \frac{1}{n}\sum_{i=1}^{n} L_i = \frac{1}{n}\sum_{i=1}^{n}(y_i - \text{sgn}(X_i \cdot W + b))^2.$$

En este sentido, la función de pérdida puede interpretarse como una medida local del error, mientras que la función objetivo corresponde a una medida global sobre el conjunto de entrenamiento.

Una vez aclarado lo anterior, es importante mencionar que, para minimizar la función objetivo, suele utilizarse una función de pérdida diferenciable. El error cuadrático es ampliamente utilizado debido a que produce funciones suaves y diferenciables, lo que facilita el uso de métodos de optimización basados en gradientes. Además, el error cuadrático penaliza con mayor severidad los errores grandes, haciendo que el entrenamiento modifique los parámetros de manera más significativa cuando las predicciones se alejan considerablemente de los valores observados.

En términos matemáticos, el entrenamiento de una red neuronal puede formularse como el problema de optimización:

$$\min_{\Theta} J(\Theta)$$

, donde Θ representa el conjunto de parámetros de la red neuronal.

Sin embargo, en el caso particular del perceptrón clásico, la función de activación $\phi(z) = \text{sgn}(z)$ no es diferenciable en $z = 0$. Por ello, la regla de actualización del perceptrón no puede obtenerse directamente mediante derivación ordinaria de una función objetivo diferenciable. A pesar de ello, la regla iterativa de actualización del perceptrón comparte similitudes conceptuales con los métodos de descenso de gradiente y descenso estocástico del gradiente, que se estudiarán más adelante para redes neuronales con funciones de activación diferenciables.

Durante el entrenamiento, cuando el punto de datos X_i se introduce en la red, el vector de pesos W se actualiza de la siguiente manera (Aggarwal, 2018b):

$$W_{\text{nuevo}} \leftarrow\!\shortmid W_{\text{actual}} + \alpha(y_i - \hat{y}_i(X_i))X_i.$$

El parámetro α es conocido como tasa de aprendizaje (*learning rate*). Si se incorpora un sesgo explícito, éste puede actualizarse de manera análoga:

$$b_{\text{nuevo}} \leftarrow\!\shortmid b_{\text{actual}} + \alpha(y_i - \hat{y}_i(X_i)).$$

El algoritmo del perceptrón recorre de manera cíclica todos los datos de entrenamiento y, de manera iterativa, ajusta los pesos hasta alcanzar la convergencia. Cada dato del conjunto de entrenamiento puede recorrerse varias veces; a dichos recorridos se les conoce como *épocas*. De la ecuación anterior se puede observar que el vector de pesos se actualizará únicamente en los puntos donde $y_i \neq \hat{y}_i(X_i)$, lo que ocurre sólo cuando se ha cometido algún error en la predicción.

El algoritmo del perceptrón tiene un buen desempeño cuando se trata de clasificar con-

juntos de datos que son linealmente separables, pero no logrará buenos resultados cuando los datos no son linealmente separables. Sin embargo, es importante entender su funcionamiento, ya que este algoritmo funge como base para otros algoritmos de aprendizaje profundo.

Veamos un ejemplo detallado del funcionamiento del perceptrón. Para ello, retomemos el ejemplo 1.2.2. Recordemos que la idea es clasificar el nivel socioeconómico de un individuo basado en cuatro características: el material del suelo de la vivienda, el número de focos, el número de habitantes y la cantidad de vehículos. Consideremos esta vez que sólo tenemos dos niveles socioeconómicos: el bajo y el alto. Esto último con la finalidad de desarrollar un ejemplo de algoritmo de perceptrón clásico, donde utilizaremos la función signo.

Supongamos que tenemos los siguientes datos:

Dato	Material del suelo	Focos	Habitantes	Vehículos	Nivel socioeconómico
1	1	5	5	1	Bajo
2	5	15	4	4	Alto
3	0	4	6	0	Bajo
4	4	18	3	2	Alto

Cuadro 1.1: Tabla de datos para el ejemplo

Ejemplo 1.2.4. Trabajaremos con los datos de la tabla 1.1.

Primero vamos a determinar que el nivel socioeconómico bajo reciba la etiqueta $y_i = -1$ mientras que el nivel socioeconómico alto reciba la etiqueta $y_i = 1$. De esta manera vamos a obtener cuatro elementos de entrada X_i y los vamos a acomodar junto con sus valores de salida y_i con $i = \overline{1,4}$.

Sean

$$X_1 = [1,5,5,1] \text{con su valor observado } y_1 = -1$$
$$X_2 = [5,15,4,4] \text{con su valor observado } y_2 = 1$$
$$X_3 = [0,4,6,0] \text{con su valor observado } y_3 = -1$$
$$X_4 = [4,18,3,2] \text{con su valor observado } y_4 = 1$$

Sean, además, los parámetros del perceptrón $W^{(0)} = [1,2,2,1]$, la tasa de aprendizaje $\alpha = 0.5$ y la función de activación $\phi(z) = \text{sgn}(z)$. Para este ejemplo utilizaremos una neurona artificial sin sesgo.

Sabemos que los ciclos realizados en cada época pueden hacerse en orden aleatorio, pero para efectos del ejemplo haremos el ciclo siguiendo un orden ascendente y

actualizaremos el vector de pesos en caso de que el error $E(X_i) \neq 0$.

Primera época

Tomamos el primer dato (X_1, y_1) y calculamos el valor de salida $\hat{y}_1^{(0)}(X_1)$

$$\begin{aligned}\hat{y}_1^{(0)}(X_1) &= \text{sgn}(X_1 \cdot W^{(0)}) \\ &= \text{sgn}([1,5,5,1] \cdot [1,2,2,1]) \\ &= \text{sgn}(1+10+10+1) \\ &= \text{sgn}(22) \\ &= 1\end{aligned}$$

Comparamos el valor observado y_1 con el valor predicho $\hat{y}_1^{(0)}(X_1)$ para ver si hay error:

$$E(X_1) = y_1 - \hat{y}_1^{(0)}(X_1) = -1 - 1 = -2 \neq 0$$

Como $E(X_1) \neq 0$, entonces debemos actualizar los parámetros:

$$\begin{aligned}W^{(1)} &\leftarrow\!\shortmid W^{(0)} + \alpha(y_1 - \hat{y}_1^{(0)}(X_1)) \cdot X_1 \\ &\leftarrow\!\shortmid [1,2,2,1] + 0.5(-1-1) \cdot [1,5,5,1] \\ &\leftarrow\!\shortmid [1,2,2,1] + 0.5(-2) \cdot [1,5,5,1] \\ &\leftarrow\!\shortmid [1,2,2,1] + -1 \cdot [1,5,5,1] \\ &\leftarrow\!\shortmid [1,2,2,1] + [-1,-5,-5,-1] \\ &\leftarrow\!\shortmid [0,-3,-3,0]\end{aligned}$$

Así, nuestros nuevos parámetros son $W^{(1)} = [0,-3,-3,0]$.

Ahora, con los parámetros actualizados, repetimos el proceso con el segundo dato (X_2, y_2).

$$\begin{aligned}\hat{y}_2^{(0)}(X_2) &= \text{sgn}(X_2 \cdot W^{(1)}) \\ &= \text{sgn}([5,15,4,4] \cdot [0,-3,-3,0]) \\ &= \text{sgn}(0-45-12+0) \\ &= \text{sgn}(-57) \\ &= -1\end{aligned}$$

Comparamos el valor observado y_2 con el valor predicho $\hat{y}_2^{(0)}(X_2)$ para ver si hay error:

$$E(X_2) = y_2 - \hat{y}_2^{(0)}(X_2) = 1 - (-1) = 2 \neq 0$$

Como $E(X_2) \neq 0$, entonces debemos actualizar los parámetros:

$$\begin{aligned} W^{(2)} &\leftarrow\!\shortmid W^{(1)} + \alpha(y_2 - \hat{y}_2^{(0)}(X_2)) \cdot X_2 \\ &\leftarrow\!\shortmid [0,-3,-3,0] + 0.5(1-(-1)) \cdot [5,15,4,4] \\ &\leftarrow\!\shortmid [0,-3,-3,0] + 0.5(2) \cdot [5,15,4,4] \\ &\leftarrow\!\shortmid [0,-3,-3,0] + 1 \cdot [5,15,4,4] \\ &\leftarrow\!\shortmid [0,-3,-3,0] + [5,15,4,4] \\ &\leftarrow\!\shortmid [5,12,1,4] \end{aligned}$$

Así, nuestros nuevos parámetros son $W^{(2)} = [5,12,1,4]$.
Continuamos con el siguiente dato (X_3, y_3):

$$\begin{aligned} \hat{y}_3^{(0)}(X_3) &= \text{sgn}(X_3 \cdot W^{(2)}) \\ &= \text{sgn}([0,4,6,0] \cdot [5,12,1,4]) \\ &= \text{sgn}(0+48+6+0) \\ &= \text{sgn}(54) \\ &= 1 \end{aligned}$$

Comparamos el valor observado y_3 con el valor predicho $\hat{y}_3^{(0)}(X_3)$ para ver si hay error:

$$E(X_3) = y_3 - \hat{y}_3^{(0)}(X_3) = -1 - 1 = -2 \neq 0$$

Nuevamente actualizamos los parámetros:

$$\begin{aligned} W^{(3)} &\leftarrow\!\rightarrow W^{(2)} + \alpha(y_3 - \hat{y}_3^{(0)}(X_3)) \cdot X_3 \\ &\leftarrow\!\rightarrow [5, 12, 1, 4] + 0.5(-1 - 1) \cdot [0, 4, 6, 0] \\ &\leftarrow\!\rightarrow [5, 12, 1, 4] + 0.5(-2) \cdot [0, 4, 6, 0] \\ &\leftarrow\!\rightarrow [5, 12, 1, 4] - 1 \cdot [0, 4, 6, 0] \\ &\leftarrow\!\rightarrow [5, 12, 1, 4] + [0, -4, -6, 0] \\ &\leftarrow\!\rightarrow [5, 8, -5, 4] \end{aligned}$$

Los parámetros se actualizan a $W^{(3)} = [5, 8, -5, 4]$.

Repetimos los pasos para el dato (X_4, y_4):

$$\begin{aligned} \hat{y}_4^{(0)}(X_4) &= \text{sgn}(X_4 \cdot W^{(3)}) \\ &= \text{sgn}([4, 18, 3, 2] \cdot [5, 8, -5, 4]) \\ &= \text{sgn}(20 + 144 - 15 + 8) \\ &= \text{sgn}(157) \\ &= 1 \end{aligned}$$

Comparamos el valor observado y_4 con el valor predicho $\hat{y}_4^{(0)}(X_4)$ para ver si hay error:

$$E(X_4) = y_4 - \hat{y}_4^{(0)}(X_4) = 1 - 1 = 0$$

En este caso, el error es cero, así que los parámetros se mantienen como $W^{(3)} = [5, 8, -5, 4]$.

Notemos que ya hemos recorrido todos los valores de nuestro conjunto de datos. Por lo tanto, aquí termina la primera época. Sin embargo, el algoritmo aún no termina, pues no sabemos si los parámetros $W^{(3)}$ ajustan todos los datos, por el momento sabemos que ajustan el cuarto valor (X_4, y_4).

Segunda época

Como el proceso es iterativo, vamos a obviar algunos pasos.

Tomamos el primer dato, calculamos el valor predicho y comparamos para ver si existe error.

$$\begin{aligned}\hat{y}_1^{(1)}(X_1) &= \text{sgn}([1,5,5,1]\cdot[5,8,-5,4]) \\ &= \text{sgn}(24) \\ &= 1\end{aligned}$$

Como $y_1 = -1$ y $\hat{y}_1^{(1)}(X_1) = 1$ existe un error y hay que actualizar los parámetros $W^{(3)}$

$$\begin{aligned}W^{(4)} &\leftarrow\!\shortmid W^{(3)} + \alpha(y_1 - \hat{y}_1^{(1)}(X_1))\cdot X_1 \\ &\leftarrow\!\shortmid [5,8,-5,4] + 0.5(-1-1)\cdot[1,5,5,1] \\ &\leftarrow\!\shortmid [5,8,-5,4] + [-1,-5,-5,-1] \\ &\leftarrow\!\shortmid [4,3,-10,3]\end{aligned}$$

Ahora, tomamos el dato (X_2, y_2) y verificamos el vector $W^{(4)} = [4,3,-10,3]$:

$$\begin{aligned}\hat{y}_2^{(1)}(X_2) &= \text{sgn}([5,15,4,4]\cdot[4,3,-10,3]) \\ &= \text{sgn}(37) \\ &= 1\end{aligned}$$

En este caso, el valor predicho $\hat{y}_2^{(1)}(X_2) = 1$ coincide con el valor observado $y_2 = 1$, por lo tanto, no hay que ajustar los parámetros.

Verificamos para el dato (X_3, y_3):

$$\begin{aligned}\hat{y}_3^{(1)}(X_3) &= \text{sgn}([0,4,6,0]\cdot[4,3,-10,3]) \\ &= \text{sgn}(-48) \\ &= -1\end{aligned}$$

Una vez más, el valor predicho $\hat{y}_3^{(1)}(X_3) = -1$ coincide con el valor observado $y_3 = -1$, por lo tanto, los parámetros se mantienen.

Verificamos para el dato (X_4, y_4):

$$\begin{aligned}\hat{y}_4^{(1)}(X_4) &= \text{sgn}([4,18,3,2]\cdot[4,3,-10,3]) \\ &= \text{sgn}(42) \\ &= 1\end{aligned}$$

Finalmente, el valor predicho $\hat{y}_4^{(1)}(X_4) = 1$ coincide con el valor observado $y_4 = 1$, por lo tanto, no hay que ajustar los parámetros.

Aquí termina la segunda época, pero el algoritmo aún no se detiene, pues en esta época los parámetros $W^{(4)}$ se actualizaron una vez. Así que es necesario iniciar otra época.

Tercera época

Tomaremos el primer dato (X_1, y_1) y calculamos el valor predicho $\hat{y}_1^{(2)}(X_1)$

$$\begin{aligned}\hat{y}_1^{(2)}(X_1) &= \text{sgn}([1,5,5,1]\cdot[4,3,-10,3]) \\ &= \text{sgn}(-28) \\ &= -1\end{aligned}$$

Con esto verificamos que el valor observado $y_1 = -1$ coincide con el valor predicho $\hat{y}_1^{(2)}(X_1) = -1$. Así que los parámetros no se actualizan.

Es necesario verificar para los otros tres datos restantes, pero como

$$W^{(4)} = [4, 3, -10, 3]$$

no se actualizó, conocemos los resultados por la segunda época, así que obviaremos los cálculos. Simplemente, concluiremos que en la tercera época ya no hubo ninguna actualización y, por lo tanto, el algoritmo se detiene.

Con los parámetros resultantes se alcanza la convergencia y los datos se ajustan adecuadamente.

Si los datos son linealmente separables, el algoritmo se detendrá en alguna época, pero si no lo son, los parámetros seguirán actualizándose y el algoritmo puede no converger.

Este ejemplo pretende ilustrar cómo funciona un perceptrón simple. Pero hay que tomar en cuenta que los datos tomados no son datos reales, y que la convergencia se alcanzó con cierta facilidad debido a que son pocos datos. En un modelo real se trabaja con conjuntos de datos mucho mayores y la convergencia puede alcanzarse en cientos o miles de épocas o, incluso, podría no alcanzarse.

1.2.2 Función objetivo y Función de pérdida

En la sección anterior, describimos el algoritmo del perceptrón y su objetivo: minimizar el error ajustando los parámetros Θ (pesos sinápticos y sesgos). En esta sección profundizaremos en los conceptos de *función objetivo* y *función de pérdida* ya que éstas son las responsables del "proceso de aprendizaje" de la Red Neuronal Artificial. El propósito fun-

damental del entrenamiento de una Red Neuronal es hallar los parámetros adecuados que generen una red $N_\Theta(X_i)$ que pueda hacer predicciones sobre datos no etiquetados.

Si recordamos el ejemplo 1.2.4, teníamos cuatro datos etiquetados, es decir, sabíamos a qué clase social pertenecía cada individuo; la idea ahora es que, con los parámetros ya ajustados, la red pueda predecir o clasificar nuevos datos donde no se conozca la salida, es decir, utilizar un nuevo conjunto de datos $\mathcal{D}^* = \{X_i\}_{i=1}^n$ que no tenga asociado ningún valor observado y_i.

Para lograr que la red sea capaz de hacer dichas predicciones o clasificaciones, necesitamos que el error entre y_i y $\hat{y}_i$ sea suficientemente pequeño para cada observación del conjunto de entrenamiento. En términos prácticos, esto significa que existe una tolerancia $\epsilon > 0$ tal que el error de predicción satisfaga

$$|E(X_i)| = L_i < \epsilon$$

El parámetro ϵ representa la tolerancia permitida entre el valor observado y el valor predicho. Así, garantizaremos que los valores ajustados para Θ generalizan adecuadamente, permitiendo una buena predicción para un nuevo dato X_{n+1} no utilizado durante el entrenamiento.

Siguiendo a (Goodfellow et al., 2016) podemos definir estos conceptos de la siguiente manera:

Definición 1.2.7. La función de pérdida (*loss function*), denotada por L_i, es una función $L_i(y_i, N_\Theta(X_i)) \geq 0$ que representa el error para la i-ésima entrada durante el entrenamiento. Mide la distancia entre el valor observado y_i y el valor predicho $N_\Theta(X_i)$ para una instancia particular de datos X_i.

En el caso particular del perceptrón simple, la salida de la red coincide con la salida de su única neurona, es decir

$$N_\Theta(X_i) = \hat{y}_i(X_i),$$

por simplicidad de notación, en la sección anterior se utilizó $\hat{y}_i(X_i)$. Sin embargo, es importante notar que, para una red multicapa, o una red monocapa con más de una neurona, $\hat{y}_i(X_i)$ representa el valor de una neurona, mientras que $N_\Theta(X_i)$ corresponde con la predicción de la red neuronal.

Existen distintas maneras de representar dicho error, dependiendo del problema que se busque modelar. Por ejemplo, puede utilizarse el error absoluto:

$$L_i = |y_i - N_\Theta(X_i)|$$

o el error cuadrático

$$L_i = (y_i - N_\Theta(X_i))^2$$

En algunos ejemplos utilizaremos el error cuadrático debido a que produce funciones suaves y diferenciables, lo que facilita el uso de métodos de optimización basados en gradientes.

En el ejemplo 1.2.4, cada vez que comparábamos el valor observado con el valor predicho, lo que estábamos haciendo era calcular la función de pérdida:

$$L_i = y_i - \hat{y}_i(X_i)$$

y esto nos daba un parámetro de ajuste: si dicha diferencia era cero, entonces no era necesario actualizar Θ, pero si era distinta de cero, había que ajustar los parámetros.

Por otra parte tenemos la siguiente

Definición 1.2.8. La función objetivo (*objective function*), denotada por

$$J(L_1, L_2, \ldots, L_n) \geq 0$$

es una función que representa el error global y es la que se busca minimizar.

Dado que las funciones de pérdida L_i y la función objetivo J dependen de $N_\Theta(X_i)$, comúnmente suelen denotarse por $L_i(\Theta)$ y $J(\Theta)$ para indicar que hay dependencia de todos los parámetros de la red.

Una elección frecuente de función objetivo es el promedio del error cuadrático.

$$J(\Theta) = \frac{1}{n}\sum_{i=1}^{n} L_i = \frac{1}{n}\sum_{i=1}^{n}(y_i - N_\Theta(X_i))^2.$$

En el caso particular del perceptrón, suele elegirse la suma de los errores cuadráticos:

$$J(\Theta) = \frac{1}{2}\sum_{i=1}^{n}(y_i - \hat{y}_i(X_i))^2.$$

Sustituyendo la salida $\hat{y}_i(X_i)$ del perceptrón tenemos:

$$J(\Theta) = \frac{1}{2}\sum_{i=1}^{n}(y_i - \operatorname{sgn}(X_i \cdot W))^2, \tag{1.1}$$

donde el factor $\frac{1}{2}$ se introduce únicamente para simplificar el cálculo de derivadas parciales durante el proceso de entrenamiento.

Aunque la función objetivo depende tanto de los pesos sinápticos como de los sesgos, es común utilizar un modelo de neurona con sesgo implícito. Por esta razón, en la ecuación (1.1) el sesgo no se considera de manera separada.

En el caso del perceptrón y de las redes que exploraremos a continuación, utilizaremos el Error Cuadrático Medio:

$$J(\Theta) = \frac{1}{n}\sum_{i=1}^{n} L_i$$

o la suma de las funciones de pérdida:

$$J(\Theta) = \frac{1}{2}\sum_{i=1}^{n} L_i$$

dependiendo del ejemplo que queramos mostrar.

Una vez aclarado lo anterior, vamos a describir paso a paso cómo es que se optimiza la función objetivo $J(\Theta)$ a través de los ajustes al vector de pesos $\tilde{W}$ para que nuestra Red Neuronal Artificial sea capaz de hacer predicciones o clasificaciones.

Para el siguiente proceso tomaremos $\tilde{W} = [w_1, w_2, \ldots, w_n, b]$, y por consiguiente consideraremos $\tilde{X} = [x_1, x_2, \ldots, x_n, 1]$, además $J(\Theta)$ será el error cuadrático medio.

Sin embargo, también podemos pensar dicha función objetivo $J(\Theta)$ como una función de los pesos sinápticos y los sesgos. Lo que resultará particularmente útil durante el algoritmo de optimización.

Para visualizarlo mejor, supongamos que tenemos una red con dos neuronas en la capa oculta, es decir, $H = [h_1, h_2]$; tenemos además una neurona en la capa de salida, esto es, $O = [o_1]$ y que nuestro vector de datos es el mismo del ejemplo 1.2.4, o sea, el vector de entrada contempla: el material del suelo, la cantidad de focos en la vivienda, el número de habitantes y la cantidad de vehículos; pero ahora con la forma $\tilde{X}_i = [x_1, x_2, x_3, x_4, 1]$ porque estamos considerando el sesgo implícito. Veamos el esquema (1.1):

Sabemos que cada neurona tiene su función de activación ϕ que depende de los pesos sinápticos y del sesgo en cada una de ellas, así, redefiniremos nuestra capa oculta como $\tilde{H} = [h_1, h_2, 1]$ donde cada neurona de $\tilde{H}$ tiene esta forma:

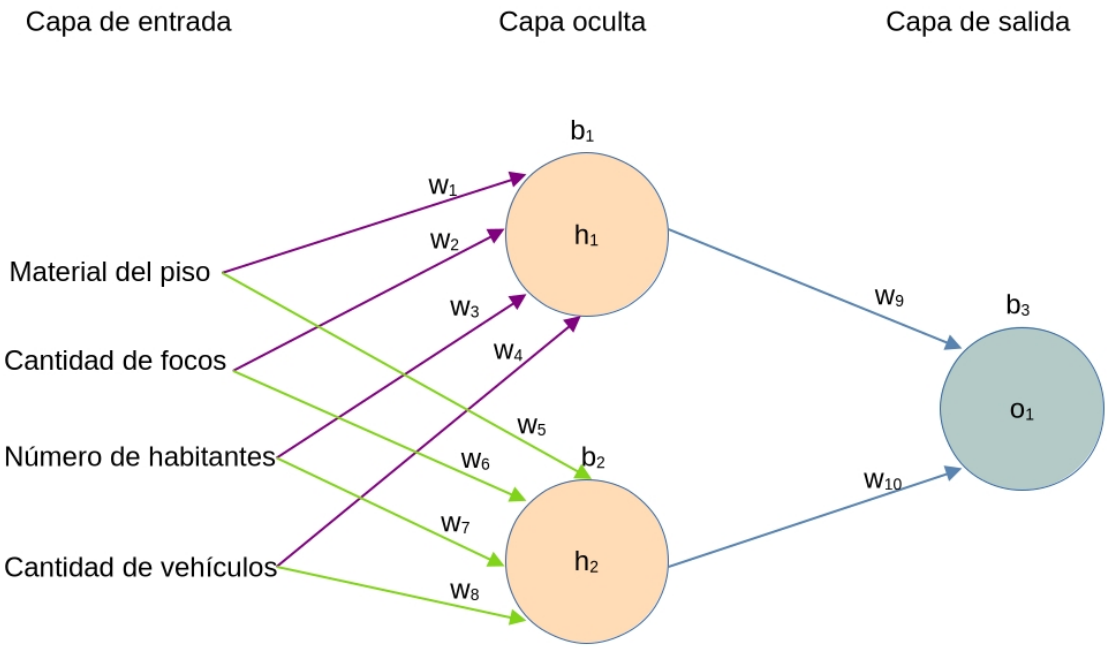
$$h_1(X) = \phi(\tilde{W_{h1}} \cdot X) = \phi(w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b_1)$$

$$h_2(X) = \phi(\tilde{W_{h2}} \cdot X) = \phi(w_5x_1 + w_6x_2 + w_7x_3 + w_8x_4 + b_2)$$

con $\tilde{W_{h1}} = [w_1, w_2, w_3, w_4, b_1]$ y $\tilde{W_{h2}} = [w_5, w_6, w_7, w_8, b_2]$. La neurona de la capa de salida $O = [o_1]$ tiene como entradas a $A_1 = \tilde{H}(X_i) = [h_1(X_i), h_2(X_i), 1]$ puede expresarse así:

$$o_1(A_1) = \phi(\tilde{W_{o1}} \cdot A_1) = \phi(w_9h_1 + w_{10}h_2 + b_3), \quad \text{con } \tilde{W_{o1}} = [w_9, w_{10}, b_3].$$

Figura 1.1: Esquema de red de dos neuronas en la capa oculta y una en la capa de salida



La idea es minimizar la función objetivo, y sabemos que si modificamos los valores de los pesos w_i y de los sesgos b_j, cambiará el resultado de $J(\Theta)$, lo que queremos saber ahora es cómo se modifica la función objetivo si cambiamos un peso en específico. En otras palabras, buscamos una forma de saber si, al modificar el valor de w_1, por ejemplo, la función $J(\Theta)$ crece o decrece, y para ello, podemos utilizar la derivada.

Para simplificar los cálculos, supongamos que tenemos un solo dato

$$\tilde{X}_1 = [x_1, x_2, x_3, x_4, 1],$$

de esta forma, podemos expresar nuestra función objetivo de la siguiente manera:

$$J(\Theta) = \frac{1}{1}\sum_{i=1}^{1}(y_i - N_\Theta(X_1))^2 \tag{1.2}$$

En este caso particular, tenemos dos neuronas en la capa oculta, y una en la capa de salida, entonces $\Theta = [\tilde{W_{h1}}, \tilde{W_{h2}}, \tilde{W_{o1}}]$, donde los sesgos se consideran como una componente del vector de pesos sinápticos.

Imaginemos que cambiamos el valor de w_1, queremos saber cómo cambiará $J(\Theta)$ en consecuencia, para ello, debemos encontrar la derivada parcial de $J(\Theta)$ con respecto de w_1 y por regla de la cadena tenemos que:

$$\frac{\partial J(\Theta)}{\partial w_1} = \frac{\partial J(\Theta)}{\partial N_\Theta(X_1)} \cdot \frac{\partial N_\Theta(X_1)}{\partial w_1}$$

Ahora, por la ecuación (1.2), sabemos que $J(\Theta) = (y - N_\Theta(X_1))^2$, porque tenemos un solo dato, así, podemos expresar la derivada parcial como:

$$\begin{aligned}\frac{\partial J(\Theta)}{\partial N_\Theta(X_1)} &= \frac{\partial}{\partial N_\Theta(X_1)}(y - N_\Theta(X_1))^2 \\ &= 2(y - N_\Theta(X_1)) \cdot \frac{\partial}{\partial N_\Theta(X_1)}(y - N_\Theta(X_1)) \\ &= 2(y - N_\Theta(X_1))(-1) \\ &= -2(y - N_\Theta(X_1))\end{aligned}$$

Para encontrar la derivada parcial de $N_\Theta(X_1)$ con respecto de w_1 vamos a utilizar el hecho de que $N_\Theta(X_1) = o_1 = \phi(w_9 h_1 + w_{10} h_2 + b_3)$ que es una función que depende de w_9, w_{10} y b_3 según nuestro esquema. Sin embargo, siguiendo el mismo esquema, podemos observar que w_1 sólo afecta a h_1, de esta manera, por la regla de la cadena, tenemos que:

$$\frac{\partial N_\Theta(X_1)}{\partial w_1} = \frac{\partial N_\Theta(X_1)}{\partial h_1} \cdot \frac{\partial h_1}{\partial w_1}$$

Así

$$\begin{aligned}\frac{\partial N_\Theta(X_1)}{\partial h_1} &= \frac{\partial}{\partial h_1}\phi(h_1 w_9 + h_2 w_{10} + b_3) \\ &= w_9 \cdot \frac{\partial}{\partial z_3}\phi(h_1 w_9 + h_2 w_{10} + b_3)\end{aligned}$$

con $z_3 = \tilde{W}_{o1} \cdot \tilde{H} = w_9 h_1 + w_{10} h_2 + b_3$

Ahora, para encontrar la parcial de h_1 con respecto de w_1 sabemos que:

$$h_1 = \phi(\tilde{W}_{h1} \cdot \tilde{X}) = \phi(w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4 x_4 + b_1)$$

así:

$$\begin{aligned}\frac{\partial h_1}{\partial w_1} &= \frac{\partial}{\partial w_1}\phi(w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4 x_4 + b_1) \\ &= x_1 \cdot \frac{\partial}{\partial z_1}\phi(w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4 x_4 + b_1)\end{aligned}$$

con $z_1 = \tilde{W}_{h1} \cdot \tilde{X} = w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4 x_4 + b_1$

Finalmente, por las ecuaciones anteriores, tenemos que:

$$\frac{\partial J(\Theta)}{\partial w_1} = \frac{\partial J(\Theta)}{\partial N_\Theta(X_1)} \cdot \frac{\partial N_\Theta(X_1)}{\partial h_1} \cdot \frac{\partial h_1}{\partial w_1}$$

de donde se deduce

$$\frac{\partial J(\Theta)}{\partial w_1} = -2(y - N_\Theta(X_1)) \cdot w_9 \cdot \frac{\partial \phi(z_3)}{\partial z_3} \cdot x_1 \cdot \frac{\partial \phi(z_1)}{\partial z_1} \tag{1.3}$$

La expresión $\dfrac{\partial J(\Theta)}{\partial w_1}$ que acabamos de encontrar nos indica cómo un pequeño cambio en el peso w_1 afecta la función objetivo global $J(\Theta)$. Esta información es la que el algoritmo de optimización utilizará para actualizar los pesos de forma que se reduzca el error. De esta manera, al encontrar las derivadas parciales con respecto a cada uno de nuestros pesos sinápticos y nuestros sesgos de toda la red, tendremos un parámetro que nos indica si la función crece o decrece. Con ello, sabremos si es necesario incrementar o disminuir cada valor para que, en consecuencia, nuestra función objetivo $J(\Theta)$ disminuya. Eso lo veremos más adelante pues, en la siguiente sección, revisaremos el concepto de *función de activación*. Sin embargo, antes de continuar, debemos revisar detenidamente una función objetivo denominada *entropía cruzada categórica*.

Entropía cruzada categórica

Hemos visto que el error cuadrático medio es una buena función objetivo debido a que es derivable. Sin embargo, como veremos en las siguientes secciones, esta función es muy útil para resolver problemas de clasificación binaria, pero resulta poco conveniente si se tienen más de dos clases, lo que se conoce como problema de clasificación multiclase.

En problemas multiclase, donde la salida es un vector de probabilidades, el error cuadrático medio presenta dos inconvenientes: en primer lugar, no tiene una interpretación probabilística directa, y en segundo lugar, combinado con la función de activación Softmax, produce gradientes muy pequeños cuando la red predice con alta confianza una clase incorrecta, ralentizando el aprendizaje.

Una forma ingenua de extender la clasificación binaria a tres clases sería asignar valores numéricos como $y = 0$ para clase baja, $y = 0.5$ para clase media y $y = 1$ para clase alta. Sin embargo, esto introduce un orden artificial al asumir que la clase media es "la mitad" de la clase alta, y supone que las distancias entre clases son iguales, lo que generalmente no es cierto. Por esta razón, en clasificación multiclase se utiliza codificación *one-hot*, donde cada clase se representa como un vector binario con un único 1 en la posición correspondiente a la clase verdadera.

Definición 1.2.9. Sea un problema de clasificación con k clases posibles. Se define la codificación *one-hot* como una representación vectorial de las clases mediante vectores de $\mathbb{R}^k$ tales que exactamente una de sus componentes es igual a 1 y las restantes son idénticamente 0.

Es decir, para cada clase C_i se asocia un vector

$$e_i = [0, \ldots, 0, 1, 0, \ldots, 0] \in \mathbb{R}^k,$$

donde el valor 1 aparece únicamente en la i-ésima componente.

Ahora, en lugar de que $y \in \{0, 1\}$ sea un escalar (para dos categorías), representamos la clase verdadera mediante un vector de k componentes, donde cada componente corresponde a una categoría.

Ejemplo 1.2.5. Supongamos que deseamos representar tres clases (baja, media, alta) como salida para las observaciones de un conjunto de datos $\mathcal{D} = \{(X_i, y_i)\}_{i=1}^n$ ahora podemos asociar a un individuo de clase baja como

$$Clasebaja \mapsto [1, 0, 0],$$

uno de clase media como

$$Clasemedia \mapsto [0, 1, 0],$$

y uno de clase alta como

$$Clasealta \mapsto [0, 0, 1].$$

De esta manera, cada clase queda representada por un vector distinto.

Obsérvese que los vectores obtenidos mediante codificación *one-hot* corresponden con los vectores de la base canónica de $\mathbb{R}^k$.

Con la codificación *one-hot* podemos representar una salida observada y_{ij} cuando se tienen más de dos clases. En este caso, i representa la i-ésima observación, mientras que j indica que la observación pertenece a ja j-ésima clase. Para realizar predicciones sobre observaciones con más de dos clases, resulta más conveniente utilizar una función objetivo $J(\Theta)$ denominada "entropía cruzada categórica".

En primera instancia, vamos a entender la idea general de la "entropía cruzada categórica" a través de un ejemplo muy simple tomado directamente de (Martín López, 2024). Supongamos que tenemos una bolsa con cuatro bolas de diferentes colores: azul, roja, verde y amarilla. Se extrae al azar alguna bola. El objetivo es determinar, con el menor número de preguntas (cuya respuesta sea "sí" o "no"), el color de la bola extraída. Una

estrategia consiste en preguntar si la bola es roja o azul. Si la respuesta es afirmativa, entonces preguntamos si es azul. Si la respuesta es negativa, sabremos que la bola es verde o amarilla, así que sólo debemos preguntar si es verde. De esta manera, con sólo dos preguntas podemos determinar el color de la bola. Tomando en cuenta que, en este ejemplo, la probabilidad de sacar cualquier color es de 1/4.

Ahora supongamos que en la bolsa hay 12 bolas azules, 6 rojas, 3 verdes y 3 amarillas. Esta vez, la estrategia sería preguntar primero si la bola es azul, ya que la mitad de las bolas en la bolsa es azul, o sea que tenemos $P(azul) = 1/2$ de probabilidad de acertar, si la respuesta es negativa, entonces podríamos preguntar si es roja pues $P(roja) = 1/4$ y si la respuesta es negativa, entonces ya podemos preguntar si es verde (o amarilla, pues ambos colores tienen la misma probabilidad). Entonces, si es azul, nos cuesta una pregunta determinarlo, si es roja, nos lleva dos preguntas, y si es verde o amarilla nos lleva tres preguntas. Por lo tanto, el número de preguntas promedio para adivinar de qué color es la bola es:

$$\frac{1}{2} \cdot 1(azul) + \frac{1}{4} \cdot 2(roja) + \frac{1}{8} \cdot 3(verde) + \frac{1}{8} \cdot 3(amarilla) = \frac{7}{4} = 1.75,$$

donde el 1, el 2 y el 3 corresponde con el número de preguntas necesarias para encontrar el color.

Si todas las bolas en la bolsa fueran del mismo color, entonces no necesitaríamos hacer ninguna pregunta, ya sabemos el color de cualquier bola que se saque al azar.

Así, si solo podemos hacer preguntas cuya respuesta sea afirmativa o negativa, tenemos que: ninguna pregunta nos da una sola posibilidad (o sea la respuesta segura, donde la probabilidad $P(x) = 1$)); una pregunta nos permite distinguir dos posibilidades (si tenemos dos colores, basta con una pregunta que nos indique si la bola es de alguno de esos colores); dos preguntas nos ayudarán a distinguir cuatro posibilidades (como en el problema planteado previamente); y tres preguntas nos ayudarían a distinguir 8 posibilidades; y así sucesivamente. Las posibilidades que podemos encontrar con cada pregunta se van multiplicando en potencias de dos. Si hacemos k preguntas, podremos encontrar 2^k posibles categorías.

Si tenemos una bolsa con bolas de 32 colores distintos idénticamente distribuidos (es decir, cada color tiene el mismo número de bolas), entonces necesitamos en promedio cinco preguntas para poder adivinar el color de una bola sacada al azar. Pero si la distribución no es uniforme (es decir, hay más bolas de algún color que de otro), tenemos mayor probabilidad de sacar algún color; entonces podemos reducir el promedio de preguntas necesarias, como se observó previamente, donde el promedio de preguntas es menor que dos, a pesar de que son los mismos cuatro colores.

Entonces, si nuestra distribución no es uniforme, utilizaremos la primera pregunta para

descartar el color más probable, la segunda para el segundo más probable, y así, hasta que nuestra quinta pregunta sea preguntar por el segundo color menos probable (el color menos probable se deduce al descartar el segundo menos probable). Y para encontrar el número de preguntas promedio tendríamos que multiplicar la probabilidad del color más probable por el número de preguntas que nos llevaría a encontrarlo, y sumarle las probabilidades de los siguientes colores multiplicadas con el número de preguntas que usamos para determinar el color.

Retomemos el problema de los cuatro colores con distribución no uniforme. Sabemos que la probabilidad de que la bola sea azul es $P(azul) = 1/2$, la probabilidad de que sea roja es $P(roja) = 1/4$ y las probabilidades de que sea verde o amarilla son $P(verde) = 1/8$ y $P(amarilla) = 1/8$ respectivamente. Entonces, si quisiéramos saber cuántas preguntas de sí o no debemos hacer para encontrar cada color sabiendo esas probabilidades, tendríamos que el número de preguntas necesarias para identificar una bola de color C es $-\log_2(P(C))$.

Donde el negativo se coloca para que, al calcular el logaritmo de un número $0 < P(C) \leq 1$ que siempre es no positivo, el resultado sea no negativo y tenga sentido con la interpretación del número de preguntas necesarias para obtener la información necesaria.

Cuando conocemos la distribución real P y diseñamos nuestra estrategia basada en ella, el número promedio de preguntas se llama entropía de P. Sin embargo, en la práctica no siempre conocemos P; debemos asumir una distribución Q (nuestra estrategia). El número promedio de preguntas que obtenemos al usar Q cuando la realidad es P se llama entropía cruzada entre P y Q. Nuestro objetivo es minimizar esta cantidad, haciendo que Q se acerque lo más posible a P.

En clasificación multiclase, una elección común para la función objetivo consiste en utilizar la entropía cruzada categórica:

Definición 1.2.10. Si se tiene un problema de clasificación con k clases posibles y un conjunto de entrenamiento $\mathcal{D} = \{(X_j, y_j)\}_{j=1}^{n}$ compuesto por n ejemplos, se define la entropía cruzada categórica como $J(\Theta) \geq 0$ de la siguiente forma:

$$J(\Theta) = -\frac{1}{n}\sum_{j=1}^{n}\sum_{i=1}^{k} y_{ji}\log(N_\Theta(X_j)_i),$$

donde:

- y_{ji} representa la i-ésima componente del vector de salida observado asociado al ejemplo j mediante codificación *one-hot*;
- $N_\Theta(X_j)_i$ representa la probabilidad predicha por la red neuronal para que el

ejemplo j pertenezca a la clase i.

La entropía cruzada mide la discrepancia entre la distribución real de clases y la distribución predicha por la red neuronal.

Ejemplo 1.2.6. Retomemos el problema de clasificación del nivel socioeconómico, pero ahora supongamos que tenemos tres clases posibles:

$$\text{baja}, \qquad \text{media}, \qquad \text{alta}.$$

Mediante codificación *one-hot*, representaremos dichas clases como:

$$\text{baja} = [1,0,0], \qquad \text{media} = [0,1,0], \qquad \text{alta} = [0,0,1].$$

Supongamos que tenemos tres datos de entrenamiento y que la red neuronal produce las siguientes predicciones:

$$y_1 = [1,0,0], \qquad N_\Theta(X_1) = [0.80, 0.15, 0.05],$$

$$y_2 = [0,1,0], \qquad N_\Theta(X_2) = [0.20, 0.65, 0.15],$$

$$y_3 = [0,0,1], \qquad N_\Theta(X_3) = [0.10, 0.25, 0.65].$$

Obsérvese que, en cada caso, la suma de las probabilidades predichas es igual a 1, pues la salida de la red corresponde con una distribución de probabilidad sobre las clases posibles, es decir, en $N_\Theta(X_1) = [0.80, 0.15, 0.05]$ podemos ver que:

$$\sum_{i=1}^{3} N_\Theta(X_1)_i = 0.80 + 0.15 + 0.05 = 1$$

La entropía cruzada categórica está dada por

$$J(\Theta) = -\frac{1}{3}\sum_{j=1}^{3}\sum_{i=1}^{3} y_{ji}\log(N_\Theta(X_j)_i).$$

Veamos con detalle el cálculo para el primer ejemplo, es decir, para $j = 1$. Sabemos que

$$y_1 = [1,0,0], \qquad N_\Theta(X_1) = [0.80, 0.15, 0.05].$$

Entonces:

$$\sum_{i=1}^{3} y_{1i} \log(N_\Theta(X_1)_i) = y_{11} \log(N_\Theta(X_1)_1) + y_{12} \log(N_\Theta(X_1)_2) + y_{13} \log(N_\Theta(X_1)_3).$$

Sustituyendo valores:

$$= 1 \cdot \log(0.80) + 0 \cdot \log(0.15) + 0 \cdot \log(0.05).$$

Como los dos últimos términos están multiplicados por cero, obtenemos:

$$= \log(0.80).$$

Por tanto, la contribución del primer ejemplo a la entropía cruzada es:

$$-\log(0.80) = 0.2231.$$

De manera análoga, como los vectores y_j están codificados mediante *one-hot*, en cada suma sólo queda el término correspondiente a la clase verdadera. Por tanto:

$$J(\Theta) = -\frac{1}{3} \left[\log(0.80) + \log(0.65) + \log(0.65)\right].$$

Calculando:

$$J(\Theta) = -\frac{1}{3} \left[-0.2231 - 0.4308 - 0.4308\right].$$

$$J(\Theta) = \frac{1.0847}{3} = 0.3615.$$

Así, el valor de la función objetivo para estos tres ejemplos es aproximadamente:

$$J(\Theta) \approx 0.3615.$$

Este valor representa el error promedio entre la distribución real de clases, dada por los vectores *one-hot*, y la distribución de probabilidad predicha por la red neuronal. Si la red asignara probabilidades más cercanas a 1 en las clases correctas, el valor de $J(\Theta)$ sería menor.

Como podemos observar, en este caso las predicciones son adecuadas, porque la mayor probabilidad de predicción corresponde con la clase correcta.

Ahora supongamos el caso en el que la red neuronal produce predicciones inadecuadas, es decir, que la probabilidad más alta no corresponde con la clase verdadera:

$$y_1 = [1, 0, 0], \qquad N_\Theta(X_1) = [0.10, 0.80, 0.10],$$

$$y_2 = [0, 1, 0], \qquad N_\Theta(X_2) = [0.70, 0.20, 0.10],$$

$$y_3 = [0, 0, 1], \qquad N_\Theta(X_3) = [0.60, 0.30, 0.10].$$

Entonces:

$$J(\Theta) = -\frac{1}{3}\left[\log(0.10) + \log(0.20) + \log(0.10)\right].$$

Calculando:

$$J(\Theta) = -\frac{1}{3}\left[-2.3026 - 1.6094 - 2.3026\right].$$

$$J(\Theta) = \frac{6.2146}{3} = 2.0715.$$

Por tanto:

$$J(\Theta) \approx 2.0715.$$

Como podemos observar, el valor de la función objetivo aumentó considerablemente con respecto al caso anterior, donde obtuvimos

$$J(\Theta) \approx 0.3615.$$

Esto ocurre porque la red asignó probabilidades pequeñas a las clases verdaderas. La entropía cruzada categórica penaliza precisamente este comportamiento: mientras menor sea la probabilidad asignada a la clase correcta, mayor será el valor de la función objetivo. De ahí que surja la necesidad de minimización.

Nota: Aunque en teoría de la información se utiliza el logaritmo base 2 para medir información en bits, en la práctica del aprendizaje automático se prefiere el logaritmo natural (base e) porque su derivada es más simple y facilita los cálculos en los algoritmos de optimización.

1.2.3 Funciones de activación

En la sección anterior, acabamos de encontrar una manera de verificar si un cambio en los parámetros de la red neuronal (o sea, los pesos y los sesgos) hace crecer o decrecer

la *función objetivo* $J(\Theta)$, sin embargo, en nuestra expresión (1.3) llegamos a:

$$\frac{\partial J(\Theta)}{\partial w_1} = -2(y - N_\Theta(X_i)) \cdot w_9 \cdot \frac{\partial \phi(z_3)}{\partial z_3} \cdot x_1 \cdot \frac{\partial \phi(z_1)}{\partial z_1},$$

donde ϕ es la *función de activación*. Por ello, resulta indispensable detenernos a verificar qué es una función de activación y qué papel desempeña en el "aprendizaje" de la Red Neuronal Artificial.

Anteriormente habíamos mencionado que las neuronas biológicas presentan excitabilidad, es decir, que se activan si reciben un estímulo suficiente del axón de la neurona anterior con la que hace sinapsis, dicho estímulo se alcanza a través de una diferencia de potencial que a su vez determina si la neurona en cuestión generará otro estímulo para la neurona siguiente o si permanecerá inactiva.

Las neuronas artificiales intentan emular dicha comunicación neuronal a través de las *funciones de activación*. Las diferencias de potencial en las neuronas biológicas se expresarán como valores numéricos en las neuronas artificiales. Si se sobrepasa un cierto valor τ, denominado *umbral*, entonces la neurona se activará y mandará la información a la siguiente neurona, pero si no se sobrepasa, entonces la neurona permanecerá inactiva. Supongamos, entonces, que mi umbral está determinado por $\tau = 0.7$, entonces, todos los valores $h_i(A_i) < \tau$ de la neurona anterior harán que la neurona actual permanezca inactiva, mientras que todos los valores $h_i(A_i) > \tau$ harán que la neurona se active. Sin embargo, necesitamos que la salida de la neurona no oscile entre valores muy dispares, pues eso haría que: se activara muy rara vez (si la mayoría de entradas fueran < 0, por ejemplo), o se activara casi siempre (si dichos valores fueran > 1). En cambio, si los valores de salida de la neurona anterior se encuentran en un rango acotado, como $0 < h_i(A_i) < 1$ o $-1 < h_i(A_i) < 1$, es más fácil para la capa siguiente aprender patrones, ya que la magnitud de las activaciones no crece descontroladamente a lo largo de la red. Esto evita problemas de inestabilidad numérica y contribuye a la estabilidad del proceso de optimización. Por ello la *función de activación* es una herramienta que permite controlar los valores de salida de mi neurona h_i independientemente de la disparidad que pueda existir en mis valores de entrada X_i.

Existen diversas *funciones de activación* que podemos utilizar en una Red Neuronal Artificial. De hecho, no es necesario que se utilice una sola *función de activación*, y pueden cambiarse en las diversas capas, es decir, puede utilizarse una función específica en la primera capa oculta, otra en la segunda capa oculta y una tercera para la capa de salida. Más aún, dentro de una misma capa se pueden cambiar las funciones de activación, como se mostró en el ejemplo (1.2.3). Esto dependerá del rango entre el que nos interesa que oscilen los valores de salida de cada capa. Sin embargo, (Martín del Brío & Sanz Molina, 2001) menciona que dichas funciones suelen ser monótonas crecientes y continuas, además

de que algunos algoritmos de optimización requieren que se cumpla la condición de ser derivable.

Podemos decir que la *función de activación* es una transformación matemática no lineal que se aplica a una combinación lineal de las entradas de una neurona. Para que la red pueda representar relaciones complejas y no lineales presentes en los datos, es necesario introducir una fuente de no linealidad. La *función de activación* es la encargada de introducir esa fuente de no linealidad.

Además, la *función de activación* cumple con otro propósito fundamental. Debido a que el aprendizaje de la red se basa en el cálculo de la función de pérdida con respecto de los pesos, una *función de activación* adecuada tiene una derivada útil para el aprendizaje, es decir, debe indicar con claridad si la *función objetivo* es creciente y en qué dirección podemos hacerla decrecer. En otras palabras, necesitamos saber cómo actualizar los pesos sin provocar que la neurona permanezca saturada o inactiva durante el entrenamiento.

En resumen, la función de activación es la que otorga a la red la capacidad de "aprendizaje", permitiéndole elaborar representaciones jerárquicas y abstractas de los datos de entrada (Goodfellow et al., 2016)

Pese a la gran diversidad de *funciones de activación* que existen para múltiples propósitos, nos centraremos únicamente en el análisis de tres de ellas: la función sigmoide, la función tangente hiperbólica y la función *softmax*, debido a que son las tres funciones presentes en el modelo que presentaremos más adelante. Sin embargo, es importante mencionar que existe otra *función de activación* que se utiliza con mucha frecuencia en las capas ocultas de redes neuronales profundas. Dicha función se denomina *Unidad Lineal Rectificada (ReLU)* y está definida como $\mathrm{ReLU}(z) = \max(0, z)$, y debe su popularidad a que mitiga el problema del desvanecimiento del gradiente al tener una derivada igual a 1 cuando $z > 0$.

La función Sigmoide

La función sigmoide (sigmoidea o logística) es una *función de activación* fundamental en la historia de las redes neuronales artificiales.

Definición 1.2.11. La función sigmoide, denotada por σ, es una función

$$\sigma : \mathbb{R} \to \mathbb{R}$$

definida por

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

Es importante mencionar que, aunque la función σ está definida para cualquier número

real, en particular, en la bibliografía sobre Redes Neuronales Artificiales, suele definirse en función de $z = X \cdot W + b = \sum_{i=1}^{n}(w_i x_i) + b$. Por ello, conservamos esta notación en la definición.

En el libro *Neural Networks and Deep Learning*, el autor recalca que "la derivada de la función sigmoide es particularmente simple cuando se expresa en términos de la salida de la sigmoide, en lugar de en términos de la entrada" (Aggarwal, 2018b).

Notemos que si $\sigma(z) = \dfrac{1}{1+e^{-z}}$ entonces:

$$\begin{aligned}
\frac{\partial \sigma(z)}{\partial z} &= \frac{\frac{\partial}{\partial z}1 \cdot (1+e^{-z}) - 1 \cdot \frac{\partial}{\partial z}(1+e^{-z})}{(1+e^{-z})^2} \\
&= \frac{0 \cdot (1+e^{-z}) - \frac{\partial}{\partial z}(1+e^{-z})}{(1+e^{-z})^2} \\
&= \frac{-\frac{\partial}{\partial z}1 - \frac{\partial}{\partial z}e^{-z}}{(1+e^{-z})^2} = \frac{-0 - \frac{\partial}{\partial z}e^{-z}}{(1+e^{-z})^2} \\
&= \frac{-\frac{\partial}{\partial z}e^{-z}}{(1+e^{-z})^2} = \frac{-e^{-z}\frac{\partial}{\partial z}(-z)}{(1+e^{-z})^2} \\
&= \frac{-(-1)e^{-z}}{(1+e^{-z})^2} \\
&= \frac{e^{-z}}{(1+e^{-z})^2}
\end{aligned}$$

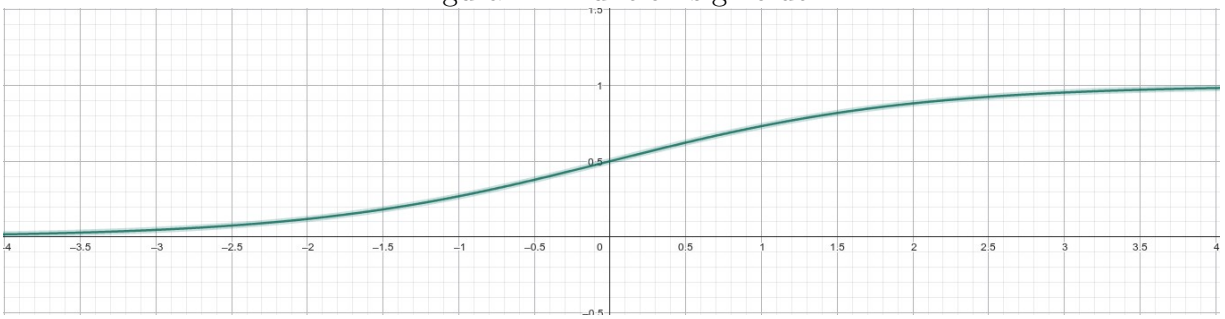
Ahora, notemos también que:

$$\begin{aligned}
\frac{e^{-z}}{(1+e^{-z})^2} &= \frac{e^{-z}+1-1}{(1+e^{-z})^2} \\
&= \frac{1+e^{-z}-1}{(1+e^{-z})^2} \\
&= \frac{1+e^{-z}}{(1+e^{-z})^2} - \frac{1}{(1+e^{-z})^2} \\
&= \frac{1}{(1+e^{-z})} - \frac{1}{(1+e^{-z})^2} \\
&= \frac{1}{(1+e^{-z})}\left(1 - \frac{1}{(1+e^{-z})}\right) \\
&= \sigma(z)(1-\sigma(z))
\end{aligned}$$

Esta última ecuación es la manera en la que suele presentarse la derivada de la función sigmoide, debido a que el valor de la derivada puede obtenerse con operaciones aritméticas simples, lo que la hace computacionalmente muy eficiente.

La gráfica de la función sigmoide se ve de esta manera:

Figura 1.2: Función sigmoide



Como podemos observar en la gráfica (1.2), si $z \to -\infty$ entonces $\sigma(z) \to 0$, mientras que si $z \to \infty$ entonces $\sigma(z) \to 1$. Además, la función es derivable.

La principal virtud de la función sigmoide es que su rango de salida $0 < \sigma(z) < 1$ permite interpretar su resultado como una probabilidad. Se utiliza, sobre todo en las capas de salida cuando la red es un clasificador binario.

Retomando nuestro ejemplo (1.2.4) en el que la red buscaba clasificar el nivel socioeconómico, si quisiéramos obtener una estimación de la probabilidad de que un individuo pertenzca a la clase alta, en lugar de una clasificación binaria rígida con la función signo en la capa de salida, la función de activación sigmoide sería una opción más adecuada. En este caso, la salida de la neurona sería $N_\Theta(X_i) = \sigma(A_i \cdot W_o + b_o)$, donde un valor de $N_\Theta(X_i)$ cercano a 1 indicaría una alta probabilidad de pertenecer a la clase alta, y un valor cercano a 0, una alta probabilidad de pertenecer a la clase baja. Sin embargo, si quisiéramos clasificar tres clases sociales: baja, media y alta, pese a que técnicamente es posible (como se mencionó anteriormente, porque podríamos imponer artificialmente una regla en la que valores cercanos a 0 implican probabilidad de pertenecer a la clase baja; valores cercanos a $\frac{1}{2}$ probabilidad de pertenecer a la clase media; y valores cercanos a 1 probabilidad de pertenecer a la clase alta); la realidad es que operativamente es muy complicado definir los rangos adecuados para clasificar más de dos clases con una función sigmoide, por ello es que su uso predominante es en clasificadores binarios.

A pesar de las ventajas que tiene la función sigmoide, también tiene limitaciones, especialmente cuando se utiliza en modelos de aprendizaje profundo.

El principal problema de la función sigmoide es el que Goodfellow et al. (2016) denomina como el "problema del desvanecimiento del gradiente". Como puede observarse en la figura (1.2), la función alcanza rápidamente valores cercanos a sus asíntotas horizontales.

Por ejemplo, para $z = 4$ ya se tiene $\sigma(z) \approx 0.98201$, mientras que para $z = -4$ se obtiene $\sigma(z) \approx 0.01871$. En ambos casos, resulta que $\sigma'(z)$ es $\sigma'(4) = (0.98201)(1 - 0.98201) \approx 0.01766$ y $\sigma'(-4) = (0.01871)(1 - 0.01871) \approx 0.01835$ respectivamente, son valores cercanos 0, a pesar de que $\sigma = 4$ no es un valor extremadamente grande y $\sigma = -4$ no es extremadamente pequeño. Para $\sigma = 10$ o $\sigma = -10$ la derivada se volverá extremadamente pequeña.En ese caso se dice que las neuronas están "saturadas". Si varias capas tienen neuronas saturadas, la señal de error que llega a las primeras capas se vuelve extremadamente pequeña y, por lo tanto, se dice que prácticamente se desvanece. Cuando el gradiente es extremadamente pequeño, las actualizaciones de los pesos son tan minúsculas que el aprendizaje prácticamente se detiene y la red no puede seguir mejorando.

Otra desventaja es que su salida no está centrada en cero, siempre es no negativa (porque $\sigma(z) \in (0, 1)$). Esto puede provocar que los gradientes de los pesos sinápticos de una capa tengan todos el mismo signo. Como consecuencia, las actualizaciones de los pesos pueden seguir una trayectoria en forma de "zig-zag", lo que en la práctica se traduce en una convergencia más lenta y menos eficiente hacia el mínimo de la función objetivo.

Finalmente, si hablamos de costo computacional, la función sigmoide involucra el cálculo de una exponencial, que es una operación más costosa computacionalmente que otras funciones más simples, como la mencionada ReLU o la función identidad.

La función Tangente Hiperbólica

La función tangente hiperbólica, denotada comúnmente como $\tanh(z)$, es otra de las funciones de activación fundamentales en el aprendizaje profundo. Puede entenderse como una versión desplazada y escalada de la función sigmoide, y fue durante muchos años la función de activación preferida para las capas ocultas de las redes neuronales.

Definición 1.2.12. La función tangente hiperbólica, denotada por tanh, es una función

$$\tanh : \mathbb{R} \Rightarrow \mathbb{R}$$

definida por

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

Una vez más, (Aggarwal, 2018b) propone otra manera de ver a la función tanh, como sigue:

$$\begin{aligned}\tanh(z) &= \frac{e^z - e^{-z}}{e^z + e^{-z}} \\ &= \frac{\frac{e^z}{1} - \frac{1}{e^z}}{\frac{e^z}{1} + \frac{1}{e^z}} \\ &= \frac{\frac{e^z e^z - 1}{e^z}}{\frac{e^z e^z + 1}{e^z}} \\ &= \frac{e^{2z} - 1}{e^{2z} + 1}\end{aligned}$$

Y de esta forma, la derivada la podemos ver como:

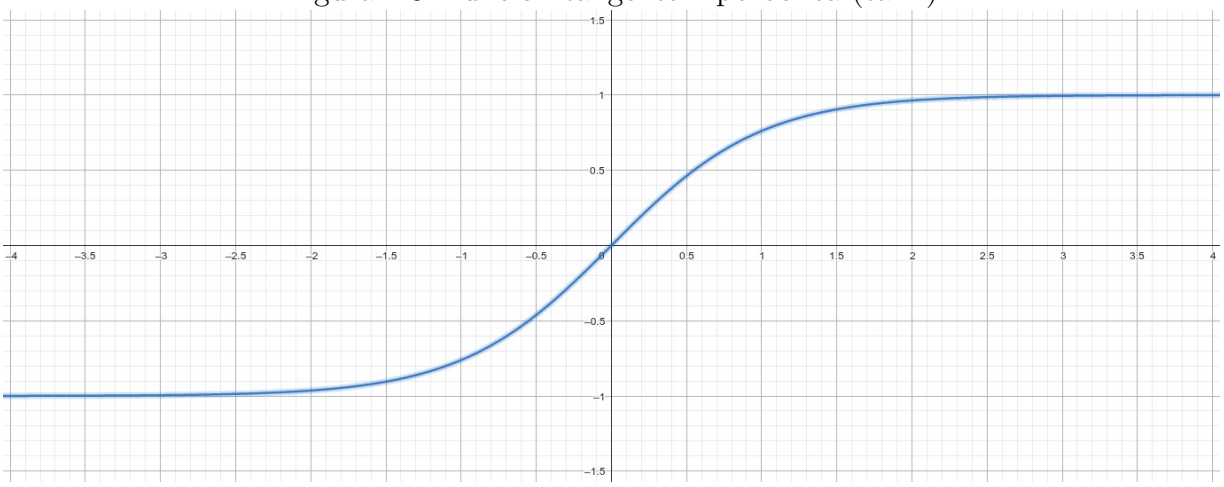
$$\begin{aligned}\frac{\partial}{\partial z}\frac{e^{2z} - 1}{e^{2z} + 1} &= \frac{(e^{2z} - 1)'(e^{2z} + 1) - (e^{2z} - 1)(e^{2z} + 1)'}{(e^{2z} + 1)^2} \\ &= \frac{(2e^{2z})(e^{2z} + 1) - (e^{2z} - 1)(2e^{2z})}{(e^{2z} + 1)^2} \\ &= \frac{(2e^{2z})[(e^{2z} + 1) - (e^{2z} - 1)]}{(e^{2z} + 1)^2} \\ &= \frac{(2e^{2z})(2)}{(e^{2z} + 1)^2} \\ &= \frac{(4e^{2z})}{(e^{2z} + 1)^2}\end{aligned}$$

Pero, similar a lo que vimos con la función sigmoide, la derivada de tanh también puede expresarse de manera simple:

$$\begin{aligned}\frac{(4e^{2z})}{(e^{2z} + 1)^2} &= \frac{(2e^{2z} + 2e^{2z})}{(e^{2z} + 1)^2} \\ &= \frac{(2e^{2z} + 2e^{2z} + e^{4z} + 1 - e^{4z} - 1)}{(e^{2z} + 1)^2} \\ &= \frac{e^{4z} + 2e^{2z} + 1 - e^{4z} + 2e^{2z} - 1}{(e^{2z} + 1)^2} = \frac{(e^{2z} + 1)^2 - (e^{2z} - 1)^2}{(e^{2z} + 1)^2} \\ &= \frac{(e^{2z} + 1)^2}{(e^{2z} + 1)^2} - \frac{(e^{2z} - 1)^2}{(e^{2z} + 1)^2} = 1 - \frac{(e^{2z} - 1)^2}{(e^{2z} + 1)^2} \\ &= 1 - \tanh 2(z)\end{aligned}$$

La gráfica de la tangente hiperbólica es la siguiente:

Figura 1.3: Función tangente hiperbólica (tanh)



Como podemos observar en la gráfica, la función tangente hiperbólica es derivable y guarda cierta similitud con la función sigmoide. Sin embargo, presenta una diferencia fundamental: su salida está centrada en cero, oscilando entre -1 y 1; lo que supone una ventaja sobre la función sigmoide. Esta propiedad facilita el aprendizaje en las capas ocultas porque, al permitir salidas tanto positivas como negativas, los gradientes de los pesos sinápticos pueden tener diferentes signos, lo que reduce las trayectorias de actualización en "zig-zag" que ralentizan la convergencia. Además, Goodfellow et al. (2016) afirma que, al estar los valores centrados en el cero, suele conducirse a un mejor condicionamiento al problema de optimización en las capas siguientes que reciben estos valores como entrada.

Sin embargo, a pesar de las ventajas que representa con respecto de la función sigmoide, la tangente hiperbólica presenta las mismas desventajas: por una parte esta función también se satura en los extremos, es decir, si $z < -10$ o $z > 10$, la función se aproxima asintóticamente a -1 y 1 respectivamente, y su derivada tiende a cero. En consecuencia, el problema del desvanecimiento del gradiente persiste.

Por otra parte, como la tangente hiperbólica también involucra el cálculo de exponenciales, su costo computacional es mayor con respecto de otras funciones.

La función SoftMax

Como hemos visto, la función sigmoide es muy útil como clasificador binario, es capaz ajustar los datos a una distribución de probabilidad con dos categorías: pertenencia o no pertenencia, éxito o fracaso, cero o uno. Es decir, funciona como una distribución Bernoulli. Sin embargo, la función sigmoide, como mencionamos anteriormente, no funciona muy bien si tenemos más de dos categorías.

Ahora, imaginemos que tenemos un problema de clasificación con k clases posibles. En ese caso buscamos que la Red Neuronal Artificial ajuste los datos X_i de un conjunto de

datos $\mathcal{D} = \{(X_i, y_i)\}_{i=1}^n$ donde la salida $y_{ij} \in \mathbb{R}^k$ representada en codificación *one-hot* tiene k clases. Para ello necesitamos que la salida de la red neuronal $N_\Theta(X_i)$ pueda interpretarse como una distribución de probabilidad sobre las k clases. Es decir, buscamos un vector $N_\Theta(X_i) = [o_{i1}, o_{i2}, ..., o_{ik}]$ que cumpla:

$0 < o_{ij} < 1 \forall j$ (cada elemento es una probabilidad)

$\sum_{j=1}^k o_{ij} = 1$ (las probabilidades suman 1)

$o_{ij} = P(C_j|X_i)$

La función sigmoide, que usamos para clasificación binaria, no puede hacer esto porque solo produce un único valor entre 0 y 1. Necesitamos una función que tome un vector de k números reales (las salidas de la última capa) y lo convierta en un vector de k probabilidades que cumplan las condiciones anteriores. Para eso sirve la función SoftMax.

Definición 1.2.13. Sea $k \in \mathbb{N}$. SoftMax es una función

$$\sigma : \mathbb{R}^k \to \mathbb{R}^k$$

definida por

$$\sigma(z) = \big(\sigma(z)_1, \sigma(z)_2, \ldots, \sigma(z)_k\big),$$

donde, para cada $i = \overline{1, k}$,

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^k e^{z_j}}.$$

Notemos que al aplicar la función exponencial a los valores de z_i, por un lado estamos garantizando que los valores resultantes sean siempre positivos; por otro lado, como la función exponencial crece muy rápido, si $z_i > z_j$ para cualesquiera i, j, entonces $e^{z_i} \gg e^{z_j}$, lo que ayuda a que la clase más "segura" tenga una probabilidad dominante. Por otra parte, al dividir cada e^{z_i} sobre $\sum_{j=1}^k e^{z_j}$ (la suma de las exponenciales de todas las clases) tenemos el factor de normalización que garantiza que $\sum_{i=1}^k \sigma(z)_i = 1$.

Una propiedad importante de la función SoftMax es que es invariante ante desplazamientos:

$$\begin{aligned}
\sigma(z+c)_i &= \frac{e^{z_i+c}}{\sum_{j=1}^{k} e^{z_j+c}} \\
&= \frac{e^{z_i}e^{c}}{\sum_{j=1}^{k} e^{z_j}e^{c}} \\
&= \frac{e^{c}e^{z_i}}{e^{c}\sum_{j=1}^{k} e^{z_j}} \\
&= \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}} \\
&= \sigma(z)_i
\end{aligned}$$

Para comprender la derivada de la función Softmax, debemos tener presente una diferencia fundamental con las funciones anteriores: mientras que la sigmoide y la tanh son funciones escalares (una entrada, una salida), la Softmax es una función vectorial: recibe un vector $z = [z_1, z_2, ..., z_k]$ y produce un vector $\sigma(z) = [\sigma(z)_1, \sigma(z)_2, ..., \sigma(z)_k]$.

Por lo tanto, cuando derivamos, no obtenemos una única derivada, sino una matriz de derivadas llamada matriz Jacobiana. Cada elemento de esta matriz, denotado como $\frac{\partial \sigma(z)_i}{\partial z_j}$, nos indica cómo cambia la probabilidad de la clase i cuando modificamos ligeramente el logit de la clase j.

Aquí debemos tomar en cuenta dos casos, cuando $i = j$, es decir, cuando estamos derivando con respecto del mismo índice, y cuando $i \neq j$, es decir, cuando estamos derivando con respecto de un índice distinto.

Cuando $i = j$, estamos derivando la probabilidad de la clase i con respecto a su propio logit z_i.

$$
\begin{aligned}
\frac{\partial \sigma(z)_i}{\partial z_i} &= \frac{\partial}{\partial z_i}\left(\frac{e^{z_i}}{\sum_{m=1}^{k} e^{z_m}}\right) \\
&= \frac{\left(\frac{\partial}{\partial z_i} e^{z_i}\right) \cdot \left(\sum_{m=1}^{k} e^{z_m}\right) - e^{z_i} \cdot \left(\frac{\partial}{\partial z_i} \sum_{m=1}^{k} e^{z_m}\right)}{\left(\sum_{m=1}^{k} e^{z_m}\right)^2} \\
&= \frac{e^{z_i} \cdot \left(\sum_{m=1}^{k} e^{z_m}\right) - e^{z_i} \cdot e^{z_i}}{\left(\sum_{m=1}^{k} e^{z_m}\right)^2} \\
&= \frac{e^{z_i}\left(\sum_{m=1}^{k} e^{z_m} - e^{z_i}\right)}{\left(\sum_{m=1}^{k} e^{z_m}\right)^2} \\
&= \frac{e^{z_i}}{\sum_{m=1}^{k} e^{z_m}} \cdot \frac{\sum_{m=1}^{k} e^{z_m} - e^{z_i}}{\sum_{m=1}^{k} e^{z_m}} \\
&= \sigma(z)_i \left(1 - \sigma(z)_i\right)
\end{aligned}
$$

Calculemos cada derivada por separado:

$$\frac{\partial}{\partial z_i} e^{z_i} = e^{z_i}$$

$$\frac{\partial}{\partial z_i} \sum_{m=1}^{k} e^{z_m} = e^{z_i}$$

porque al derivar la suma, todos los términos donde $m \neq i$ tienen derivada cero, y el término e^{z_i} tiene derivada e^{z_i}

Sustituyendo:

$$\begin{aligned}\frac{\partial}{\partial z_i}\sigma(z)_i &= \frac{e^{z_i}\cdot\left(\sum_{m=1}^{k} e^{z_m}\right) - e^{z_i}\cdot e^{z_i}}{\left(\sum_{m=1}^{k} e^{z_m}\right)^2} \\ &= \frac{e^{z_i}\left(\sum_{m=1}^{k} e^{z_m} - e^{z_i}\right)}{\left(\sum_{m=1}^{k} e^{z_m}\right)^2}\end{aligned}$$

Ahora, recordemos que $\sigma(z)_i = \dfrac{e^{z_i}}{\sum_{m=1}^{k} e^{z_m}}$. Podemos reescribir la expresión anterior:

$$\begin{aligned}\frac{\partial}{\partial z_i}\sigma(z)_i &= \frac{e^{z_i}}{\sum_{m=1}^{k} e^{z_m}} \cdot \frac{\sum_{m=1}^{k} e^{z_m} - e^{z_i}}{\sum_{m=1}^{k} e^{z_m}} \\ &= \sigma(z)_i \cdot \left(1 - \frac{e^{z_i}}{\sum_{m=1}^{k} e^{z_m}}\right) \\ &= \sigma(z)_i \left(1 - \sigma(z)_i\right)\end{aligned}$$

Para el caso $i = j$, la derivada de Softmax tiene exactamente la misma forma que la derivada de la función sigmoide.

Ahora, derivamos la probabilidad de la clase i con respecto al logit de una clase diferente z_j, con $i \neq j$.

$$
\begin{aligned}
\frac{\partial \sigma(z)_i}{\partial z_j} &= \frac{\partial}{\partial z_j}\left(\frac{e^{z_i}}{\sum_{m=1}^{k} e^{z_m}}\right) \\
&= \frac{\left(\frac{\partial}{\partial z_j} e^{z_i}\right) \cdot \left(\sum_{m=1}^{k} e^{z_m}\right) - e^{z_i} \cdot \left(\frac{\partial}{\partial z_j} \sum_{m=1}^{k} e^{z_m}\right)}{\left(\sum_{m=1}^{k} e^{z_m}\right)^2} \\
&= \frac{0 \cdot \left(\sum_{m=1}^{k} e^{z_m}\right) - e^{z_i} \cdot e^{z_j}}{\left(\sum_{m=1}^{k} e^{z_m}\right)^2} \\
&= -\frac{e^{z_i} e^{z_j}}{\left(\sum_{m=1}^{k} e^{z_m}\right)^2} \\
&= -\frac{e^{z_i}}{\sum_{m=1}^{k} e^{z_m}} \cdot \frac{e^{z_j}}{\sum_{m=1}^{k} e^{z_m}} \\
&= -\sigma(z)_i \cdot \sigma(z)_j
\end{aligned}
$$

Ahora, observemos con atención las derivadas parciales:
$\frac{\partial}{\partial z_j} e^{z_i} = 0$, porque z_i y z_j son variables independientes cuando $i \neq j$. La derivada de una función que no depende de z_j es cero.

$$\frac{\partial}{\partial z_j} \sum_{m=1}^{k} e^{z_m} = e^{z_j}$$

Sustituyendo:

$$
\begin{aligned}
\frac{\partial}{\partial z_j} \sigma(z)_i &= \frac{0 \cdot \left(\sum_{m=1}^{k} e^{z_m}\right) - e^{z_i} \cdot e^{z_j}}{\left(\sum_{m=1}^{k} e^{z_m}\right)^2} \\
&= -\frac{e^{z_i} e^{z_j}}{\left(\sum_{m=1}^{k} e^{z_m}\right)^2}
\end{aligned}
$$

Ahora, reescribamos este resultado en términos de las probabilidades $\sigma(z)_i$ y $\sigma(z)_j$:

$$\begin{aligned}\frac{\partial}{\partial z_j}\sigma(z)_i &= -\frac{e^{z_i}}{\sum_{m=1}^{k} e^{z_m}} \cdot \frac{e^{z_j}}{\sum_{m=1}^{k} e^{z_m}} \\ &= -\sigma(z)_i \cdot \sigma(z)_j\end{aligned}$$

De esta manera obtenemos expresiones aritméticas simples para calcular las derivadas de la función Softmax. Cabe señalar que, como lo menciona (Goodfellow et al., 2016), no todas las funciones objetivo son compatibles con esta función de activación. Específicamente, funciones que no utilizan el logaritmo —como el error cuadrático medio— suelen detener el aprendizaje cuando el argumento de la exponencial se vuelve muy negativo, provocando el problema del desvanecimiento del gradiente. En cambio, la función de entropía cruzada (*cross-entropy*) resulta particularmente adecuada para combinarse con SoftMax, ya que el logaritmo presente en la entropía cruzada simplifica las derivadas resultantes al combinarse con la exponencial de SoftMax, produciendo expresiones más estables para el cálculo del gradiente.

1.2.4 Algunos algoritmos de optimización

Como mencionamos anteriormente, el mecanismo de "aprendizaje" de una Red Neuronal Artificial consiste en minimizar la *función objetivo*. Es decir, buscamos que a través de un algoritmo, se modifiquen los parámetros (pesos sinápticos y sesgos) de cada neurona para lograr que la distancia entre el valor observado y el valor predicho por la red sea la más pequeña posible. Para lograr este objetivo, es necesario encontrar un método de optimización que nos permita modificar dichos parámetros de manera tal que, en cada modificación, los valores predichos por la red se parezcan cada vez más a los valores observados en nuestros datos de entrenamiento.

Por ello, no podemos modificar los parámetros de forma aleatoria con la esperanza de encontrar algún valor que haga que $|y_i - N_\Theta(X_i)| \to 0$, pues esto implicaría un enorme costo computacional sin la garantía de que en algún momento se alcanzarían los valores aceptables.

Sin embargo, contamos con una herramienta sumamente poderosa que nos puede indicar de manera certera hacia dónde debemos movernos si lo que buscamos es reducir la diferencia entre el valor observado y_i y el valor predicho $N_\Theta(X_i)$: el cálculo.

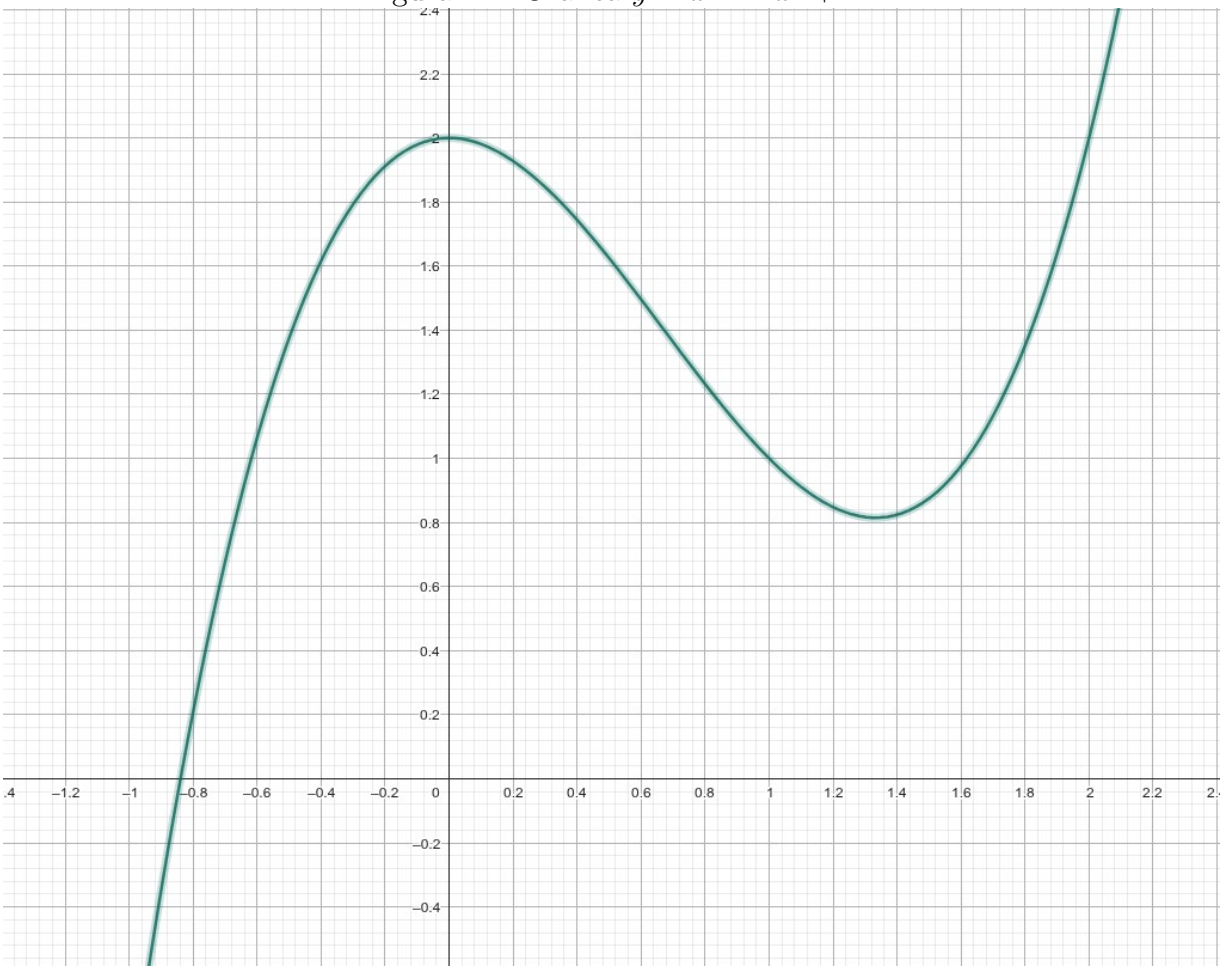
Imaginemos por un momento que nuestro conjunto de datos sólo tiene una variable. Entonces, la *función objetivo* que queremos optimizar es $y = f(x)$ con $x \in \mathbb{R}$. En este caso x sería el equivalente a nuestro vector de pesos W, que sólo tiene una entrada (es de una dimensión). En ese caso, sabemos que la derivada $f'(x)$ nos indica en algún punto

particular x si la función $f(x)$ crece o decrece. De esta manera podemos movernos un poco hacia la derecha o hacia la izquierda para lograr que $f(x)$ decrezca.

Para esclarecer un poco más cómo funciona el algoritmo denominado *descenso del gradiente* vamos a ilustrarlo con el siguiente ejemplo simple cuya función es solo explicativa y no corresponde con ningún conjunto de datos real.

Supongamos que nuestra *función objetivo* es $f(x) = x^3 - 2x^2 + 2$, cuya gráfica se presenta a continuación:

Figura 1.4: Gráfica $y = x^3 - 2x^2 + 2$



Ejemplo 1.2.7. En la gráfica anterior podemos observar que tenemos un mínimo local, sin embargo, el mínimo global no se alcanza.

Vamos a suponer que aleatoriamente nos situamos en $x = 2$. Queremos saber si en dicho punto la función es creciente o decreciente. Así que, para averiguarlo, debemos encontrar su derivada, recordando que si en $x = 2$, $f'(x) < 0$ entonces la función decrece, si $f'(x) > 0$ entonces la función crece, y si $f'(x) = 0$ estamos en un punto

crítico, donde la función no crece ni decrece, es decir, la recta tangente en ese punto es paralela al eje x.

Entonces, primero vamos a hallar la derivada:

$$\frac{d}{dx}\left(x^3 - 2x^2 + 2\right) = 3x^2 - 4x$$

Y ahora vamos a evaluar en $x = 2$, entonces $f'(2) = 3(2)^2 - 4(2) = 4 > 0$, por lo tanto sabemos que en $x = 2$ la función es creciente.

Como buscamos "ir hacia abajo" (a un punto x_i que haga que $f(x_i) < f(x)$), necesitamos movernos hacia la izquierda un paso (porque sabemos que la función es creciente en ese punto). Así que vamos a actualizar nuestro punto de partida con el siguiente algoritmo: $x_n = x_{n-1} - \alpha f'(x_{n-1})$ donde α, llamado *coeficiente de aprendizaje*, determinará el tamaño de paso, es decir, que tan rápido nos vamos a mover.

Tomemos $\alpha = 0.1$. Así, como $x_1 = 2$, entonces tenemos que:

$$x_2 = x_1 - 0.1 \cdot f'(2) = 2 - 0.1 \cdot 4 = 1.6$$

.

Evaluamos nuevamente nuestra derivada, pero ahora en $x_2 = 1.6$:

$$f'(1.6) = 3(1.6)^2 - 4(1.6) = 1.28 > 0$$

Por lo tanto, nuestra función sigue siendo creciente, y debemos movernos una vez más un paso hacia la izquierda. Sin embargo, notemos que

$$f(2) = 2^3 - 2 \cdot 2^2 + 2 = 2 > f(1.6) = 1.6^3 - 2 \cdot 1.6^2 + 2 = 0.976$$

Y sabemos que vamos en la dirección correcta, porque nos dirigimos hacia valores de x que hacen que $f(x)$ sea más pequeña.

Vamos a retroceder un paso más: $x_3 = x_2 - 0.1 f'(x_2) = 1.6 - 0.1 \cdot 1.28 = 1.472$

Ahora, evaluamos $f'(x_3) = f'(1.472) = 3(1.472)^2 - 4(1.472) \approx 0.6123 > 0$. Una vez más, sabemos que la función $f(x)$ es creciente en $x_3 = 1.472$, pero notemos que $f(1.472) \approx 0.8559$, por lo tanto $f(x_3) < f(x_2) < f(x_1)$, así que seguimos moviéndonos en dirección correcta.

Avancemos un par de pasos más: $x_4 \approx 1.4107$

$$f'(1.4107) = 3(1.4107)^2 - 4(1.4107) \approx 0.4618 > 0$$

y $x_5 \approx 1.3645$

$$f'(1.3645) = 3(1.3645)^2 - 4(1.3645) \approx 0.1275 > 0$$

Veamos que $f(x_5) = f(1.3645) = 1.3645^3 - 2(1.3645)^2 + 2 = 0.8167$, pero en x_5 la pendiente ya se acerca a cero. Por lo que vamos llegando hacia el mínimo local. Es decir, el algoritmo realmente está minimizando la función y va alcanzando la convergencia.

En este ejemplo, el algoritmo converge hacia el mínimo local porque la función es relativamente simple y la tasa de aprendizaje utilizada es adecuada. Sin embargo, en funciones más complejas (como las que aparecen en redes profundas), el comportamiento puede ser diferente. Dependiendo del punto inicial y de la tasa de aprendizaje, el algoritmo puede converger lentamente, estancarse en regiones planas o aproximarse a puntos de silla. A pesar de ello, en la práctica suele encontrar soluciones suficientemente buenas.

Si bien es cierto que podríamos encontrar los puntos críticos igualando la derivada a cero, y con ello podríamos encontrar el punto exacto donde se encuentran los mínimos locales, cuando pasamos a más dimensiones esto no es tan evidente, ni es fácil de obtener. Por otro lado, el algoritmo está diseñado para moverse en dirección de decrecimiento de la función objetivo, por lo que frecuentemente se aproxima a un mínimo local.

La tasa de aprendizaje α, que determina el tamaño del paso, puede ser cualquier número. Sin embargo, un número muy pequeño haría que nuestros pasos fueran igualmente muy pequeños, retardando la convergencia (como lo mencionamos en las secciones anteriores). Pero un número muy grande podría hacer que pasáramos de un lado al otro todo el tiempo, haciendo que la convergencia fuera igual de lenta. Veámoslo con un ejemplo numérico, esta vez mi tasa de aprendizaje será $\alpha = 2$, así: si comienzo en $x_1 = 2$, tendré $x_2 = x_1 - 2f'(x_1) = 2 - 2 \cdot 4 = -6$

Entonces:

$$f'(x_2) = 3(-6)^2 - 4(-6) = 132 > 0$$

En este caso, en la gráfica podemos ver que si $x = -6$, la función es creciente en ese punto, pero si nos alejamos más hacia la izquierda, el algoritmo continúa desplazándose hacia valores cada vez más negativos de x, alejándose de la región donde se encuentra el mínimo local.

Ahora tomemos un caso menos extremo con nuestra tasa de aprendizaje 0.4, así $x_2 = 2 - 0.4 \cdot 4 = 0.4$ y ahora:

$$f'(x_2) = 3(0.4)^2 - 4(0.4) = -1.12 < 0$$

Entonces, como la pendiente es negativa y nuestra función decrece, debemos movernos hacia la derecha, pero notemos que el algoritmo ya se encarga de dirigirnos adecuadamente, pues:

$$x_3 = x_2 - \alpha f'(x_2) = 0.4 - (0.4) \cdot (-1.12) = 0.4 + 0.448 = 0.848$$

Lo anterior se debe a que mi derivada $f(x_2) < 0$, y con el signo negativo del algoritmo $x_3 = x_2 - \alpha f'(x_2)$ en vez de una resta que hace más pequeño nuestro nuevo valor de x, tenemos una suma que lo vuelve un poco más grande. En otras palabras, el algoritmo nos lleva en la dirección opuesta hacia donde la función crece, es decir, nos lleva siempre hacia donde la función decrece.

El ejemplo anterior es un intento de representar de manera gráfica y numérica el algoritmo de optimización de descenso de gradiente, una generalización para funciones en $\mathbb{R}^n$, que desarrollaremos a continuación.

Antes de continuar, hay algo que debemos tomar en consideración. En el ejemplo, inicié aleatoriamente en $x_1 = 2$. Si observamos la gráfica, podemos ver con facilidad que si comienzo en cualquier punto aleatorio $x > 0$, invariablemente voy a alcanzar el mínimo local con este algoritmo. Sin embargo, si comenzamos suficientemente lejos a la izquierda, entonces el algoritmo no alcanzaría la convergencia; de hecho, comenzaría a diverger muy rápido, incluso si mi tasa de aprendizaje es baja. Operativamente, una Red Neuronal Artificial inicia sus pesos siempre de manera pseudoaleatoria, y es el algoritmo el encargado de dirigir el proceso de optimización hacia el mínimo local más cercano al punto de inicio. Obviamente, si cambiamos el punto de inicio, cambiarán también las direcciones de movimiento, y podríamos, incluso, converger a un mínimo local diferente.

Método por Descenso de gradiente

El ejemplo anterior fue una presentación gráfica del algoritmo de optimización de descenso de gradiente para funciones de una variable. Lo que hemos ilustrado se generaliza de manera natural a funciones de múltiples variables. En el caso de una red neuronal, la función objetivo $J(\Theta)$ depende de un vector de parámetros $\Theta = [w_1, w_2, ..., w_n, b_1, b_2, ...]$. La derivada $f'(x)$ se generaliza al gradiente $\nabla J(\Theta)$, un vector que contiene las derivadas parciales con respecto a cada parámetro. La dirección de máximo decrecimiento ya no es simplemente "izquierda" o "derecha", sino la dirección opuesta al vector gradiente: $-\nabla J(\Theta)$.

El gradiente es un vector que contiene las derivadas parciales de J con respecto de cada uno de sus parámetros Θ:

$$\nabla J(\Theta) = [\frac{\partial J}{\partial w_1}, \frac{\partial J}{\partial w_2}, \frac{\partial J}{\partial w_3}, \ldots, \frac{\partial J}{\partial w_n}, \frac{\partial J}{\partial b_1}, \ldots, \frac{\partial J}{\partial b_m}]$$

El algoritmo del descenso del gradiente entonces se expresa como:

$$\Theta_{nuevo} = \Theta_{actual} - \alpha \nabla J(\Theta_{actual})$$

En la sección 1.2.2 vimos cómo hallar $\frac{\partial J}{\partial w_i}$, y de manera muy similar podemos encontrar también $\frac{\partial J}{\partial b_r}$. Sin embargo, debemos tomar en cuenta varias cosas:

En primer lugar, si nos situamos en un punto aleatorio, podemos comenzar a movernos hacia un mínimo local, pero no vamos a lograr que nuestra función objetivo sea cero, así que podemos detener el algoritmo en algún valor determinado $J(\Theta) = \epsilon$, o podemos detenerlo después de cierto número de épocas. Hay que tener en consideración que si ϵ es muy pequeño, el costo computacional para alcanzar dicho valor puede ser muy elevado, y por ello, a veces se prefiere detener el algoritmo después de n épocas; por otro lado, detener el algoritmo en cierto número de épocas podría hacer que nos detengamos en algún punto muy lejano del mínimo local al que pretendemos llegar, pero aumentar considerablemente el número de épocas, también puede repercutir en un costo computacional muy elevado.

Por otra parte, si el conjunto de datos es demasiado grande, el costo computacional también se va a elevar. Por lo anterior, en un intento de reducir el costo computacional sin sacrificar la convergencia fue lo que motivó al algoritmo de *descenso estocástico de gradiente* que veremos a continuación:

Método por Descenso estocástico de gradiente (SGD)

En el método de optimización de *descenso de gradiente*, el gradiente se calcula sobre todos los datos del conjunto de entrenamiento. Sin embargo, como vimos en el ejemplo del perceptrón (1.2.4), cada nueva actualización requiere volver a evaluar todos los ejemplos del conjunto de entrenamiento para calcular nuevamente el gradiente. En problemas de aprendizaje profundo, donde el conjunto de datos de entrenamiento puede ser del orden de millones o cientos de millones, el cálculo resulta muy costoso en cuanto a los recursos computacionales utilizados. Además, el problema del desvanecimiento del gradiente implica que los gradientes pueden llegar a ser muy pequeños, de modo que las actualizaciones realizadas sobre los parámetros son mínimas.

El método por descenso estocástico de gradiente intenta abordar esta dificultad a través de una idea simple: en lugar de calcular el gradiente exacto sobre el conjunto de datos para el entrenamiento, se estima usando un subconjunto aleatorio de los datos de entrenamiento.

Si tenemos un conjunto de datos de tamaño n, y tomamos un subconjunto de tamaño

m donde $m < n$, entonces la regla de actualización del descenso estocástico del gradiente es:

$$\Theta_{nuevo} = \Theta_{actual} - \alpha\left(\frac{1}{m}\sum_{i=1}^{m}\nabla_{\Theta}L(y_i, N_{\Theta}(X_i))\right)$$

De esta manera, en lugar de tomar el conjunto de datos completo, tomamos el promedio sobre un subconjunto aleatorio de tamaño m. Dichos subconjuntos se denominan *mini-batch* en la literatura especializada de Redes Neuronales Artificiales.

Este método presenta varias ventajas con respecto al descenso de gradiente común, la principal es que es más eficiente computacionalmente, pues trabaja sobre subconjuntos de datos más pequeños. Presenta una convergencia más rápida en términos de tiempo computacional, debido a que puede realizar muchas más actualizaciones por segundo que el gradiente tradicional, aunque cada actualización sea menos precisa, o sea, mientras el descenso de gradiente tradicional realiza una sola actualización después de recorrer todos los ejemplos de entrenamiento (una época), el descenso estocástico de gradiente realiza múltiples actualizaciones en la misma época. Esto permite que el algoritmo avance más rápidamente hacia una solución aceptable. Finalmente, el carácter ruidoso de la estimación del gradiente (debido al muestreo aleatorio) tiene un efecto beneficioso: puede ayudar al algoritmo a "saltar" mínimos locales poco profundos y continuar hacia regiones de menor error. Esta propiedad contrasta con el descenso de gradiente tradicional que puede quedar atrapado en mínimos locales o regiones planas de la función objetivo.

Ahora, debemos tomar en cuenta que al utilizar un conjunto aleatorio de datos para calcular el descenso estocástico de gradiente, podemos encontrar algunas dificultades. La primera de ellas es que la trayectoria será un tanto errática (de ahí que se considere estocástica), pues no tenemos los datos completos, por lo que el cálculo puede oscilar debido al ruido generado por los datos faltantes. Además, la dirección de descenso utilizada en cada iteración es una estimación del gradiente verdadero, por lo que la trayectoria presenta oscilaciones y la convergencia suele ser menos estable que en el descenso de gradiente tradicional.

Una consideración fundamental para el correcto funcionamiento del descenso estocástico de gradiente es que los subconjuntos deben seleccionarse de manera aleatoria en cada época. La ley de los grandes números sugiere que cuando el conjunto es muy grande y los subconjuntos se seleccionan aleatoriamente, sus propiedades estadísticas tienden a aproximarse a las del conjunto completo. Si se elimina la aleatoriedad (por ejemplo, tomando siempre los mismos ejemplos o siguiendo un orden fijo), la estimación del gradiente podría sesgarse, afectando negativamente la convergencia y la capacidad de generalización del modelo.

Aunque el método de descenso estocástico de gradiente es uno de los más utilizados

en la implementación de Redes Neuronales Artificiales con grandes volúmenes de datos, en nuestro caso particular —donde trabajamos con conjuntos de datos más reducidos— el descenso de gradiente tradicional puede ser suficiente. No obstante, el estudio del descenso estocástico de gradiente es relevante porque sienta las bases para entender variantes más avanzadas y porque el conocimiento de sus ventajas y limitaciones es fundamental en cualquier aplicación de aprendizaje profundo.

1.2.5 Arquitecturas para el Procesamiento del Lenguaje Natural

Antes de profundizar en el funcionamiento detallado de la arquitectura *Long-Short Term Memory* (LSTM), es necesario contextualizar el tipo de arquitecturas que se utilizan para procesar datos secuenciales (como el lenguaje), así como las alternativas existentes y las razones que motivaron la elección de LSTM en este trabajo.

Redes Neuronales Recurrentes (RNN)

Las Redes Neuronales Recurrentes (RNN, por sus siglas en inglés) son una familia de arquitecturas diseñadas para trabajar con datos que presentan una estructura secuencial, como series de tiempo, audio, texto o cualquier otra variable donde el orden de los elementos es relevante. A diferencia de las redes feedforward (como el perceptrón), donde cada entrada se procesa de manera independiente, las RNN incorporan una conexión recurrente que permite que la información se transmita de un paso temporal al siguiente.

Formalmente, una RNN procesa una secuencia de vectores $x_1, x_2, \ldots, x_T$ manteniendo un estado oculto h_t que se actualiza en cada paso mediante la recurrencia:

$$h_t = \tanh(W_{xh}x_t + W_{hh}h_{t-1} + b_h)$$

Este estado oculto h_t resume la información de la secuencia hasta el instante t, y puede utilizarse para predecir una salida en cada paso o al final de la secuencia.

Sin embargo, las RNN clásicas presentan una limitación fundamental: el problema del desvanecimiento del gradiente (*vanishing gradient*). Como se explicó en la sección anterior, al propagar el error a través de muchos pasos temporales, el gradiente se multiplica repetidamente por los mismos pesos, lo que provoca que su magnitud disminuya exponencialmente. En la práctica, esto impide que la red aprenda eficazmente dependencias de largo alcance, incluso cuando éstas se encuentran separadas por decenas de pasos temporales (Hochreiter & Schmidhuber, 1997). Para problemas como la atribución de autoría, donde las marcas estilísticas pueden aparecer separadas por cientos de palabras, es necesario mitigar esta limitación.

Otras arquitecturas para secuencias

Con el objetivo de superar las limitaciones de las RNN clásicas, se han desarrollado diversas arquitecturas. Entre las más relevantes se encuentran las siguientes.

La *GRU (Gated Recurrent Unit)* es una variante simplificada de LSTM que combina las compuertas de olvido y entrada en una única compuerta de actualización. Reduce el número de parámetros y es computacionalmente más eficiente, aunque en muchos casos su rendimiento es comparable al de LSTM.

Por otra parte, los llamados *Transformers* prescinden por completo de la recurrencia y utilizan un mecanismo de atención para capturar dependencias a larga distancia. Actualmente esta arquitectura es la más utilizada en procesamiento de lenguaje natural, pero requiere grandes volúmenes de datos y recursos computacionales considerables (múltiples GPUs, grandes cantidades de memoria y tiempo de entrenamiento), además de que su entrenamiento eficiente suele beneficiarse de hardware especializado (GPUs o TPUs), y gran parte del ecosistema de aprendizaje profundo se encuentra optimizado para arquitecturas compatibles con CUDA.

Finalmente, las *Bidirectional RNN / LSTM* extienden las arquitecturas unidireccionales añadiendo una capa recurrente que procesa la secuencia en sentido inverso, permitiendo que el contexto futuro influya en la representación de cada paso. Son muy útiles cuando se dispone de la secuencia completa, como en clasificación de textos.

Justificación de la elección de LSTM

Para el problema de atribución de autoría que nos ocupa, se optó por la arquitectura LSTM por varias razones.

En primer lugar, su capacidad para capturar dependencias a largo plazo. A diferencia de las RNN clásicas, LSTM está diseñada específicamente para retener información durante largos intervalos de tiempo gracias a su estado de célula y sus compuertas. Esto es esencial para detectar patrones estilísticos que pueden manifestarse a lo largo de oraciones completas o incluso párrafos.

En segundo lugar, es una excelente opción de balance entre rendimiento y recursos. Si bien los Transformers ofrecen un rendimiento superior en tareas de PLN con grandes conjuntos de datos, su implementación eficiente requiere hardware especializado (GPUs de alta gama con gran memoria) y conjuntos de datos masivos, siendo esta su principal limitación para nuestro proyecto. En nuestro caso, se dispone de un conjunto de datos limitado (251 cuentos) y se busca que el modelo pueda ejecutarse en entornos con recursos moderados, como la versión gratuita de Google Colaboratory.

En tercer lugar, las LSTM cuentan con más de dos décadas de desarrollo; existe abundante documentación, implementaciones probadas en múltiples frameworks (TensorFlow,

PyTorch, Keras) y una vasta experiencia reportada en la literatura. Esto facilita la depuración, el ajuste de hiperparámetros y la reproducibilidad de los resultados.

También se consideraron los resultados previos obtenidos con las compuertas LSTM en atribución de autoría de textos. Trabajos como los de (Gagala, 2018) y (Zhao et al., 2025) han empleado LSTM con éxito en tareas similares, demostrando su idoneidad para este dominio.

Finalmente, como se detallará en la siguiente sección, la arquitectura LSTM mitiga el problema del gradiente desvaneciente mediante las compuertas de olvido, entrada y salida, así como la suma directa del estado de la célula. Esto permite entrenar redes más profundas y con secuencias más largas sin que el aprendizaje se estanque.

Por todas estas razones, la LSTM se erige como la opción más equilibrada para este trabajo: combina la potencia necesaria para capturar el estilo literario con requerimientos computacionales asequibles, y cuenta con un respaldo teórico y práctico sólido. En la siguiente subsección se explica en detalle su funcionamiento interno.

1.2.6 La arquitectura LSTM (Long Short-Term Memory)

Hasta el momento hemos visto que el perceptrón es un tipo de neurona artificial que intenta emular a la neurona biológica. La combinación de numerosas neuronas artificiales en una red hace que un programa computacional sea capaz de clasificar datos a partir de un conjunto de entrenamiento. En otras palabras, una Red Neuronal Artificial es capaz de encontrar patrones y generalizar comportamientos a partir de los datos mismos. En los ejemplos que hemos planteado hasta el momento, presentamos conjuntos de datos, y asociamos dichos datos a una clase.

El ejemplo del perceptrón plantea que una Red Neuronal Artificial compuesta por perceptrones sería capaz de clasificar si un individuo pertenece a la clase social alta o baja dependiendo de la cantidad de focos que hay en su casa, el número de vehículos con los que cuenta la familia, el material del que está hecho el piso y el número de habitantes que hay en la casa. El resultado es una combinación lineal de estos elementos. Pero debemos notar que, aunque puedan estar relacionados, cada elemento es independiente del otro. Es decir, el número de vehículos no depende del material del piso. El número de habitantes no depende de la cantidad de focos.

Ahora, imaginemos un nuevo contexto. Pensemos en estos juegos de Rol, tipo Calabozos y Dragones, en donde tomar una decisión nos lleva a un resultado, pero que depende específicamente de la decisión previa. Seamos más concretos: tenemos un juego en el que podemos elegir tres clases: enanos, elfos y humanos. Posteriormente podremos elegir un arma, pero con las siguientes restricciones: espadas y dagas sólo pueden usarlas los humanos, las hachas y los mazos pueden ser elegidas por los enanos, los arcos y las lanzas

pueden elegirlas los elfos. Entonces, la posibilidad de utilizar en el juego un arco, depende completamente de que previamente hayamos seleccionado a un elfo. En este caso, los datos no son independientes, y la clasificación tendría que considerar no sólo a los datos, sino la dependencia que existe entre ellos.

El problema que nos atañe, o sea, la clasificación de textos, tiene justamente esa característica. El lenguaje es un sistema de signos que dependen de contextos, es decir, en una oración el significado global no es la suma de los significados de cada palabra, sino el contexto que relaciona una palabra con otra, amén de varios factores más que afectan la semántica. En otras palabras, un mensaje escrito no sólo depende de las palabras de la oración, sino del contexto de estas palabras. En las oraciones: "Las **mariposas** monarcas viajan cientos de kilómetros para reproducirse." y "Siento **mariposas** en el estómago.", la palabra resaltada en negritas es la misma, pero en contexto se refiere a cosas distintas. En la primera oración hablamos de unos insectos de colores negro y amarillo que habitan en América, en la segunda oración se refiere al acto de estar enamorado, o cuando menos, nervioso.

En el procesamiento de secuencias —como la clasificación de textos, donde cada palabra depende de las anteriores para construir el sentido— las redes neuronales recurrentes clásicas (RNN) presentan una limitación fundamental: el problema del desvanecimiento del gradiente.

Como vimos en secciones anteriores, el algoritmo de retropropagación calcula cómo debe modificarse cada peso para reducir el error. En una Red Neuronal Artificial Recurrente clásica, este cálculo implica multiplicar repetidamente los mismos valores a lo largo del tiempo. Cuando la secuencia es larga (como un texto completo), estas multiplicaciones pueden hacer que el gradiente se vuelva extremadamente pequeño, hasta el punto de que las actualizaciones de los pesos se vuelven insignificantes.

En nuestro problema de detección de autoría, esta limitación es especialmente grave. Consideremos un texto de Jorge Luis Borges. Su estilo se caracteriza por ciertas asociaciones léxicas que pueden aparecer separadas por muchas palabras —por ejemplo, "jardín" y "laberinto" pueden estar separados por varias oraciones. Para identificar correctamente al autor, la red necesita "recordar" que apareció la palabra "jardín" muchas palabras atrás cuando finalmente encuentre la palabra "laberinto". Una Red Neuronal Recurrente clásica perdería esta información en el camino. La solución que emplearemos en este trabajo es la arquitectura LSTM.

La LSTM (Long Short-Term Memory), introducida por Hochreiter y Schmidhuber en 1997, fue diseñada específicamente para superar esta limitación. Su nombre, que podríamos traducir como "memoria a largo plazo", refleja precisamente su capacidad para retener información durante largos intervalos de tiempo.

El elemento clave de la LSTM es la incorporación de un estado de célula (cell state), que

actúa como una "cinta transportadora" de información a lo largo de toda la secuencia. A diferencia de las RNN clásicas, donde la información se transforma en cada paso mediante multiplicaciones que pueden hacerla desaparecer, en la LSTM la información puede fluir a través del tiempo prácticamente sin alteraciones, gracias a un mecanismo de compuertas (gates) que decide qué información conservar, qué información olvidar y qué información agregar.

Estas compuertas son:

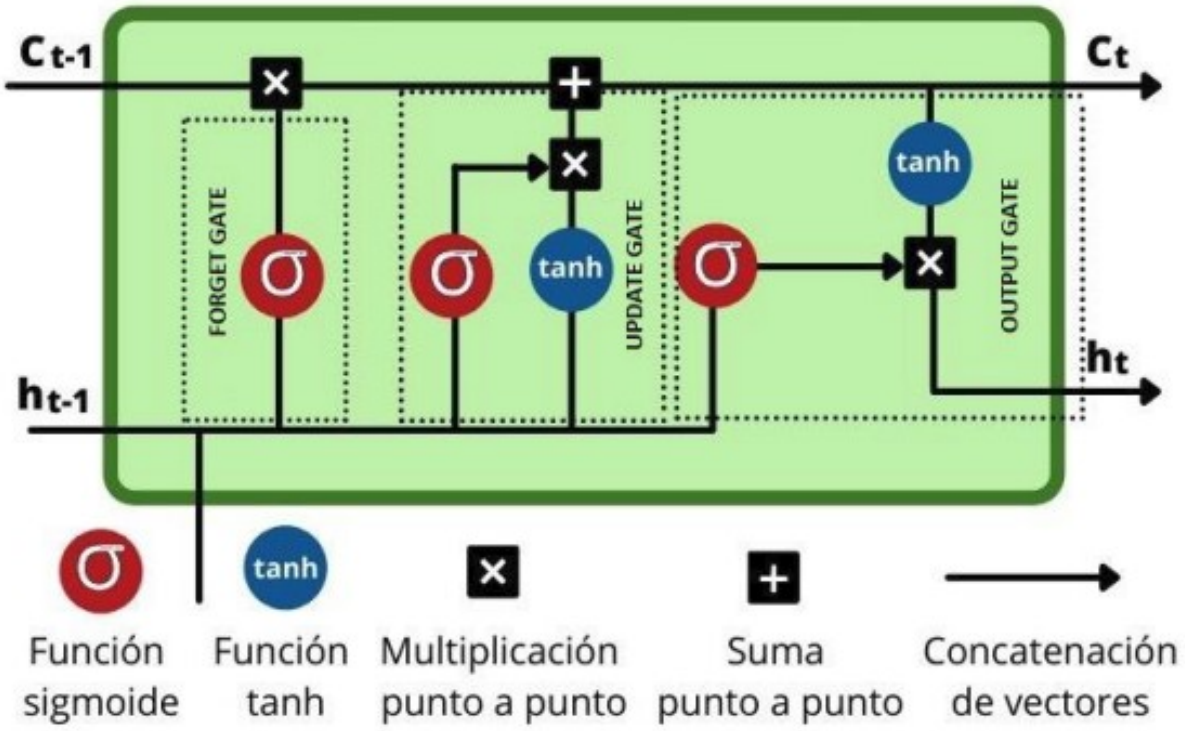
Compuerta de olvido (forget gate): decide qué información del estado anterior debe descartarse.

Compuerta de entrada (input gate): decide qué nueva información debe almacenarse.

Compuerta de salida (output gate): decide qué información debe emitirse como salida en cada paso.

La siguiente imagen muestra el funcionamiento básico de la arquitectura LSTM

Figura 1.5: Arquitectura LSTM tomada de Vega, I. *Predicción de los Niveles de Glucosa en Sangre para Pacientes Diabéticos a Partir de Redes Neuronales Recurrentes tipo LSTM*



En la imagen podemos ver cómo fluye la información a través de la arquitectura LSTM. Para entender su funcionamiento, examinemos cada uno de sus componentes. Utilizaremos la siguiente notación:

- x_t: entrada en el tiempo t (por ejemplo, la representación vectorial de una palabra).

- h_{t-1}: estado oculto anterior (la "memoria a corto plazo").
- C_{t-1}: estado de la célula anterior (la "memoria a largo plazo").
- h_t: nuevo estado oculto.
- C_t: nuevo estado de la célula.
- σ: función sigmoide (que produce valores entre 0 y 1, actuando como una compuerta).
- tanh: función tangente hiperbólica (que produce valores entre -1 y 1).

La compuerta de olvido decide qué información del estado de la célula anterior debemos descartar. Se calcula como:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

El resultado f_t es un vector de valores entre 0 y 1. Un valor cercano a 0 indica "olvida completamente", mientras que un valor cercano a 1 indica "conserva completamente". Mientras que $[h_{t-1}, x_t]$ denota la concatenación de ambos vectores.

En nuestro problema de detección de autoría, esta compuerta podría decidir olvidar información sobre la estructura de una oración que ya terminó, para no confundirla con la siguiente.

La compuerta de entrada decide qué nueva información vamos a almacenar en el estado de la célula. Tiene dos partes:

Una compuerta sigmoide que decide qué valores actualizar:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

y una transformación mediante la función de tangente hiperbólica que genera un vector de valores candidatos:

$$\tilde{C}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$$

El nuevo estado de la célula se actualiza combinando la información que decidimos conservar del estado anterior (multiplicando C_{t-1} por f_t) con la nueva información que decidimos agregar (multiplicando $\tilde{C}_t$ por i_t):

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$$

donde $\odot$ denota el producto elemento a elemento (Hadamard).

Esta actualización es clave: al sumar directamente la información anterior con la nueva, el gradiente puede fluir a través del tiempo sin desvanecerse, resolviendo el problema que afecta a las RNN clásicas.

En nuestro ejemplo, esta compuerta podría decidir almacenar información sobre una palabra clave que indica el estilo de un autor (por ejemplo, el uso de ciertos verbos o construcciones).

Finalmente, la compuerta de salida decide qué información del estado de la célula se utilizará como salida (el nuevo estado oculto h_t):

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

$$h_t = o_t \odot \tanh(C_t)$$

El estado oculto h_t es lo que la red LSTM entrega a la siguiente capa (o a la capa final de clasificación). Contiene información filtrada del estado de la célula, y será la base para la predicción final.

Para nuestro caso particular, donde deseamos que la red sea capaz de clasificar un texto determinado en tres clases distintas (los autores), la idea, en términos simples, es la siguiente: cada palabra del texto se representa a través de un vector denso. Todo el texto deberá pasar a través de la arquitectura LSTM, pero lo hará de manera secuencial, palabra por palabra.

Al inicio del procesamiento, el estado de la célula C_0 y el estado oculto h_0 se inicializan como vectores nulos, pues aún no se ha procesado ninguna palabra. Cuando ingresa la primera palabra x_1, la LSTM combina este vector nulo con x_1 para producir los nuevos estados C_1 y h_1, que contendrán información de esta primera palabra.

En el siguiente paso, la compuerta toma la segunda palabra x_2. Esta vez C_1 ya no es un vector nulo, pues ya contiene la información de x_1. La nueva información pasa por la compuerta de olvido, la compuerta de entrada y la compuerta de salida. Las operaciones correspondientes hacen que C_2 tome un nuevo valor que resulta de la combinación de las palabras x_1 y x_2.

Este proceso se repite hasta completar la totalidad del texto. Si el texto contiene n palabras, después de que cada una de las n palabras haya pasado por las tres compuertas de la LSTM, el estado final h_n contendrá una representación vectorial de todo el texto. Este vector es el que se utiliza como entrada para la capa de clasificación que determina el autor.

Ejemplo 1.2.8. Para ilustrar el funcionamiento interno de una LSTM, consideremos

un ejemplo numérico simplificado con vectores de 2 dimensiones.

Configuración inicial:

$$h_0 = [0, 0] \qquad C_0 = [0, 0]$$

Supongamos que la primera palabra "El" tiene embedding $x_1 = [0.3, 0.5]$.

Paso 1: Compuerta de olvido

$$f_1 = \sigma(W_f \cdot [h_0, x_1] + b_f)$$

Supongamos que el resultado es $f_1 = [1, 1]$ (no olvidamos nada al inicio).

Paso 2: Compuerta de entrada

$$i_1 = \sigma(W_i \cdot [h_0, x_1] + b_i) = [0.8, 0.9]$$

$$\tilde{C}_1 = \tanh(W_c \cdot [h_0, x_1] + b_c) = [0.3, 0.5]$$

Paso 3: Actualización del estado de célula

$$C_1 = f_1 \odot C_0 + i_1 \odot \tilde{C}_1 = [0, 0] + [0.8 \cdot 0.3, 0.9 \cdot 0.5] = [0.24, 0.45]$$

Paso 4: Compuerta de salida

$$o_1 = \sigma(W_o \cdot [h_0, x_1] + b_o) = [0.9, 0.9]$$

$$h_1 = o_1 \odot \tanh(C_1) = [0.9, 0.9] \odot \tanh([0.24, 0.45]) \approx [0.9 \cdot 0.23, 0.9 \cdot 0.42] = [0.21, 0.38]$$

Resultado: Después de procesar la primera palabra, el estado oculto es $h_1 \approx [0.21, 0.38]$.

Este proceso se repite para cada palabra del texto. Al final, el último estado h_n resume el texto completo y se utiliza para la clasificación.

Ahora que hemos revisado cómo funciona matemáticamente una Red Neuronal Artificial con compuertas LSTM, es el momento de implementar un modelo computacional de atribución de autores. Por ello, en el siguiente capítulo, revisaremos los pormenores de la implementación de dicho modelo.

2 Implementación de la red neuronal

2.1 Obtención de los datos y su procesamiento

Obtener los datos para el entrenamiento de una red neuronal no es un proceso trivial. De hecho, una de las mayores dificultades a las que me enfrenté al momento de realizar este trabajo fue justamente el acceder a un conjunto de datos suficientemente grande que me permitiera entrenar el modelo.

El conjunto de datos necesario debía consistir en textos de varios autores, escritos en español, de extensión breve y con un catálogo suficientemente amplio.

La necesidad de que se entrene a la red con varios autores responde a que el modelo busca determinar el estilo sintáctico de cada autor, y de esta manera determinar la probabilidad de que un texto haya sido escrito por un autor particular. Si se utilizara únicamente un autor, entonces el modelo podría decir si un texto fue escrito o no por ese autor. Pero el objetivo planteado fue un poco más ambicioso. Así que al final se determinó que se trabajaría con la obra de tres escritores distintos para que el modelo pudiera predecir quién de ellos es el autor de un texto cuya autoría se desconoce. Lo cierto es que la capacidad de determinar la autoría de un texto desconocido estará limitada por el conjunto de datos de entrenamiento; entre mayor sea el conjunto, el modelo será más robusto.

La decisión de que los textos utilizados en el conjunto de datos fuera en español fue un tanto arbitraria, y respondió más a una cuestión personal que a una dificultad técnica. El modelo puede trabajar con la misma eficiencia en cualquiera lengua. Sin embargo, lo que sí es un requisito técnico es que los tres autores produzcan sus obras en la misma lengua; de lo contrario, tendríamos un sesgo en la elección de los datos. Supongamos, por ejemplo, que se eligieran las obras de William Shakespeare, de Miguel de Cervantes y de Víctor Hugo para entrenar el modelo. Como estos textos están en inglés, español y francés respectivamente, sólo es necesario identificar el idioma de un texto no etiquetado para "adivinar" quién es el autor, sin necesidad de buscar marcas estilísticas; por ello, se vuelve indispensable uniformar el idioma utilizado para el conjunto de datos de entrenamiento.

Por lo anterior, también es necesario utilizar autores cuya distancia temporal no sea demasiado lejana, ya que esto también podría provocar un sobre ajuste del modelo. Su-

pongamos que todos los textos se eligen en español, pero se toma *El poema del Mio Cid*, *El Ingenioso Hidalgo don Quijote de la Mancha* y *Cien años de Soledad*, de autor anónimo, Miguel de Cervantes y Gabriel García Márquez, respectivamente. En este caso, el sobre ajuste se generará por las variantes diacrónicas de la lengua; entonces, cuando a este modelo se le diera cualquier texto escrito en la época contemporánea, lo identificaría como de Gabriel García Márquez, mientras que cualquier texto escrito en español renacentista se lo atribuiría a Miguel de Cervantes, y toda obra escrita en español antiguo la catalogaría como autor anónimo.

El criterio de extensión breve responde a una dificultad técnica. Si se utilizan textos demasiado largos, los vectores de entrada con los que se entrenará la red serían muy grandes, lo que implica un costo computacional bastante considerable al momento de procesarlos.

Finalmente, la necesidad de que cada autor cuente con un amplio catálogo de textos se debe a que el modelo necesita un conjunto de datos muy grande para poder entrenarse adecuadamente.

Recordemos que el modelo predice la probabilidad de que un texto haya sido escrito por alguno de los tres autores que se utilizaron para su entrenamiento. Por ello, para evitar que las probabilidades queden desbalanceadas, es necesario que la cantidad de textos de cada autor utilizados para el entrenamiento sea más o menos homogénea. Imaginemos que las cantidades no son homogéneas, y que tenemos, por ejemplo, 28 textos del primer autor, 70 del segundo y 300 del tercero; en este caso, en el modelo puede darse un sobre ajuste debido a la gran diferencia que existe entre los textos del tercer autor y de los otros dos. Lo que sucederá es que el modelo aprenderá los patrones de escritura del tercer autor y tratará de encontrar dichos patrones en cualquier otro texto del que quiera predecirse su autoría, dando como resultado un sesgo que favorecerá siempre al tercer autor.

La última dificultad que tuvo que solventarse fue la de obtener los datos digitalizados para su procesamiento. Si bien es cierto que el catalogo de literatura en español es muy amplio, también es cierto que digitalizar la obra completa de algún autor que no se encuentre previamente digitalizada es una tarea muy compleja.

Por todas las cuestiones anteriores, se eligió a tres cuentistas latinoamericanos más o menos contemporáneos. El cuento representa una ventaja sobre otro tipo de textos, como la novela, la crónica o el ensayo, debido a la brevedad típica de este género literario. Además, en la red puede accederse fácilmente a un gran acervo de cuentos. Para ello, elegí la biblioteca digital de Luis López Nieves, quien se ha encargado de recopilar una gran colección de cuentos y poemas de autores clásicos de la literatura universal. La biblioteca digital puede consultarse en la página de ciudadseva.com (López Nieves, 1996).

Los autores elegidos para entrenar el modelo fueron: Jorge Luis Borges, Mario Benedetti y Julio Cortázar. Esta decisión no responde únicamente a un gusto personal, sino a

que fueron los autores con mayor número de cuentos en la biblioteca consultada, además de que el número de cuentos de cada autor es más o menos homogéneo pues se extrajeron 82 cuentos de Borges, 87 de Benedetti y 80 de Cortázar.

Para acceder a los cuentos, fue necesario implementar un código de Python haciendo uso de la librería *BeautifulSoup* que permite a nuestro programa solicitar información a una página de internet. En la biblioteca de Ciudad Seva (López Nieves, 1996), los cuentos y poemas están organizados por autores en diferentes vínculos. Al acceder al vínculo de un autor particular, nos conduce a otros dos vínculos; uno nos dirige al acervo de cuentos del autor y el otro al acervo de poemas. Al seleccionar el vínculo que nos dirige al acervo de cuentos, encontramos una nueva lista de vínculos, donde cada uno de ellos nos llevará hacia el cuento buscado. El código automatiza el proceso de entrar vínculo por vínculo para acceder a todos los cuentos de cada autor.

Para poder utilizar los cuentos, el contenido de éstos se almacenará en un arreglo de tipo lista llamado *text*; cada entrada de este arreglo será un cuento y se almacenará como una cadena de caracteres. Posteriormente, se generará otro arreglo de tipo lista llamado *labels* que almacenará una etiqueta numérica de cada autor. De esta forma, tendremos dos arreglos ordenados: el primero contiene los cuentos de los tres autores; el segundo contiene elnúmero de la etiqueta del autor de cada cuento. Las etiquetas asignadas a cada autor fueron las siguientes: Borges fue etiquetado con el número 0, Benedetti con el 1 y Cortázar con el 2. Finalmente, se creará un tercer arreglo de tipo diccionario denominado *label_map* que asignará el nombre de cada autor como cadena de caracteres a cada etiqueta numérica de la lista *label*.

El código utilizado para extraer los datos es el siguiente:

```
#Primero extraemos los textos.
import requests
from bs4 import BeautifulSoup

# URLs de las páginas de los autores en Ciudad Seva
urls = {
    "Jorge Luis Borges": "https://ciudadseva.com/autor/jorge-luis-borges/cuentos/",
    "Mario Benedetti": "https://ciudadseva.com/autor/mario-benedetti/cuentos/",
    "Julio Cortázar": "https://ciudadseva.com/autor/julio-cortazar/cuentos/"
}

# Función para extraer enlaces a los cuentos
def get_story_links(url):
    try:
        response = requests.get(url)
        response.raise_for_status()  # Verificar si la solicitud fue exitosa
```

```
        soup = BeautifulSoup(response.text, "html.parser")

        # Buscar los enlaces a los cuentos
        story_links = []
        for link in soup.select("a"):  # Buscar todos los enlaces
            href = link.get("href")
            if href and "texto" in href:  # Filtrar solo enlaces a cuentos
                story_links.append(href)
        return story_links
    except Exception as e:
        print(f"Error al obtener enlaces desde {url}: {e}")
        return []

# Función para extraer el texto de un cuento
def get_story_text(url):
    try:
        response = requests.get(url)
        response.raise_for_status()  # Verificar si la solicitud fue exitosa
        soup = BeautifulSoup(response.text, "html.parser")

        # Extraer el texto del cuento
        story_text = ""
        for paragraph in soup.select("p"):  # Buscar todos los párrafos
            story_text += paragraph.get_text() + "\n"
        return story_text.strip()
    except Exception as e:
        print(f"Error al extraer el cuento {url}: {e}")
        return None

# Extraer cuentos de cada autor
texts = []
labels = []
label_map = {"Jorge Luis Borges": 0, "Mario Benedetti": 1, "Julio Cortázar":
    2}

for author, url in urls.items():
    print(f"Extrayendo cuentos de {author}...")
    story_links = get_story_links(url)
    print(f"Enlaces encontrados: {len(story_links)}")

    for link in story_links:
        story_text = get_story_text(link)
        if story_text:  # Solo añadir si se extrajo el texto correctamente
            texts.append(story_text)
            labels.append(label_map[author])
```

```
            print(f"Cuento extraído: {link}")
        else:
            print(f"Error: No se pudo extraer el cuento {link}")

# Mostrar resultados
print(f"Total de cuentos extraídos: {len(texts)}")
print(f"Etiquetas: {labels}")
```

Listing 2.1: Código para extraer los cuentos de ciudadseva.com

El funcionamiento del código anterior se ilustra en el siguiente ejemplo:

Ejemplo 2.1.1. Supongamos que tenemos los siguientes fragmentos de textos de tres autores:

De Mario Benedetti tenemos

A la izquierda del roble

No sé si alguna vez les ha pasado a ustedes
Pero el jardín botánico es un parque dormido

Incitación

En el muro quedaron los tatuajes del juego,
el tiempo me conmina, pero no me doblego,

De Jorge Luis Borges tenemos

Edipo y el enigma

Cuadrúpedo en la aurora, alto en el día
y con tres pies errando por en vano

La lluvia

Bruscamente la tarde se ha aclarado
Porque ya cae la lluvia minuciosa.

y de Julio Cortázar tenemos

Te amo por ceja

Te amo por ceja, por cabello, te debato en corredores
blanqísimos donde se juegan las fuentes de la luz

Breve amor

Con qué tersa dulzura
me levanta del lecho en que soñaba.

Necesitamos guardar los textos en un arreglo, así definimos la lista:

text = [“A la izquierda del roble No sé si alguna vez les ha pasado a ustedes Pero el jardín botánico es un parque dormido”, “Incitación En el muro quedaron los tatuajes del juego, en tiempo me conmina, pero no me doblego”, “Edipo y el enigma

Cuadrúpedo en la aurora, alto en el día y con tres pies errando por en vano", "La lluvia Bruscamente la tarde se ha aclarado Porque ya cae la lluvia minuciosa", "Te amo por ceja Te amo por ceja, por cabello, te debato en corredores blanquísimos donde se juegan las fuentes de la luz", "Breve amor Con qué tersa dulzura me levanta del lecho en que soñaba"]

El segundo arreglo contendrá las etiquetas:

labels = [0,0,1,1,2,2]

El último arreglo contiene el mapa de las etiquetas:

label_map = {"Mario Benedetti": 0, "Jorge Luis Borges": 1, "Julio Cortázar": 2}

Así quedan constituidos nuestros tres arreglos que utilizaremos para empezar a trabajar nuestro modelo de red neuronal artificial.

Nótese que la lista de etiquetas está en orden ascendente porque así fuimos acomodando los textos en la lista de textos, primero los dos textos de Benedetti, luego los dos de Borges y finalmente los dos de Cortázar. Si los cuentos se acomodan de otra manera en la lista de textos, será necesario también reordenar la lista de etiquetas. Así, si la lista de cuentos se reacomoda:

text = ["Breve amor Con qué tersa dulzura me levanta del lecho en que soñaba", "A la izquierda del roble No sé si alguna vez les ha pasado a ustedes Pero el jardín botánico es un parque dormido", "Edipo y el enigma Cuadrúpedo en la aurora, alto en el día y con tres pies errando por en vano", "La lluvia Bruscamente la tarde se ha aclarado Porque ya cae la lluvia minuciosa", "Incitación En el muro quedaron los tatuajes del juego, en tiempo me conmina, pero no me doblego", "Te amo por ceja Te amo por ceja, por cabello, te debato en corredores blanquísimos donde se juegan las fuentes de la luz"]

Entonces nuestra lista de etiquetas se modifica de la siguiente manera:

labels = [2,0,1,1,0,2]

Una vez que se tienen los datos acomodados en sus arreglos respectivos, es momento de proceder a su limpieza.

2.2 Limpieza de los datos

Aunque los cuentos tomados de ciudadseva.com (López Nieves, 1996) en general están limpios, sí hubo necesidad de hacer algunos cuantos ajustes para su manejo apropiado.

En primer lugar, hubo que quitar los encabezados y pies de los cuentos: todos los textos comienzan con la leyenda "Casa digital del escritor Luis López Nieves Casa digital del escritor Luis López Nieves Suscríbete a NotiCuento Recibe gratis un cuento clásico semanal [Minicuento - Texto completo.]" y concluyen con la leyenda "FIN Recibe gratis

un poema clásico semanal". Fue necesario eliminar estas leyendas debido a que son ruido para el modelo. Ninguna de estas leyendas ayuda a capturar los rasgos estilísticos de los autores, además, estas leyendas no fueron escritas por los autores de los textos y conservarlas implica un detrimento a la precisión de las predicciones de la red neuronal.

Sin embargo, como todos los textos incluyen estas leyendas, quitarlas de manera manual conlleva una inversión de tiempo bastante considerable. Por ello, se implementaron códigos de Python para hacer esta primera limpieza.

```python
#Vamos a escribir un código para quitar los encabezados de los cuentos.
def quitar_encabezado(textos):
    encabezado_base = "Casa digital del escritor Luis López Nieves Casa digital del escritor Luis López Nieves Suscríbete a NotiCuento Recibe gratis un cuento clásico semanal [Minicuento - Texto completo.]"
    textos_limpios = []
    for texto in textos:
        # Normaliza espacios y saltos de línea en ambos textos
        texto_normalizado = ' '.join(texto.split())
        encabezado_normalizado = ' '.join(encabezado_base.split())
        # Elimina el encabezado si está presente
        texto_limpio = texto_normalizado.replace(encabezado_normalizado, "").strip()
        textos_limpios.append(texto_limpio)
    return textos_limpios

#Aplicar la función
textos_limpios = quitar_encabezado(texts)
print(textos_limpios)
```

Listing 2.2: Código para eliminar los encabezados

```python
def quitar_pies(textos):
    pie_base = "FIN Recibe gratis un poema clásico semanal"
    textos_limpios = []
    for texto in textos:
        # Normaliza espacios y saltos de línea en ambos textos
        texto_normalizado = ' '.join(texto.split())
        pie_normalizado = ' '.join(pie_base.split())
        # Elimina el encabezado si está presente
        texto_limpio = texto_normalizado.replace(pie_normalizado, "").strip()
        textos_limpios.append(texto_limpio)
    return textos_limpios

#Aplicar la función
textos_limpios2 = quitar_pies(textos_limpios)
print(textos_limpios2)
```

Listing 2.3: Código para eliminar los pies

Después de este preprocesamiento de los datos, el arreglo denominado *textos_ limpios2* contiene únicamente el contenido de los cuentos, sin encabezados, sin pies, y sin títulos, sólo los cuentos acomodados como cadenas de caracteres.

El siguiente paso consistió en eliminar las palabras estructurales. Se denominan palabras estructurales (*stopwords* en inglés) a las palabras que sirven para dar estructura a las oraciones, pero que, en sí, tienen una carga semántica mínima o nula. Por ejemplo, en la oración "El sol se levanta todas las mañanas por el oriente", la carga semántica de la oración recae en las palabras "sol", que es el núcleo del sujeto, "levanta", que es el verbo, "mañanas" y "oriente". Por otro lado, las palabras "El", "se", "todas" y "por" están ahí para dar estructura y precisar o matizar el significado de la oración. Sin embargo, si prescindimos de ellas, el mensaje, aunque más críptico, aún puede interpretarse sin perder su significado (aunque sí se disminuye su fuerza elocutiva o se pierde claridad). La oración sin estas palabras estructurales diría "sol levanta mañanas oriente". Notemos que no es la misma oración; de hecho, esta nueva oración puede tener una interpretación diferente porque estamos modificando tanto la sintaxis como la gramática de la versión original. Sin embargo, con un esfuerzo extra de nuestra parte, es posible reconstruir el mensaje original sólo con estas palabras.

Podemos imaginar que las palabras estructurales son esas líneas blancas pintadas en la carretera que nos ayudan a no salirnos del carril, mientras que el camino en sí está constituido por las palabras no estructurales. Podemos llegar al mismo destino sin palabras estructurales, pero quizás debamos esforzarnos más para llegar en línea recta.

Sin embargo, cuando pensamos en enviar un mensaje de la manera más eficiente posible, eliminar dichas palabras estructurales puede reducir el tamaño del mensaje de manera considerable sin sacrificar su significado original. Esta es la idea que subyace detrás del procedimiento de eliminar palabras estructurales: se conserva el mismo mensaje, pero se utilizan menos palabras para expresarlo, pues quitamos todas las palabras cuya carga semántica es mínima y cuya aportación es insignificante.

Eliminar las palabras estructurales de un texto literario sí puede influir en las predicciones del modelo. Pues los autores de literatura pueden utilizar estas palabras y resignificarlas, o acomodarlas de alguna manera particular como parte de su estilo propio. Sin embargo, este tipo de prácticas es más común en la poesía que en la prosa. Para implementar el modelo, se realizaron algunas pruebas previas para determinar la pertinencia de eliminar o no las palabras estructurales; con dichas pruebas se llegó a la conclusión de que eliminarlas volvió más eficiente el modelo en cuanto a costo computacional, pues redujo el tiempo de compilación a menos de la mitad, pero no se redujo sustancialmente

la precisión al momento de probar el modelo con datos no etiquetados. Por ello, se llegó a la conclusión de que eliminarlas es la mejor opción. Todos estos resultados se presentarán con más detalle en la sección de Resultados.

No existe un consenso universal sobre cuáles son las palabras estructurales del español. Normalmente, se consideran como estructurales las preposiciones, las conjunciones, las onomatopeyas y una gran cantidad de adverbios. Sin embargo, también tienen cabida en esta lista algunos pronombres y muchos verbos de uso frecuente, además de ciertas locuciones. Para este modelo, se hizo uso de una base de datos pre-entrenada llamada *spacy*, su versión en español contiene una lista de palabras estructurales que utilizamos para nuestro modelo.

En el siguiente código se instala la librería *spacy*, se descarga la versión en español y, posteriormente, se utiliza esta base de datos para eliminar las palabras estructurales de nuestros cuentos.

```
#Instalamos la librería spacy
!python -m spacy validate

#Descargamos spacy en español
!python -m spacy download es_core_news_sm

#Ahora vamos a revisar la lista de palabras estructurales de la librería
    spaCy
import spacy
nlp = spacy.load('es_core_news_sm')
stop_words = spacy.lang.es.stop_words.STOP_WORDS
print(stop_words)

#Quitamos palabras estructurales de nuestros textos
def eliminar_palabras_estructurales(lista_textos):
    textos_procesados = []
    for texto in lista_textos:
        doc = nlp(texto)
        palabras_filtradas = [token.text for token in doc if not token.
    is_stop and not token.is_punct]
        textos_procesados.append(" ".join(palabras_filtradas))
    return textos_procesados

# Aplicar la función a la lista completa
textos_sin_stopwords = eliminar_palabras_estructurales(textos_limpios2)
print("Texto original:", textos_limpios2)
print("Texto sin stop words:", textos_sin_stopwords)
```

Listing 2.4: Código para eliminar las palabras estructurales

Una vez que se tienen los datos como un arreglo de cadenas de caracteres llamado *texto_ sin_ stopwords* el siguiente paso es convertir estos arreglos en vectores que puedan ser leídos por la red neuronal artificial.

2.2.1 *One-hot Encoding*

Hasta el momento, el vector de etiquetas de nuestros autores es *labels = [0,1,2]* que corresponde con que Borges está etiquetado como 0, Benedetti como 1 y Cortázar como 2. Estas etiquetas constituyen una forma de distinguir las tres categorías (autores) que estamos definiendo; su función no es numérica (ordinal) sino categórica (nominal). Sin embargo, al crear el vector de etiquetas de esta manera, existe el riesgo de que la red neuronal interprete que Borges $<$ Benedetti $<$ Cortázar, debido a que $0 < 1 < 2$, pues "asignar valores numéricos secuenciales implica una relación ordinal inexistente entre clases categóricas" (James et al., 2023). Para resolver este problema se recurre al *one-hot encoding*.

Definición 2.2.1. *one-hot encoding* es una técnica de representación de variables categóricas donde cada categoría se mapea a un vector binario de longitud igual al número de categorías posibles, con un solo 1 en la posición correspondiente a la categoría y un 0 en todas las demás posiciones.

Dado un conjunto Ω de tamaño $|\Omega| = n$, cada elemento $\omega_i \in \Omega$ se representa como:

$$\omega_i = [0, 0, 0, ..., 1, ..., 0]$$

donde el 1 aparece en la i-ésima posición.

Una de las ventajas de utilizar la técnica *one-hot encoding* para el etiquetado es que cada elemento $\omega_i \in \Omega$ se convierte en un vector de magnitud 1 (en la norma L^2), lo que elimina las relaciones ordinales artificiales entre categorías.

Además, "La representación *one-hot* es matemáticamente compatible con la función *softmax* en la capa de salida, que produce una distribución de probabilidad sobre clases mutuamente excluyentes" (Goodfellow et al., 2016) lo que representa otra ventaja significativa, ya que buscamos atribuir cada texto a uno y solo un autor, por lo tanto, en nuestro caso, las categorías sí son efectivamente mutuamente excluyentes. De esta manera se representa adecuadamente que cada texto pertenece exactamente a una clase (autor).

La combinación de una función de activación *softmax* con la función de pérdida de entropía cruzada categórica en la capa de salida es óptima para la clasificación multiclase con el etiquetado *One-hot Encoding* (Goodfellow et al., 2016).

El etiquetado se implementó a través de la librería *Keras* de *TensorFlow*. Dicha librería

incluye las herramientas necesarias para realizar el etiquetado en una sola línea de código que se muestra a continuación:

```
import tensorflow as tf

# Convertir etiquetas a one-hot encoding
labels = tf.keras.utils.to_categorical(labels)
```

Listing 2.5: Código para convertir las etiquetas al formato *One-hot Encoding*

En este caso, estamos renombrando la variable *labels* que habíamos declarado anteriormente como una lista simple, y la convertimos ahora en un arreglo que contiene los vectores creados mediante *One-hot Encoding* como se muestra en el siguiente ejemplo:

Ejemplo 2.2.1. Inicialmente, declaramos la variable *labels* con las etiquetas asignadas a cada autor:

labels = [0,1,2]

posteriormente renombramos nuestra variable con la función "to_categorical" de *Keras*, empleando nuestra lista como parámetro:

labels = tensorflow.keras.utils.to_categorical(labels)

como resultado, la variable *labels* se convierte en un arreglo de la forma:

labels = [[1,0,0],[0,1,0],[0,0,1]]

así obtenemos nuestro arreglo con las etiquetas en formato *One-hot Encoding*

Ahora que tenemos listas nuestras etiquetas para las variables de salida, es momento de procesar los textos para convertirlos en las variables de entrada.

2.2.2 Tokenización y *Padding*

Es necesario convertir los cuentos en vectores numéricos que puedan ser utilizados como entradas de la red neuronal. Recordemos que las entradas de una red neuronal deben ser uniformes, es decir, cualquier elemento debe ser un vector n-dimensional donde n es un número fijo, no puede variar entre una entrada y otra. Sin embargo, los cuentos contenidos en nuestra base de datos, son bastante dispares en cuanto a su extensión. Por otro lado, la red neuronal, al ser una función matemática, trabaja con números, y nuestros cuentos son, hasta este momento, un arreglo de cadenas de caracteres que no pueden fungir como parámetros para nuestra red neuronal. Por ello es que necesitamos, en un primer momento, transformar nuestros cuentos en arreglos numéricos. Para ello, comenzaremos por un proceso de tokenización.

La tokenización es el proceso fundamental en el Procesamiento de Lenguaje Natural (PLN) que consiste en segmentar un texto en unidades más pequeñas llamadas *tokens*. Es-

tos tokens pueden ser palabras, subpalabras, caracteres o incluso unidades más granulares, según el nivel de análisis requerido.

Definición 2.2.2. Dado un texto T como una secuencia de caracteres, la tokenización es la función:

$$\tau : T \to \{t_1, t_2, \ldots, t_n\}$$

donde cada t_i representa un token y n es el número total de tokens en el texto (Jurafsky & Martin, 2020).

Los tokens pueden formarse de diferentes formas, pueden ser, por ejemplo, las unidades más simples, o sea, las letras. De esta forma, podríamos asignarle a la letra "a" el número uno, a la "b" el número dos, y así sucesivamente hasta la "z". Sin embargo, esto traería varios problemas, por ejemplo, si quisiéramos distinguir minúsculas y mayúsculas, tendríamos que asignar un número "a" y otro a la letra "A". Además, también deberíamos distinguir con un número diferente a la letra "á", porque en español, la tilde sobre una vocal modifica el significado de las palabras (no es lo mismo "público", que "publico", que "publicó"). Lo anterior haría que tuviéramos que formar, para cada palabra, un número muy grande y difícil de manejar.

En su lugar, optamos por asignar a cada palabra un token. De esta forma, por ejemplo, a la palabra "lluvia" le asignamos el número 1 y a la palabra "agua" le asignamos el número 2. Lo que buscamos con el proceso de tokenización es asignar una etiqueta numérica a una unidad lingüística, en este caso, a cada palabra. De esta forma, una oración que antes se veía como una cadena de caracteres "La lluvia cae y el suelo se llena de agua" se convierte primero en una lista de palabras ["La", "lluvia", "cae", "y", "el","suelo","se","llena","de","agua"] y luego en un vector numérico [3,1,29,56,10,94,35,7,2], donde cada entrada equivale a una palabra de la oración. Notemos que, de esta forma, las palabras "La" y "la" simbólicamente son distintas, y a cada una se le asignará un número diferente. Pero esta vez será por palabras completas y no por la cantidad de letras contenidas en cada palabra.

Ahora, para asignar cada etiqueta, es decir, para decidir qué número corresponde a cada palabra, hay diferentes métodos. Sin embargo, en esta ocasión utilizamos la función *Tokenizer* incluida en la librería *keras* de *tensorflow*. El siguiente código comprende el proceso de tokenización utilizado en nuestro modelo:

```
# 1. Preprocesar los textos
tokenizer = Tokenizer(num_words=5000)
tokenizer.fit_on_texts(textos_sin_stopwords)
sequences = tokenizer.texts_to_sequences(textos_sin_stopwords)
```

```
5 word_index = tokenizer.word_index
```

Listing 2.6: Código de tokenización de los textos

El algoritmo de la función *Tokenizer* es el siguiente: Primero, tomará todos los cuentos y los unirá en una sola cadena de caracteres; posteriormente, contará la frecuencia de cada palabra; ordena de manera descendente todas las palabras a partir de su frecuencia; a la palabra con mayor frecuencia (la que más se repite) le asigna el número 1, y a la que menos se repite le asigna el número n (tomando en cuenta que n es el número de palabras diferentes contenidas en los textos.

En nuestro código, en la línea 2 se tiene "Tokenizer(num_words=5000)". El parámetro "num_words = 5000" implica que, de todas las palabras contenidas en nuestra base de datos, se tomarán unicamente las cinco mil más frecuentes, el resto serán desechadas y no se tomarán en cuenta al momento de hacer nuestros vectores numéricos. Este proceso tiene la finalidad de limpiar los textos de palabras poco frecuentes, sobre todo, quitar palabras que sólo aparecen una vez en todo el corpus, debido a que dichos vocablos pueden representar ruido para el modelo, puesto que su uso es mínimo y no representan marcas estilísticas propias de un autor en particular. Por otro lado, el mismo algoritmo de *Tokenizer* mapea todas las palabras menos frecuentes a un valor nulo o desconocido. Así, si un autor utiliza de manera muy frecuente alguna palabra de uso no frecuente (sin importar cuál sea esa palabra), el modelo lo detectará como una marca estilística propia de ese autor. Finalmente, en español, las cinco mil palabras mas frecuentes abarcan, por lo general, más del noventa por ciento de la información, así que, con ese número se garantiza que se abarque la mayor parte de las palabras utilizadas, al mismo tiempo que la base de datos se vuelve computacionalmente manejable.

La línea 3 del código pide que se ajuste nuestra nueva base de datos (las cinco mil palabras más frecuentes) a los textos que fueron previamente tratados (que ya no tienen encabezados o pies, ni títulos, y en los que ya se sustrajeron las palabras estructurales). La línea 13 se encarga de convertir los cuentos ya ajustados a vectores numéricos, donde a cada palabra ya se le asignó su etiqueta numérica correspondiente, y donde las palabras menos frecuentes fueron desechadas. La última línea se encarga de crear un diccionario de mapeo donde se especifica qué número corresponde con qué palabra.

Una cuestión importante que debe tomarse en cuenta es que, por defecto, la función *Tokenizer* elimina los signos de puntuación (puntos, comas, signos de interrogación o de admiración, etcétera) y convierte todo a minúsculas. Así, a las palabras "Cielo" y "cielo" se les asigna el mismo número. Esto debe tomarse en cuenta, pues puede ser que algún autor en particular utilice los signos de puntuación como una marca estilística propia (como en el caso de José Saramago que optó por no utilizar signos de puntuación), sin embargo, conservar los signos de puntuación o distinguir entre palabras escritas con mayúsculas y

palabras escritas con minúsculas implica un incremento considerable de la dimensionalidad de los vectores de entrada, y por consiguiente del costo computacional derivado de dicho aumento, por ello, a pesar de que los rasgos estilísticos asociados a los signos de puntuación no serán considerados, se optó por correr el riesgo de perder dicha marca estilística (si existe) en pro de reducir el costo computacional.

El proceso de tokenización completo se describe de manera completa en el siguiente ejemplo:

Ejemplo 2.2.2. Supongamos que tenemos los siguientes cuentos de Augusto Monterroso:

1) "Cuando despertó, el dinosaurio todavía estaba allí."

2) "Por fin, según el cable, la semana pasada la tortuga llegó a la meta. En rueda de prensa declaró modestamente que siempre temió perder, pues su contrincante le pisó todo el tiempo los talones. En efecto, una diezmiltrillonésima de segundo después, como una flecha y maldiciendo a Zenón de Elea, llegó Aquiles."

3) "Hubo una vez un Rayo que cayó dos veces en el mismo sitio; pero encontró que ya la primera había hecho suficiente daño, que ya no era necesario, y se deprimió mucho."

4) "El humor la timidez generalmente se dan juntos. Tú no eres una excepción. El humor es una máscara y la timidez otra. No dejes que te quiten las dos al mismo tiempo."

Nota: Lo primero que tendríamos que hacer es quitar las palabras estructurales, pero como estos cuentos son muy pequeños las vamos a dejar.

En principio, tomaremos los cuatro textos y los juntaremos en uno solo:

"Cuando despertó, el dinosaurio todavía estaba allí. Por fin, según el cable, la semana pasada la tortuga llegó a la meta. En rueda de prensa declaró modestamente que siempre temió perder, pues su contrincante le pisó todo el tiempo los talones. En efecto, una diezmiltrillonésima de segundo después, como una flecha y maldiciendo a Zenón de Elea, llegó Aquiles. Hubo una vez un Rayo que cayó dos veces en el mismo sitio; pero encontró que ya la primera había hecho suficiente daño, que ya no era necesario, y se deprimió mucho. El humor la timidez generalmente se dan juntos. Tú no eres una excepción. El humor es una máscara y la timidez otra. No dejes que te quiten las dos al mismo tiempo."

El siguiente paso es convertir todo a minúsculas y quitar signos de puntuación:

"cuando despertó el dinosaurio todavía estaba allí por fin según el cable la semana pasada la tortuga llegó a la meta en rueda de prensa declaró modestamente que siempre temió perder pues su contrincante le pisó todo el tiempo los talones en efecto una

diezmiltrillonésima de segundo después como una flecha y maldiciendo a zenón de elea llegó aquiles hubo una vez un rayo que cayó dos veces en el mismo sitio pero encontró que ya la primera había hecho suficiente daño que ya no era necesario y se deprimió mucho el humor la timidez generalmente se dan juntos tú no eres una excepción el humor es una máscara y la timidez otra no dejes que te quiten las dos al mismo tiempo"

Ahora se numeran todas las palabras diferentes y se cuenta su frecuencia en el texto unificado.

'el': 6, 'la': 6, 'que': 5, 'una': 5, 'en': 3, 'no': 3, 'de': 3, 'y': 3, 'llegó': 2, 'a': 2, 'tiempo': 2, 'dos': 2, mismo': 2, 'ya': 2, 'se': 2, 'humor': 2, 'timidez': 2, 'cuando': 1, 'despertó': 1, 'dinosaurio': 1, 'todavía': 1, 'estaba': 1, 'allí': 1, 'por': 1, 'fin': 1, 'según': 1, 'cable': 1, 'semana': 1, 'pasada': 1, 'tortuga': 1, 'meta': 1, 'rueda': 1, 'prensa': 1, 'declaró': 1, 'modestamente': 1, 'siempre': 1, 'temió': 1, 'perder': 1, 'pues': 1, 'su': 1, 'contrincante': 1, 'le': 1, 'pisó': 1, 'todo': 1, 'los': 1, 'talones': 1, 'efecto': 1, 'diezmiltrillonésima': 1, 'segundo': 1, 'después': 1, 'como': 1, 'flecha': 1, 'maldiciendo': 1, 'zenón': 1, 'elea': 1, 'aquiles': 1, 'hubo': 1, 'vez': 1, 'un': 1, 'rayo': 1, 'cayó': 1, 'veces': 1, ', 'sitio': 1, 'pero': 1, 'encontró': 1, 'primera': 1, 'había': 1, 'hecho': 1, 'suficiente': 1, 'daño': 1, 'era': 1, 'necesario': 1, 'deprimió': 1, 'mucho': 1, 'generalmente': 1, 'dan': 1, 'juntos': 1, 'tú': 1, 'eres': 1, 'excepción': 1, 'es': 1, 'máscara': 1, 'otra': 1, 'dejes': 1, 'te': 1, 'quiten': 1, 'las': 1, 'al': 1

Notemos que aquí, por ser textos muy breves, la mayoría de las palabras sólo aparecen una vez, y las palabras más frecuentes son las estructurales.

Una vez que tenemos las frecuencias, hay que asignar un valor numérico a cada palabra. La palabra más frecuente tendrá el número 1 y la menos frecuente el número n, pero en este caso, tenemos dos varias palabras con la misma frecuencia, así que las numeraremos sin seguir ningún criterio en específico:

'el': 1, 'la': 2, 'que': 3, 'una': 4, 'en': 5, 'no': 6, 'de': 7, 'y': 8, 'llegó': 9, 'a': 10, 'tiempo': 11, 'dos': 12, mismo': 13, 'ya': 14, 'se': 15, 'humor': 16, 'timidez': 17, 'cuando': 18, 'despertó': 19, 'dinosaurio': 20, 'todavía': 21, 'estaba': 22, 'allí': 23, 'por': 24, 'fin': 25, 'según': 26, 'cable': 27, 'semana': 28, 'pasada': 29, 'tortuga': 30, 'meta': 31, 'rueda': 32, 'prensa': 33, 'declaró': 34, 'modestamente': 35, 'siempre': 36, 'temió': 37, 'perder': 38, 'pues': 39, 'su': 40, 'contrincante': 41, 'le': 42, 'pisó': 43, 'todo': 44, 'los': 45, 'talones': 46, 'efecto': 47, 'diezmiltrillonésima': 48, 'segundo': 49, 'después': 50, 'como': 51, 'flecha': 52, 'maldiciendo': 53, 'zenón': 54, 'elea': 55, 'aquiles': 56, 'hubo': 57, 'vez': 58, 'un': 59, 'rayo': 60, 'cayó': 61, 'veces': 62, ', 'sitio': 63, 'pero': 64, 'encontró': 65, 'primera': 66, 'había': 67, 'hecho': 68, 'suficiente': 69, 'daño': 70, 'era': 71, 'necesario': 72, 'deprimió': 73, 'mucho': 74, 'generalmente': 75, 'dan': 76, 'juntos': 77, 'tú': 78, 'eres': 79, 'excepción': 80, 'es': 81, 'máscara': 82, 'otra': 83, 'dejes': 84, 'te': 85, 'quiten': 86,

'las': 87, 'al': 88

Tenemos 88 palabras diferentes. si tomamos el 95% de ellas tendríamos apróximadamente 84 palabras a las que se les asignará un valor y a las cuatro últimas se les daría un valor nulo. Aunque en este ejemplo, como estamos trabajando con tan pocas palabras, asignar un valor nulo a cuatro palabras puede modificar sustancialmente el sentido de un cuento, porque, si le quitáramos cuatro palabras al cuento 1, por ejemplo, estaríamos perdiendo la mitad del cuento. Aunque estas obras de Augusto Monterroso son atípicas, por ser extremadamente breves, el ejemplo sirve para ilustrar por qué es tan importante trabajar con volúmenes considerables de datos cuando se entrena una red neuronal artificial.

Para continuar con el ejemplo, tomemos las palabras 85, 86, 87 y 88 como valores nulos. Y con esto tendríamos los cuentos tokenizados como vectores numéricos, de la siguiente manera:

1) [18,19,1,20,21,22,23]

2) [24,25,26,1,27,2,28,29,2,30,9,10,2,31,5,32,7,33,34,35,3,36,37,38,39,40,41,42,43, 44,1,45,46,5,47,4,48,7,49,50,51,4,52,8,53,10,54,7,55,9,56]

3) [57,4,58,59,60,3,61,12,62,5,1,13,63,64,65,3,14,2,66,67,68,69,70,3,14,6,71,72,8,15, 73,74]

4) [1,16,2,17,75,15,76,77,78,6,79,4,80,1,16,81,4,82,8,2,17,83,6,84,3,null,null,null,12, null,13,11]

De esta manera tenemos ya los cuentos tokenizados en vectores numéricos.

En el ejemplo anterior, convertimos cuatro cuentos breves a vectores numéricos. Sin embargo, la dimensión de cada vector es distinta, y así no podemos utilizar dichos vectores como entradas de nuestra red neuronal artificial, para ello, es necesario utilizar la técnica de *padding*.

El *padding* consiste en uniformar las dimensiones de todos nuestros vectores rellenando con ceros a la derecha o a la izquierda hasta que todos los vectores tengan exactamente la misma dimensión.

En el ejemplo anterior, el texto más breve es de solo siete palabras, lo que equivale a un vector de dimensión siete, mientras que el más extenso es de cincuenta y un palabras. Esta disparidad también implica cierto ruido para la red, por lo que se opta por reducir la dimensión del vector más largo. Para ello, se toma el 95% del tamaño del vector mayor, de esta manera se garantiza que los cuentos más largos estén casi completos, al mismo tiempo que se busca que los más breves tengan menos ceros (más ruido).

El siguiente código muestra el proceso de *padding* utilizado en nuestro modelo:

```
# Calcular maxlen dinámicamente
text_lengths = [len(seq) for seq in sequences]
maxlen = int(np.percentile(text_lengths, 95))  # Percentil 95
```

```
print(f"Maxlen calculado: {maxlen}")

# Aplicar padding
data = pad_sequences(sequences, maxlen=maxlen)
```

Listing 2.7: Código de *padding* de los textos

Podemos observar que lo primero es medir el vector más largo, y calcular la longitud máxima con el 95% de la longitud del vector mayor. Posteriormente se ajustan todos los textos a esa longitud calculada.

El proceso de *padding* se ilustra en el siguiente ejemplo:

Ejemplo 2.2.3. Tomemos los vectores ya tokenizados del ejemplo anterior:

1) [18,19,1,20,21,22,23]

2) [24,25,26,1,27,2,28,29,2,30,9,10,2,31,5,32,7,33,34,35,3,36,37,38,39,40,41,42,43, 44,1,45,46,5,47,4,48,7,49,50,51,4,52,8,53,10,54,7,55,9,56]

3) [57,4,58,59,60,3,61,12,62,5,1,13,63,64,65,3,14,2,66,67,68,69,70,3,14,6,71,72,8,15, 73,74]

4) [1,16,2,17,75,15,76,77,78,6,79,4,80,1,16,81,4,82,8,2,17,83,6,84,3,null,null,null,12, null,13,11]

Ahora, el vector más largo es de dimensión 51, así que nuestra longitud máxima será de 48 (aproximadamente el 95% de 51). En este caso vamos a completar con ceros a la derecha hasta alcanzar la dimensión 48 (en el caso del vector más largo, tendremos que reducirlo, para ello, eliminaremos las últimas entradas del vector). Así nuestros vectores quedarían de la siguiente manera:

1) [18,19,1,20,21,22,23,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0, 0,0,0,0,0,0,0,0]

2) [24,25,26,1,27,2,28,29,2,30,9,10,2,31,5,32,7,33,34,35,3,36,37,38,39,40,41,42,43, 44,1,45,46,5,47,4,48,7,49,50,51,4,52,8,53,10,54,7]

3) [57,4,58,59,60,3,61,12,62,5,1,13,63,64,65,3,14,2,66,67,68,69,70,3,14,6,71,72,8,15, 73,74,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]

4) [1,16,2,17,75,15,76,77,78,6,79,4,80,1,16,81,4,82,8,2,17,83,6,84,3,null,null,null,12, null,13,11,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]

Así nuestros vectores ya tienen todos la misma dimensión y pueden utilizarse como entradas de la red neuronal artifical.

Con el proceso de *padding* ya tenemos los textos convertidos en vectores numéricos n dimensionales que servirán de entrada para nuestra red neuronal. Sin embargo, para poder capturar las marcas estilísticas, aún falta un proceso más: el proceso de *embedding*. Aunque dicho proceso forma parte de la estructura del modelo de la red neuronal como tal,

no se trata de un tipo de neurona, por ello resulta relevante describir su funcionamiento en la siguiente sección.

2.2.3 Embedding

El proceso de *embedding* consiste en convertir cada palabra en un vector denso en $\mathbb{R}^d$ con la finalidad de capturar relaciones analógicas entre palabras. De manera formal, podemos utilizar la siguiente definición:

Definición 2.2.3. Dado un vocabulario V de tamaño $|V|$, un *embedding* es una función:

$$f : V \to \mathbb{R}^d$$

donde $d << |V|$.

Goodfellow et al. (2016) define dos tipos de *embedding*: al primero de ellos lo denomina como "Bolsa continua de palabras" (CBOW por sus siglas en inglés Continuous Bag Of Words), en el que, dado un texto, se calcula la probabilidad de la palabra w_i, donde i denota la posición de la palabra w en el texto, dadas las palabras $w_{i-t}, w_{i-t+1}, w_{i-t+2}, ...$ anteriores y las palabras $w_{i+1}, w_{i+2}, ...$ posteriores. De esta manera, la Red Neuronal se utiliza para predecir las palabras dentro de un texto, es decir, después del entrenamiento, y dados un grupo de palabras, la Red Neuronal será capaz de predecir cuál debería ser la palabra siguiente en un texto incompleto.

Al otro tipo de *embedding* se denomina Modelo de Skip-Gram, en este caso, la idea es que, dada la palabra w_i la Red Neuronal pueda predecir el contexto de uso de dicha palabra, es decir, qué palabras son las que deben rondar alrededor de w_i.

La idea básica del *embedding* es encontrar una función matemática que nos ayude a definir campos semánticos, es decir, cada palabra en un texto se convierte en un vector $w_i \in \mathbb{R}^d$, y se busca que, dadas dos palabras w_i y w_j en un mismo texto, si estas dos palabras pertenecen a un mismo campo semántico, la distancia entre los vectores w_i y el w_j será pequeña, mientras que palabras que pertenezcan a campos semánticos distintos tendrán una mayor distancia entre sus vectores correspondientes.

Veamos un ejemplo simple de cómo el proceso de *embeddig* puede ayudarnos a establecer relaciones semánticas entre palabras:

Ejemplo 2.2.4. Supongamos que desamos establecer relaciones semánticas entre diversos animales: perro, gato, colibrí, tiburón, ballena, ágila.

En principio podríamos utilizar la técnica de *One-hot encoding* para convertir cada una de estas palabras en vectores, de la siguiente manera:

perro = [1,0,0,0,0,0] gato = [0,1,0,0,0,0] colibrí = [0,0,1,0,0,0] tiburón = [0,0,0,1,0,0] ballena = [0,0,0,0,1,0] águila = [0,0,0,0,0,1]

Podemos notar que tenemos una etiqueta para cada palabra, sin embargo, si medimos las distancias entre cada palabra, podremos notar que todas son equidistantes (pues la distancia entre cualesquiera dos vectores siempre será $\sqrt{2}$), por lo tanto, no podemos establecer relaciones semánticas entre ellos.

Ahora utilicemos un proceso de *embedding* de forma manual (porque nosotros vamos a establecer las relaciones semánticas de manera consciente), formando vectores en $\mathbb{R}^3$, siguiendo esta lógica: en la primera entrada del vector, los valores cercanos a 1 corresponderán con animales terrestres, mientras que valores cercanos a $\frac{1}{2}$ corresponderán con animales aéreos, y los valores cercanos a 0 serán animales acuáticos; para la segunda entrada del vector, los valores cercanos a 1 serán animales domésticos, mientras que entradas cercanas a 0 corresponderán con animales ferales; la tercera entrada del vector será como tal el animal, así, nuestros animales tendrían los siguientes valores:

perro = [0.9,0.9,0.8] gato = [0.8,0.7,0.9] colibrí = [0.4,0.1,0.2] tiburón = [0.1,0.1,0.3] ballena = [0.2,0.1,0.7] águila = [0.5,0.3,0.5]

De esta manera la distancia semántca entre “perro” y “gato” será:

$$d_{pg} = \sqrt{(0.9-0.8)^2 + (0.9-0.7)^2 + (0.8-0.9)^2} \approx 0.2449$$

Pero la distancia entre “perro” y “tiburón” será de:

$$d_{pt} = \sqrt{(0.9-0.1)^2 + (0.9-0.1)^2 + (0.8-0.3)^2} \approx 1.2369$$

De esta manera, podemos observar que distancias más grandes implican menor cercanía semántica, pues un perro está más relacionado con un gato, pues ambos son animales terrestres, domésticos, normalmente son mascotas; que con un tiburón, que es un animal acuático y feral.

Podemos observar también que, entre mayor sea la dimensión d de mis vectores, mayor será el rango de categorías semánticas que pueden relacionar una palabra con otra. En el ejemplo anterior tenemos sólo animales, y los clasificamos en: hábitat (primer campo semántico), domesticación (segundo campo semántico), y animal (tercer campo semántico). En este sentido, si introdujéramos un lobo, éste tendría que ser cercano con el perro en la primera y última entrada, pero lejano en la segunda entrada de nuestro vector de tres dimensiones; sin embargo, si el vector fuera de cinco

dimensiones, la tercera entrada podría ser la familia, donde el perro debería tener el mismo valor que el lobo, pero debería estar más alejado que el gato; la cuarta entrada podría ser la especie; mientras que la quinta podría representar la raza, por poner un ejemplo.

En el ejemplo anterior, observamos como funciona el *embedding* de manera "manual", es decir, las categorías se obtuvieron razonando los campos semánticos. Sin embargo, nuestro razonamiento es, de cierta forma, arbitrario, y está descontextualizado. Mientras que yo propuse que la primera entrada del vector correspondiera con el hábitat, y dividí esta categoría en tres opciones posibles (tierra, agua y aire), alguien más podría establecer que la primera entrada del vector corresponda a otro campo semántico, como el género gramatical de la palabra; donde el género masculino obtuviera valores cercanos a 0 y el femenino a 1.

Por ello es que la Red Neuronal trabajará de manera distinta. La Red Neuronal no establecerá estas relaciones semánticas siguiendo forzosamente una lógica de campos semánticos, sino que establecerá las relaciones con base en las frecuencias de las palabras y su cercanía con todas las demás palabras del corpus. Es decir, si volvemos al ejemplo de la palabra "perro", la Red Neuronal observará las otras palabras que giran en torno a ésta en los textos proporcionados para su entrenamiento, así, puede establecer una relación entre "perro" y "patio", "alimento", "amigo", "leal", "mordida", entre otras, además de que le asignará los artículos "el" y "un", porque así funciona en el lenguaje natural. De esta forma, si los textos de entrenamiento hablan sobre mascotas, es probable que "perro" y "gato" mantengan una distancia pequeña entre sí, como en el ejemplo anterior, pero si los textos hablan sobre sentimientos y emociones, es posible que "perro" sea una palabra muy lejana a "gato", incluso, puede que estas palabras no entren en el campo semántico de los animales debido a que, en el contexto, estas palabras pueden ser tomadas como adjetivos: "perra suerte" o "vida de perros". Así que la relación entre palabras la establecerá el contexto de uso en los textos de entrenamiento.

Finalmente, vale la pena resaltar que nuestra Red Neuronal no utiliza la técnica de *embedding* para generar textos, sino para buscar contextos de uso, o sea, trabajamos con el modelo Skip-Gram. Lo que buscamos es encontrar relaciones contextuales de uso de las palabras en cada autor, es decir, queremos hallar los contextos particulares en los que Borges utiliza la palabra "noche", por ejemplo, y distinguirlos de los contextos en los que Cortázar y Benedetti utilizan la misma palabra, de esta manera es cómo buscamos que nuestra Red Neuronal sea capaz de predecir adecuadamente al autor de un texto.

En la sección anterior, vimos cómo los textos se habían convertido en vectores donde cada entrada representaba una palabra a la que se le había asignado un valor numérico. El proceso de *embedding* se aplica como una primera capa de nuestra Red Neuronal, por

lo que el código de implementación se verá en la siguiente sección. Sin embargo, antes de revisar a fondo la implementación de la Red Neuronal en código de *Python*, tomaremos el ejemplo de la sección anterior para comprender cómo se aplica el *embedding* en nuestro caso particular. Para ello, veremos el siguiente ejemplo:

Ejemplo 2.2.5. En el ejemplo anterior (Ejemplo 2.2.3), teníamos algunos cuentos de Augusto Monterroso ya tokenizados y ajustados a una dimensión $n = 48$ a través del *padding*. Nuestros vectores resultantes fueron los siguientes:

1) [18,19,1,20,21,22,23,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]

2) [24,25,26,1,27,2,28,29,2,30,9,10,2,31,5,32,7,33,34,35,3,36,37,38,39,40,41,42,43,44 ,1,5,46,5,47,4,48,7,49,50,51,4,52,8,53,10,54,7]

3) [57,4,58,59,60,3,61,12,62,5,1,13,63,64,65,3,14,2,66,67,68,69,70,3,14,6,71,72,8,15, 73,74,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]

4) [1,16,2,17,75,15,76,77,78,6,79,4,80,1,16,81,4,82,8,2,17,83,6,84,3,null,null,null,12, null,13,11,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]

Ahora, al aplicar el proceso de *embedding*, lo que haremos es convertir cada entrada de nuestros vectores en un vector denso, dando como resultado un arreglo de vectores (computacionalmente hablando, tenemos un arreglo de vectores, o un vector de vectores, pero desde la perspectiva matemática, hablamos de una matriz).

Supongamos entonces que nuestro *embedding* convertirá cada valor en un vector en $\mathbb{R}^3$, el resultado será que nuestros cuentos se convertirán en matrices $M_i \in M_{48\times 3}$. Sin embargo, para efectos del ejemplo, tomaremos únicamente las primeras siete palabras de cada cuento, con la finalidad de hacer más manejables los datos para presentarlos por escrito, así nuestras matrices serán del tipo $M_i \in M_{7\times 3}$.

Recordemos también que el proceso *embedding* asigna valores de manera aleatoria a cada vector generado, y posteriormente estos valores se ajustarán en las capas posteriores de la Red Neuronal. Para efectos de este ejemplo, los valores se asignarán sin seguir ningún criterio, serán sólo ilustrativos.

Así, nuestros cuentos reducidos se verán de la siguiente forma:

1) [18,19,1,20,21,22,23]

2) [24,25,26,1,27,2,28]

3) [57,4,58,59,60,3,61]

4) [1,16,2,17,75,15,76]

Ahora aplicamos el *embedding*

1) [[0.1,0.8,0.4],[0.1,0.1,0.1],[0.1,0.9,0.5],[0.2,0.0,0.9],[0.2,0.1,0.8],
[0.2,0.2,0.5],[0.2,0.3,0.6]]

2) [[0.2,0.4,0.6],[0.2,0.5,0.1],[0.2,0.6,0.2],[0.1,0.1,0.1],[0.2,0.7,0.4],
[0.2,0.2,0.2],[0.2,0.8,0.6]]
3) [[0.5,0.7,0.5],[0.4,0.4,0.4],[0.5,0.8,0.1],[0.5,0.9,0.5],[0.6,0.0,0.6],
[0.3,0.3,0.3],[0.6,0.1,0.6]]
4) [[0.1.0.1.0.1],[0.1.0.6.0.6],[0.2.0.2.0.2],[0.1,0.7,0.7],[0.7,0.5,0.5],
[0.1,0.5,0.5],[0.7,0.6,0.4]]

Ahora tenemos los arreglos que podemos representar como matrices:

1) $\begin{bmatrix} 0.1 & 0.1 & 0.1 & 0.2 & 0.2 & 0.2 & 0.2 \\ 0.8 & 0.1 & 0.9 & 0.0 & 0.1 & 0.2 & 0.3 \\ 0.4 & 0.1 & 0.5 & 0.9 & 0.8 & 0.5 & 0.6 \end{bmatrix}$

2) $\begin{bmatrix} 0.2 & 0.2 & 0.2 & 0.1 & 0.2 & 0.2 & 0.2 \\ 0.4 & 0.5 & 0.6 & 0.1 & 0.7 & 0.2 & 0.8 \\ 0.6 & 0.1 & 0.2 & 0.1 & 0.4 & 0.2 & 0.6 \end{bmatrix}$

3) $\begin{bmatrix} 0.5 & 0.4 & 0.5 & 0.5 & 0.6 & 0.3 & 0.6 \\ 0.7 & 0.4 & 0.8 & 0.9 & 0.0 & 0.3 & 0.1 \\ 0.5 & 0.4 & 0.1 & 0.5 & 0.6 & 0.3 & 0.6 \end{bmatrix}$

4) $\begin{bmatrix} 0.1 & 0.1 & 0.2 & 0.1 & 0.7 & 0.1 & 0.7 \\ 0.1 & 0.6 & 0.2 & 0.7 & 0.5 & 0.5 & 0.6 \\ 0.1 & 0.6 & 0.2 & 0.7 & 0.5 & 0.5 & 0.4 \end{bmatrix}$

Notemos que, pese a que los valores asignados a cada palabra son aleatorios, se utilizará el mismo vector para designar a la misma palabra.

Así es como se forman las matrices que representan a nuestros cuentos, en este caso, serían cuentos de siete palabras.

Una vez que se aplica el *embedding*, tenemos listas las matrices que pasarán por la siguiente capa, la arquitectura LSTM que será la que modificará los valores de estas matrices para encontrar las relaciones semanticas de cada una de nuestras palabras en los textos de entrenamiento. Y así es como la Red Neuronal asignará contextos de uso a cada Autor.

Como nota final quisiera resaltar que en nuestra implementación real, que veremos en la siguiente sección, la dimensión asignada por el *embedding* fue de $d = 128$, por lo que las matrices reales con las que trabajará nuestra Red Neuronal serán $M_i \in M_{1682\times128}$.

2.3 La red neuronal

Una vez que se ha hecho la limpieza de los datos, es momento de implementar el modelo de Red Neuronal Artificial. El siguiente código muestra la implementeción de nuestra Red Neuronal completa:

```python
import numpy as np
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense, Dropout
from sklearn.model_selection import train_test_split

#Extracción de los datos
"""Los textos se extrajeron y fueron guardados junto con sus etiquetas en las variables textos_sin_stopwords y labels, respectivamente."""

# 1. Preprocesar los textos
tokenizer = Tokenizer(num_words=5000)
tokenizer.fit_on_texts(textos_sin_stopwords)
sequences = tokenizer.texts_to_sequences(textos_sin_stopwords)
word_index = tokenizer.word_index

# Calcular maxlen dinámicamente
text_lengths = [len(seq) for seq in sequences]
maxlen = int(np.percentile(text_lengths, 95))  # Percentil 95
print(f"Maxlen calculado: {maxlen}")

# Aplicar padding
data = pad_sequences(sequences, maxlen=maxlen)

# Convertir etiquetas a one-hot encoding
labels = tf.keras.utils.to_categorical(labels)

# 2. Dividir datos
X_train, X_test, y_train, y_test = train_test_split(data, labels, test_size=0.2)

# 3. Crear modelo LSTM
model = Sequential()
model.add(Embedding(5000, 128, input_length=maxlen))
model.add(LSTM(128, dropout=0.2, recurrent_dropout=0.2))
model.add(Dropout(0.5))  # Añadir dropout para regularización
model.add(Dense(labels.shape[1], activation='softmax'))

# 4. Compilar y entrenar
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
history = model.fit(X_train, y_train, epochs=10, batch_size=32, validation_data=(X_test, y_test))
```

```
42
43 # 5. Evaluar
44 loss, accuracy = model.evaluate(X_test, y_test)
45 print(f"Accuracy: {accuracy}")
```

Listing 2.8: Código que implementa la Red Neuronal Artificial completa

Las primeras siete líneas de código corresponden a la importación de las librerías necesarias para nuestro modelo. En este caso utilizaremos *Tensorflow*, una biblioteca especializada en Redes Neuronales Artificiales que describiremos con más detalle más adelante.

La línea diez es sólo un mensaje que nos indica el proceso de limpieza de los datos que se hizo previamente: eliminar encabezados y pies de los textos y quitar las palabras estructurales.

Posteriormente, de las líneas doce a la dieciséis se lleva a cabo el proceso de "tokenización". La línea trece utiliza la función *Tokenizer* que convertirá cada palabra en un número natural utilizando como criterio de asignación la frecuencia de las palabras (la palabra más frecuente tendrá el número 1). El parámetro de esta función indica que se considerarán únicamente las cinco mil palabras más frecuentes del vocabulario. La línea catorce toma la variable donde están almacenados nuestros textos (textos_sin_stopwords) y construye el vocabulario, es decir, anaĺıza todas las palabras las ordena por frecuencia. En la línea quince, los textos son convertidos en secuencias, es decir, cada cuento se convierte en un vector numérico. Finalmente, la línea dieciséis genera un diccionario de correspondencias entre la palabra y su número asignado.

Una vez que cada cuento se ha convertido en un vector, es necesario uniformar la dimensión de los vectores. Sin embargo, recordemos que las dimensiones son muy heterogéneas, pues los cuentos tienen extensiones muy diversas. No podemos fijar esta dimensión de forma aleatoria, porque tendríamos una gran pérdida de información si la mayoría de los cuentos son más largos que el número que fijamos, pero generaríamos demasiado ruido si la mayoría de los texto son más breves del número fijado.

Por lo anterior, las lineas dieciocho a veintiuno contienen el código para calcular la longitud máxima de los vectores a considerar, es decir, vamos a ajustar de manera automática el número de palabras máximas que tendrá cada vector generado por el proceso de tokenización. La línea diecinueve calcula la longitud de cada secuencia generada. La línea veinte calcula el percentil noventaicinco de las longitudes, es decir, si el 95% de los textos tiene menos que n número de palabras, utiliza n como longitud máxima. De esta forma se reduce al máximo tanto el costo computacional como el ruido. La línea ventiuno simplemente imprime en pantalla cuál fue el número máximo calculado, es decir la dimensión a la que se ajustarán todos los vectores que vamos a utilizar. En nuestro caso particular, con los datos utilizados, la dimensión fue de $n = 1682$.

La línea veinticuatro aplica el proceso de *padding*, es decir, los textos cuyo vector sea

de dimensión mayor que n se ajustarán a n y los que tengan dimensión menor que n se rellenarán con ceros hasta que alcancen una dimensión igual que n.

La línea veintisiete convierte las etiquetas de los autores al formato *one-hot encoding*, es decir, pasan de ser: Borges: 0, Benedetti : 1, Cortázar: 2; al formato Borges = [1,0,0], Benedetti = [0,1,0], Cortázar = [0,0,1].

En la línea treinta se dividen los datos, se utilizará el 80% de los datos para el entrenamiento y el 20% para la validación.

Las líneas treinta y dos a treinta y siete crean el modelo como tal.

La línea trenta y tres indica que el modelo será secuencial, es decir, que se pondrá una capa tras otra y que los datos irán pasando por cada capa una vez por cada época.

La línea treinta y cuatro integra la primera capa de la red neuronal, en este caso, es una capa de *embedding*, o sea, lo primero que hará nuestro modelo es tomar las secuencias (vectores) generados con la tokenización y el *padding*, y generara matrices (arreglos de vectores). En este caso, tomará el vocabulario de las cinco mil palabras más frecuentes, y cada palabra la convertirá en un vector denso de dimensión $d = 128$, y lo aplicará a secuencias de tamaño $n = 1682$ calculadas previamente.

La línea treinta y cinco añade una capa de compuertas LSTM a nuestro modelo. En este caso agregará 128 compuertas de tipo LSTM. Para evitar el sobreajuste se emplean dos tecnicas de *dorpout*. La primera de ellas consiste en desactivar temporalmente (en cada época) de manera aleatoria, el 20% de las entradas que pasan por las compuertas, de esta manera, evitamos que la Red Neuronal "memorice" los contextos de uso de una palabra con la finalidad de evitar que se le asigne siempre el mismo contexto a una palabra determinada. Por otro lado, *recurrent_dropout* en este caso va a desactivar de manera aleatoria el 20% de los estados de memoria anterior. Aunque la finalidad es la misma, la diferencia consiste en que *dropout* desactiva las entradas, mientras que *recurrent_droput* desactiva los estados de memoria en las compuertas. Nuestro modelo combina ambas técnicas debido a que nuestros datos son limitados. Si tuvieramos más textos, sería inecesario utilizar *recurrent_dropout*, pues cada palabra estaría situada en contextos tan diversos que sería muy difícil que la compuerta "memorizara" dichos contextos de uso. A su vez, si nuestros datos no sólo fueran más grandes, sino también más diversos (es decir, que se utilizaran mayor número de palabras en los textos, o sea, que nuestro vocabulario fuera muy variado), tampoco sería necesaro añadir *droput* a nuestras compuertas.

Recordemos que nuestro modelo trabaja con datos muy limitados, por ello es necesario añadir *dropout*, no sólo al interior de las compuertas LSTM, sino al exterior también. Por ello, en la línea treinta y seis se añade una "capa" de *dropout*. Como tal no se trata de una nueva capa de neuronas, sino de una indicación de que, en cada época se desactive de manera aleatoria el 50% de las neuronas. De esta forma garantizamos que nuestras compuertas LSTM no se sobreajusten.

Finalmente, la línea treinta y siete añade una capa de salida densa (totalmente conectada, es decir, que cada compuerta LSTM pasará por las tres neuronas de la capa de salida), donde se asigna una neurona para cada categoría, o sea, una neurona por autor (en nuestro caso particular serán tres neuronas, una por autor). Para esta última capa se utilizará una función *softmax* como función de activación. Recordemos que dicha función sirve para convertir cada salida de las tres neuronas en probabilidades que sumen 1, con ello buscamos que la Red pueda predecir con cierta probabilidad al autor de un texto determinado.

Una vez que se tiene establecido el modelo computacional, resta compilarlo y entrenarlo.

La línea cuarenta se encarga de compilar el modelo. Para ello utilizará la entropía cruzada categórica como función de pérdida, el algoritmo Adam para optimizar, y utilizará la "precisión" como métrica para ayudarnos a visualizar qué tan bien funciona nuestro modelo.

La línea cuarenta y uno se encarga del entrenamiento. Se toman los datos del entrenamiento, se entrenará por diez épocas e irá procesando de 32 datos a la vez. Una vez que se concluyen las diez épocas se utilizarán los datos para validación, y de esta forma se podrá calcula la precisión del modelo.

Para finalizar, las líneas cuarenta y cuatro y cuarenta y cinco evaluan la precisión e imprimen el porcentaje obtenido de dicha métrica.

Con ello, nuestro modelo de Red Neuronal Artificial está completo.

2.3.1 Entrenamiento de la red neuronal

El entrenamiento de nuestra red neuronal consiste en que cad matriz que representa a un cuento pasará por la capa de compuertas LSTM para ajustar sus valores. Recordemos que la capa de *embedding* asigna valores aleatorios a los vectores densos. Y estos valores se irán ajustando en cada época.

Nuestro modelo implementa diez épocas debido a que los datos son limitados. Utilizar menos épocas reduciría el costo computacional y el tiempo de compilación, pero también reduciría la precisión del modelo, debido a que las matrices no alcanzarían valores óptimos de operación, o en otras palabras, no lograrían encontrar las relaciones semánticas y sintácticas que estamos buscando. Por otro lado, aumentar el número de épocas no sólo incrementa el costo computacional y aumenta el tiempo de compilación, además corremos el riesgo de sobreajuste y que la Red Neuronal "memorice" los datos, haciendo que la precisión se incremente a valores muy cercanos al 100%, pero impidiendo que pudiera trabajar con datos distintos de los usados para el entrenamiento.

El riesgo de sobreajuste se reduce considerablemente entre más datos se tengan para

el entrenamiento de la red. Como ya vimos, esto se debe a que, al tener una variedad de ejemplos mucho mayor, los contextos de uso de los vocablos en un texto son mucho más variados, lo que evita que las palabras se asocien únicamente a contextos limitados por los datos disponibles y con ello se mejoran considerablemente los resultados obtenidos. Sin embargo, nuestr modelo está entrenado específicamente para trabajar con pocos datos, pues esto simula de mejor manera las condiciones reales de trabajo, donde se cuenta con una cantidad limitada de textos, muchos de ellos incompletos o fragmentados.

Con todo, el procedimiento es el mismo, en cada época se van ajustando poco a poco los valores de las matrices, y estos se asocian con una categoría. En otras palabras, cada cuento pasará por el análisis de la Red Neuronal y se le asociará un autor específico. Recordemos que los valores asociados a cada columna de la matriz están vinculados con una palabra. El número de columnas implica la cantidad de palabras que tiene un cuento. El número de filas representaría la cantidad de campos semánticos o categorías semánticas que estamos utilizando para asociar las palabras entre sí, y luego asignárselas a un autor en particualr. Entonces lo que "aprende" nuestra Red Neuronal son los contextos de uso que un autor en particular utiliza para cada palabra en el vocabulario base. De esta manera va asignando probabilidades por contexto y no por vocabulario. El modelo no sólo analizará las palabras usadas por un autor en particular, sino los vocablos que giran al rededor de esa palabra en un uso particular. Esto quiere decir que el modelo logrará analisar el estilo sintáctico de cada autor. Posteriormente, cuando se trabaje con un texto sin etiquetar, tratará de ajustar el estilo sintáctico de cada autor a dicho texto, y entonces el resultado será una probabilidad de que el texto pertenezca a alguno de los autores que se utilizaron para el entrenamiento.

Al finalizar el entrenamiento, los valores de las palabras del *embedding* quedan fijos. El ajuste de estos valores implica que también se fijaron los contextos de uso a nuestros datos de entrenamiento. Por lo tanto, si pretendemos que el modelo prediga el texto de un autor que no fue utilizado para el entrenamiento, o si el texto es realmente muy diferente de los utilizados, la red tendrá dificultades para detectar esos cambios. Pero es justamente ese "ecosistema" cerrado de autores lo que permitirá detectar si un texto pertenece o no a los autores utilizados para el entrenamiento. Y, aunque el modelo sí tratará de ajustar cualquier texto a alguno de los tres autores usados, la probabilidad nos puede indicar que el la Red Neuronal está teniendo dificultades para identificar al autor. De hecho una probabilidad menor al 70% sería un buen indicador de que el texto no es tan cercano al estilo de los textos usados en el entrenamiento.

2.3.2 Uso de la red neuronal con datos sin etiquetar

Una vez que el modelo fue entrenado, ya puede utilizarse para hacer predicciones o clasificaciones de textos sin etiquetar. Para ello es necesario implementar una función que nos permita utilizar un texto nuevo. Dicha funcion debe preprocesar el texto (convertirlo a un vector), utilizar el modelo ya entrenado para hacer la predicción, y devolvernos el resultado.

El siguiente código presenta la función que nos permitirá utilizar nuevos textos sin etiquetar:

```
1  # Función para predecir la autoría de un nuevo texto sin etiquetar
2  def predecir_autor(texto, model, tokenizer, maxlen, label_map):
3      # 1. Preprocesar el texto
4      secuencia = tokenizer.texts_to_sequences([texto])  # Convertir el texto
       en una secuencia numérica
5      padded_secuencia = pad_sequences(secuencia, maxlen=maxlen)  # Aplicar
       padding
6
7      # 2. Hacer la predicción
8      prediccion = model.predict(padded_secuencia)  # Obtener las
       probabilidades para cada clase
9
10     # 3. Interpretar la salida
11     clase_predicha = np.argmax(prediccion, axis=1)[0]  # Obtener la clase
       con la probabilidad más alta
12     autor_predicho = list(label_map.keys())[list(label_map.values()).index(
       clase_predicha)]  # Obtener el nombre del autor
13
14     return autor_predicho, prediccion
```

Listing 2.9: Función para utilizar datos sin etiquetar

En nuestro código, las líneas cuatro y cinco aplican tokenización y *padding* respectivamente a una cadena de caracteres, en este caso, el texto nuevo.

La línea ocho aplica el modelo al nuevo vector generado en las líneas anteriores.

Finalmente, la línea once nos da el valor numérico de la clase predicha, y la línea donce nos traduce esa clase al autor correspondiente.

La función regresa tanto al autor predicho como la probabilidad de que el texto haya sido escrito por dicho autor.

Esta función nos ayudará a hacer pruebas para verificar el funcionamiento adecuado de la red. Con ella se hicieron las pruebas sobre textos sin etiquetar. Dichos textos fueron fragmentos de obras de los autores usados en el entrenamiento, o algunos cuentos breves que no fueron utilizados para entrenar la red.

Con esta función nuestro código está completo y es funcional. En el siguiente capítulo veremos los resultados obtenidos con nuestro modelo.

3 Resultados

Se hicieron varias pruebas una vez que el código estuvo listo. Sin embargo, hay que tomar en cuenta varias cuestiones: la primera es que, cuando se extraen los datos, es decir, en la primera parte, cuando se entra a la página de ciudadseva.com para obtener los cuentos de los autores, dichos cuentos se almacenan en una lista de cadenas de caracteres, donde cada cuento corresponde con un elemento de la lista. Posteriormente se modifican estos datos con el proceso de limpieza, es decir, se quitan los encabezados, los pies y se eliminan las palabras estructurales. Para todo lo anterior se escribieron códigos individuales que trabajan de manera modular. No se tiene un solo código que haga todo el proceso de principio a fin, sino que cada parte se va ejecutando de manera individual. Primero se extraen los datos, luego se quitan los encabezados, luego se quitan los pies, y finalmente se eliminan las palabras estructurales.

Esta manera de trabajar resta automatización en el proceso, pero me permitió verificar en cada parte cómo se iban modificando los textos; de esa forma podía asegurarme de que el código realmente estuviera funcionando adecuadamente, lo que reduce el riesgo de errores.

Nuestro modelo extrajo un total de 251 cuentos divididos de la siguiente forma: 82 de Jorge Luis Borges, 87 de Mario Benedetti y 82 de Julio Cortázar.

El código fue escrito y ejecutado directamente en el entorno de *Google Colaboratory*, que, según información de *Google*, la cuenta gratuita proporciona dos Unidades de Procesamiento Central (CPU por sus siglas en inglés) de tipo Intel Xeon con una frecuencia de 2.2 GHz; 13 GB de memoria RAM virtual de los cuales, aproximadamente 12 GB son utilizables; ofrece también acceso a una Unidad de Procesamiento Gráfico (GPU por sus siglas en inglés) de tipo NVidia Tesla K80 con 12 GB de VRAM, pero esta depende de la disponibilidad y del tiempo de compilación. Según la página oficial de Colaboratory, la cuenta gratuita ofrece un máximo de doce horas de compilación, y las unidades de GPU pueden o no estar disponibles.

El mismo código fue probado también en una PC de escritorio con un CPU Ryzen 7 5700x, una GPU RX7700xt y 64 GB de RAM, utilizando un sistema operativo de base Linux y el entorno de programación VSC (Visual Studio Code). Y en una laptop con un CPU Intel Core i9 14900HX, una GPU NVidia RTX4060 y 32 GB de RAM, con un sistema

operativo de base Linux y el entorno de programación VSC. En ambos equipos los tiempos de compilación se redujeron considerablemente; sin embargo, los resultados en cuanto a la precisión del modelo no mejoraron de manera importante en las pruebas realizadas, por lo que se optó por trabajar directamente en *Google Colaboratory*, que ofrece la ventaja de que puede ejecutarse desde cualquier dispositivo que cuente con conexión a internet.

Estas aclaraciones son importantes porque se hicieron varias pruebas, pero cada vez que el código se ejecuta, los valores de los vectores asignados por el *embedding* se establecen de manera aleatoria, lo que modifica los resultados en cada ejecución; además, también se toman de forma aleatoria los datos que se usarán para entrenamiento y los que se usarán para validación, lo que también puede influir. Cuando se ejecuta el código de la Red Neuronal Artificial, este hace modificaciones en los datos que contienen los textos limpios, por lo que, para hacer una nueva prueba, es necesario ejecutar el código completo desde la extracción de los datos; de lo contrario, la Red Neuronal tratará de ejecutarse desde los cuentos ya procesados y marcará un error al no reconocer los datos. Este detalle implica más tiempo de compilación en cada prueba y es una dificultad técnica que deberá corregirse en un trabajo posterior.

3.1 Resultados obtenidos

Durante las pruebas se obtuvieron diferentes porcentajes de precisión del modelo. En promedio, el porcentaje de precisión ronda el 75%. El porcentaje de precisión más alto que se alcanzó fue de 84% y es el resultado de esta prueba el que utilizaremos en nuestro análisis de resultados. Este resultado se consiguió con diez épocas, con un tiempo aproximado de compilación (solo del entrenamiento de la Red Neuronal, sin considerar la extracción y limpieza de los datos) de 4 minutos con 17 segundos. La siguiente imagen muestra la ventana de resultados del modelo:

Figura 3.1: Resultados obtenidos después del entrenamiento

```
Maxlen calculado: 1680
Epoch 1/10
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/embedding.py:90: UserWarning: Argument `input_length` is deprecated. Just remove it.
  warnings.warn(
7/7 ━━━━━━━━━━━━━━━━━━━━ 42s 5s/step - accuracy: 0.3742 - loss: 1.0972 - val_accuracy: 0.3600 - val_loss: 1.0923
Epoch 2/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 39s 5s/step - accuracy: 0.5863 - loss: 1.0783 - val_accuracy: 0.4400 - val_loss: 1.0833
Epoch 3/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 40s 5s/step - accuracy: 0.7126 - loss: 1.0389 - val_accuracy: 0.4600 - val_loss: 1.0702
Epoch 4/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 48s 5s/step - accuracy: 0.6687 - loss: 0.9388 - val_accuracy: 0.5400 - val_loss: 0.9857
Epoch 5/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 34s 5s/step - accuracy: 0.8402 - loss: 0.7044 - val_accuracy: 0.8200 - val_loss: 0.6920
Epoch 6/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 43s 5s/step - accuracy: 0.8103 - loss: 0.5956 - val_accuracy: 0.7000 - val_loss: 0.6925
Epoch 7/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 41s 5s/step - accuracy: 0.8554 - loss: 0.4951 - val_accuracy: 0.6800 - val_loss: 0.6817
Epoch 8/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 40s 5s/step - accuracy: 0.9143 - loss: 0.3796 - val_accuracy: 0.6800 - val_loss: 0.7036
Epoch 9/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 39s 5s/step - accuracy: 0.9611 - loss: 0.2201 - val_accuracy: 0.7800 - val_loss: 0.5989
Epoch 10/10
7/7 ━━━━━━━━━━━━━━━━━━━━ 41s 5s/step - accuracy: 0.9763 - loss: 0.1558 - val_accuracy: 0.8400 - val_loss: 0.5597
2/2 ━━━━━━━━━━━━━━━━━━━━ 1s 450ms/step - accuracy: 0.8100 - loss: 0.5930
Accuracy: 0.8399999737739563
```

La precisión del modelo se calcula tomando en cuenta los datos de evaluación: se verifica

cuántos datos clasificó correctamente y se divide entre el total de las predicciones. Para entender cómo funciona la "precisión" del modelo, veamos el siguiente ejemplo:

Ejemplo 3.1.1. Supongamos que tenemos a nuestros tres autores clasificados como: Borges = 0, Cortázar = 1 y Benedetti = 2, y que usamos diez textos para validación. El modelo conoce las etiquetas reales de estos datos, pues los tomó de los datos etiquetados que le dimos al principio.

Imaginemos que nuestros diez textos tienen las siguientes etiquetas reales:

```
Textos reales = [0,0,1,2,2,1,0,2,1,1]
```

Ahora, digamos que el modelo hizo las siguientes predicciones:

```
Textos predichos = [0,0,0,2,2,1,0,1,2,1]
```

Podemos notar que predijo correctamente siete textos y se equivocó en tres. Así:

$$\text{Precisión} = \frac{\text{Predichos correctamente}}{\text{Total de textos}} = \frac{7}{10} = 0.7 = 70\%$$

De manera implícita se genera la siguiente matriz de confusión:

R/P	*0*	*1*	*2*
0	3	0	0
1	1	2	1
2	0	1	2

Esto quiere decir que, para el caso de Borges, el modelo predijo correctamente los tres textos y no se confundió ninguna vez; para ortázar, predijo correctamente dos textos y se equivocó en dos (uno lo confundió con Borges y el otro con Benedetti); para Benedetti, predijo correctamente dos textos y confundió uno con Cortázar.

Así es como se determina la precisión y se interpreta la matriz de confusión.

De los 251 textos, aproximadamente 50 fueron utilizados para la validación; esto quiere decir que, en una de las pruebas estándar, nuestro modelo predijo adecuadamente cuarenta de ellos y tuvo problemas para identificar diez. La siguiente imagen presenta los resultados de otra prueba, pero ahora se incluye la matriz de confusión:

Figura 3.2: Resultados obtenidos con matriz de confusión

```
Maxlen calculado: 1718
Epoch 1/14
4/4 ━━━━━━━━━━━━━━━━━━━━ 37s 6s/step - accuracy: 0.3505 - loss: 1.0974 - val_accuracy: 0.2941 - val_loss: 1.0908
Epoch 2/14
4/4 ━━━━━━━━━━━━━━━━━━━━ 26s 6s/step - accuracy: 0.4638 - loss: 1.0799 - val_accuracy: 0.2941 - val_loss: 1.0846
Epoch 3/14
4/4 ━━━━━━━━━━━━━━━━━━━━ 26s 7s/step - accuracy: 0.5613 - loss: 1.0572 - val_accuracy: 0.3137 - val_loss: 1.0755
Epoch 4/14
4/4 ━━━━━━━━━━━━━━━━━━━━ 25s 6s/step - accuracy: 0.6842 - loss: 1.0308 - val_accuracy: 0.4118 - val_loss: 1.0616
Epoch 5/14
4/4 ━━━━━━━━━━━━━━━━━━━━ 41s 6s/step - accuracy: 0.8130 - loss: 0.9743 - val_accuracy: 0.4706 - val_loss: 1.0333
Epoch 6/14
4/4 ━━━━━━━━━━━━━━━━━━━━ 43s 7s/step - accuracy: 0.7813 - loss: 0.9673 - val_accuracy: 0.6078 - val_loss: 0.9736
Epoch 7/14
4/4 ━━━━━━━━━━━━━━━━━━━━ 42s 7s/step - accuracy: 0.8717 - loss: 0.8045 - val_accuracy: 0.6667 - val_loss: 0.9301
Epoch 8/14
4/4 ━━━━━━━━━━━━━━━━━━━━ 39s 7s/step - accuracy: 0.8770 - loss: 0.6653 - val_accuracy: 0.6667 - val_loss: 0.8265
Epoch 9/14
4/4 ━━━━━━━━━━━━━━━━━━━━ 41s 7s/step - accuracy: 0.8937 - loss: 0.5049 - val_accuracy: 0.6275 - val_loss: 0.8329
Epoch 10/14
4/4 ━━━━━━━━━━━━━━━━━━━━ 24s 6s/step - accuracy: 0.9276 - loss: 0.3729 - val_accuracy: 0.6667 - val_loss: 0.8336
Epoch 11/14
4/4 ━━━━━━━━━━━━━━━━━━━━ 26s 6s/step - accuracy: 0.9601 - loss: 0.2660 - val_accuracy: 0.7647 - val_loss: 0.6708
Epoch 12/14
4/4 ━━━━━━━━━━━━━━━━━━━━ 26s 6s/step - accuracy: 0.9662 - loss: 0.2008 - val_accuracy: 0.6275 - val_loss: 0.8339
Epoch 13/14
4/4 ━━━━━━━━━━━━━━━━━━━━ 26s 7s/step - accuracy: 0.9600 - loss: 0.1457 - val_accuracy: 0.7451 - val_loss: 0.7209
Epoch 14/14
4/4 ━━━━━━━━━━━━━━━━━━━━ 24s 6s/step - accuracy: 0.9693 - loss: 0.1159 - val_accuracy: 0.7647 - val_loss: 0.6736
2/2 ━━━━━━━━━━━━━━━━━━━━ 2s 1s/step - accuracy: 0.7702 - loss: 0.6584
Accuracy: 0.7647058963775635
2/2 ━━━━━━━━━━━━━━━━━━━━ 3s 984ms/step

Matriz de Confusión:
[[17  0  0]
 [ 3  9  2]
 [ 2  5 13]]
```

Es necesario resaltar que en esta prueba se utilizaron 14 épocas (en lugar de diez) y que se tomaron 51 cuentos para el entrenamiento (una submuestra del total de 251). La matriz de confusión nos muestra que el modelo predijo adecuadamente 17 textos de Borges, sin equivocarse en ninguno; predijo adecuadamente 9 textos de Benedetti, pero confundió tres con Borges y dos con Cortázar; y predijo adecuadamente trece de Cortázar, pero confundió dos con Borges y uno con Benedetti. Esta matriz de confusión no estuvo presente en las pruebas anteriores porque el código de implementación se adecuó de manera posterior; por ello no se tienen matrices de confusión de las pruebas anteriores.

El porcentaje promedio de precisión es bueno, y el máximo porcentaje alcanzado también es bueno, sin llegar a ser excelente. Nos gustaría tener un porcentaje de precisión superior al 90%, pero los datos de entrenamiento son escasos para alcanzar esos niveles de precisión. Sin embargo, el 75% es un buen porcentaje considerando bases de datos pequeñas, pues estamos simulando condiciones reales de trabajo.

3.1.1 Pruebas con datos no etiquetados

En cuanto al trabajo con datos sin etiquetar, se hicieron pruebas con diez textos: 3 de Mario Benedetti (dos de estos fueron fragmentos de la novela *La tregua*, el otro fue un cuento breve); 4 de Jorge Luis Borges (consistieron en fragmentos de algunos de sus

ensayos y dos cuentos breves); y 3 de Julio Cortázar (dos fueron fragmentos de la novela *Rayuela*, el otro fue un cuento breve).

Con estos textos se hicieron dos tipos de pruebas. La primera fue con los textos sin limpieza adicional, es decir, los cuentos simplemente se tokenizaron y se les aplicó el *padding*. La segunda prueba consistió en quitar las palabras estructurales previamente antes del proceso de tokenización.

Con los textos íntegros de la primera prueba, los resultados fueron muy favorables. El modelo predijo adecuadamente siete de diez textos, lo cual es congruente con el porcentaje de precisión arrojado por las pruebas con los datos de entrenamiento y validación. En cuanto a los errores, predijo un fragmento de *La tregua* de Mario Benedetti como de Borges; un fragmento de *Rayuela* de Cortázar como de Borges; y otro fragmento de *Rayuela* como de Benedetti. Esto también es congruente con los datos arrojados en la matriz de confusión, que muestran que el modelo suele confundir a Cortázar y Benedetti con Borges. Los cuatro textos de Borges fueron clasificados adecuadamente.

La matriz de confusión de las predicciones con textos sin etiquetar (considerando las etiquetas Borges = 0, Benedetti = 1, Cortázar = 2) quedaría de la siguiente forma:

$$\begin{bmatrix} 4 & 0 & 0 \\ 1 & 2 & 0 \\ 1 & 1 & 1 \end{bmatrix}$$

La siguiente imagen muestra las predicciones erróneas de nuestro modelo:

Figura 3.3: Errores de predicción con datos sin etiquetar

```
#Ejemplo de uso: Otro fragmento de "La tregua" de Benedetti
nuevo_texto4 = "Le giraba dentro del cráneo vacío, sordo y punzante. Un panal se había levantado en las cuatro paredes de su calavera

autor4, probabilidades4 = predecir_autor(nuevo_texto4, model, tokenizer, maxlen, label_map)
print(f"El texto probablemente fue escrito por: {autor4}")
print(f"Probabilidades: {probabilidades4}")
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 630ms/step
El texto probablemente fue escrito por: Jorge Luis Borges
Probabilidades: [[0.72139007 0.12411395 0.154496  ]]
```

```
#Ejemplo de uso: Este texto es un fragmento de la novela de "Rayuela" de Cortázar.
nuevo_texto5 = "Sí, pero quién nos curará del fuego sordo, del fuego sin color que corre al anochecer por la rue de la Huchette, sali

autor5, probabilidades5 = predecir_autor(nuevo_texto5, model, tokenizer, maxlen, label_map)
print(f"El texto probablemente fue escrito por: {autor5}")
print(f"Probabilidades: {probabilidades5}")
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 339ms/step
El texto probablemente fue escrito por: Jorge Luis Borges
Probabilidades: [[0.7281527  0.04014308 0.23170425]]
```

```
#Ejemplo de uso: Este texto es un fragmento de "Rayuela de Cortázar"

nuevo_texto7 = "Pero el amor, esa palabra... Moralista Horacio, temeroso de pasiones sin una razón de aguas hondas, desconcertado y a

autor7, probabilidades7 = predecir_autor(nuevo_texto7, model, tokenizer, maxlen, label_map)
print(f"El texto probablemente fue escrito por: {autor7}")
print(f"Probabilidades: {probabilidades7}")
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 382ms/step
El texto probablemente fue escrito por: Mario Benedetti
Probabilidades: [[0.16336042 0.72553635 0.11110326]]
```

La siguiente imagen muestra algunos ejemplos de textos sin etiquetar predichos ade-

cuadamente por nuestro modelo:

Figura 3.4: Ejemplos de textos etiquetados adecuadamente

```
# Ejemplo de uso: el primer texto es de Mario Benedetti, fragmento de la novela "La tregua"
nuevo_texto = "Esta tarde, cuando venía de la oficina, un borracho me detuvo en la calle. No

autor, probabilidades = predecir_autor(nuevo_texto, model, tokenizer, maxlen, label_map)
print(f"El texto probablemente fue escrito por: {autor}")
print(f"Probabilidades: {probabilidades}")
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 449ms/step
El texto probablemente fue escrito por: Mario Benedetti
Probabilidades: [[0.05153144 0.9207268  0.02774183]]
```

```
# Ejemplo de uso, este texto es de Julio Cortázar, cuento incluido en "Historias de Cronopios
nuevo_texto2 = "Los famas para conservar sus recuerdos proceden a embalsamarlos en la siguien

autor2, probabilidades2 = predecir_autor(nuevo_texto2, model, tokenizer, maxlen, label_map)
print(f"El texto probablemente fue escrito por: {autor2}")
print(f"Probabilidades: {probabilidades2}")
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 431ms/step
El texto probablemente fue escrito por: Julio Cortázar
Probabilidades: [[0.0470555  0.07655831 0.87638617]]
```

```
#Ejemplo de uso: El tercer texto es un fragmento de un ensayo de Borges.
nuevo_texto3 = "Los sueños son el género; la pesadilla, la especie. Hablaré de los sueños y,

autor3, probabilidades3 = predecir_autor(nuevo_texto3, model, tokenizer, maxlen, label_map)
print(f"El texto probablemente fue escrito por: {autor3}")
print(f"Probabilidades: {probabilidades3}")
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 601ms/step
El texto probablemente fue escrito por: Jorge Luis Borges
Probabilidades: [[0.93186474 0.01119019 0.05694506]]
```

Algo que debemos tener en cuenta es que, en los resultados de los textos predichos adecuadamente, la probabilidad asignada por el modelo es mayor que 0.8, mientras que en los predichos de manera inadecuada la probabilidad ronda 0.72 en los tres casos. Así, una probabilidad menor de 0.8 debería ser considerada como una predicción poco certera y probablemente inadecuada. Sin embargo, estos datos son solo indicadores y no siempre son del todo certeros. También hay algunos casos donde las diferencias de probabilidades son mínimas y aun así el texto se predijo adecuadamente; aunque estos casos no son los más frecuentes, sí se deben tener en consideración. La siguiente imagen muestra un ejemplo de cómo una diferencia muy pequeña hizo que el modelo se inclinara por Borges, y su predicción resultó adecuada:

Figura 3.5: Ejemplos de probabilidades muy cercanas

```
#Ejemplo de uso: Este texto es un pequeño cuento de Borges
nuevo_texto6 = "Iba y venía, delicado y fatal, cargado de infinita energía, del otro lado de
autor6, probabilidades6 = predecir_autor(nuevo_texto6, model, tokenizer, maxlen, label_map)
print(f"El texto probablemente fue escrito por: {autor6}")
print(f"Probabilidades: {probabilidades6}")
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 407ms/step
El texto probablemente fue escrito por: Jorge Luis Borges
Probabilidades: [[0.37847826 0.3415684  0.27995336]]
```

3.1.2 Resultados con eliminación de palabras estructurales

Con los textos de la segunda prueba, es decir, donde se eliminaron las palabras estructurales de los textos sin etiquetar, el modelo simplemente no funcionó adecuadamente. En todas sus predicciones el resultado fue Jorge Luis Borges. Se esperaba que las predicciones mejoraran al trabajar con textos en condiciones más cercanas a los textos de entrenamiento, pues el modelo se entrenó eliminando las palabras estructurales de los textos originales; sin embargo, esto no fue así.

La explicación que se encuentra para estos resultados es la siguiente: el vocabulario formado por el modelo ya eliminó las palabras estructurales; esto implica que, cuando se le da un nuevo texto como parámetro, todas las palabras que no se encuentren en el vocabulario formado serán ignoradas. Por lo tanto, de alguna manera, los textos nuevos ya tienen una limpieza implícita de palabras estructurales generada por el vocabulario que nuestra Red Neuronal está tomando en cuenta. Cuando nosotros eliminamos las palabras estructurales de los textos sin etiquetar, estamos modificando la sintaxis del texto, lo que genera que el modelo trabaje sobre un texto modificado y no pueda operar adecuadamente.

Aunque el modelo presenta resultados prometedores, la realidad es que aún hay varias dificultades que deben salvarse para poder utilizar esta Red Neuronal Artificial como herramienta de apoyo para la filología, el análisis literario o la atribución de autoría de textos literarios.

3.1.3 Dificultades

Algunas de las dificultades detectadas durante la implementación de nuestro modelo tienen que ver con el tamaño y distribución del conjunto de datos recopilados. Si bien es cierto que se intentó balancear las clases seleccionando autores con número similar de cuentos, también es cierto que la disparidad en las extensiones de los cuentos representa un obstáculo que debe resolverse en un trabajo futuro. Por lo tanto, habría que buscar una manera de balancear los datos de cada clase (autor) de manera que los bloques de texto tengan dimensiones semejantes. Con ello se resolverían dos problemas: primero, se reduciría

el ruido generado por el *padding*; segundo, se evitaría desechar los textos demasiado largos.

Con respecto al punto anterior, también es necesario ampliar el vocabulario para evitar dejar fuera vocablos que, si bien pueden ser de uso poco frecuente, podrían constituir una marca estilística de un autor en particular. Al tener las clases mejor balanceadas, también podría incrementarse el tamaño del vocabulario sin riesgo de sobreajuste.

Es necesario también reconsiderar la limpieza de las palabras estructurales. Si bien eliminarlas reduce considerablemente el tiempo de compilación (lo que se traduce en un menor costo computacional), también es cierto que se está dejando de lado una posible marca estilística relacionada directamente con dichas palabras. Una posible solución sería reducir el vocabulario estructural a palabras sin flexión sintáctica como preposiciones, conjunciones y algunos adverbios, o bien conservar los cuentos íntegros y utilizar equipos de cómputo con mayores recursos para trabajar con el conjunto de datos sin necesidad de limpieza de palabras estructurales. El costo computacional se incrementará, pero también podría incrementarse la eficacia del modelo al momento de clasificar adecuadamente.

En relación con el código propuesto para nuestro modelo, también es importante hacer ciertos ajustes. Si bien es cierto que el objetivo de diseñar, entrenar y probar una Red Neuronal Artificial que predijera autoría de textos sí se cumplió, la realidad es que hay muchas cosas que pueden mejorarse en un trabajo futuro. La primera mejora que habría que implementar es un intervalo de confianza para las predicciones del modelo. La implementación actual calcula las probabilidades de que un texto pertenezca a alguno de los tres autores utilizados para el entrenamiento y posteriormente asigna el texto al autor con la probabilidad más alta, sin importar que la diferencia entre probabilidades sea mínima. Es decir, si el resultado son las siguientes probabilidades: Borges = 0.34, Benedetti = 0.33 y Cortázar = 0.33, el modelo arrojará que el texto es de Borges, aunque en realidad la diferencia es mínima y no se tienen elementos suficientes para asegurarlo. Por ello, habría que implementar una solución para que el modelo atribuya el texto únicamente cuando la probabilidad supere un umbral, por ejemplo, cuando $P \geqslant 0.75$.

Además, sería importante establecer una nueva clase que permita clasificar textos que no pertenecen a ninguno de los autores utilizados para el entrenamiento. Para ello, sería necesario tomar un conjunto amplio de datos de autores diversos e integrarlos todos en esa nueva categoría, de manera que el modelo pudiera predecir si el texto es de Borges, Benedetti o Cortázar, o si, por el contrario, no puede atribuirse a ninguno de esos autores.

Finalmente, un trabajo próximo podría utilizar *embeddings* preentrenados y contextualizados para mejorar los resultados en las predicciones. El uso de estos *embeddings* está restringido por ciertas características del equipo, pues la mayoría de ellos requieren tarjetas de vídeo de la marca *Nvidia* específicamente, y no todas las tarjetas gráficas de esta marca tienen el soporte necesario para dichos modelos de Procesamiento del Lenguaje Natural. Sin embargo, utilizar estos modelos preentrenados podría significar una mejora

importante en el desempeño de nuestro modelo.

Conclusiones

Logros del trabajo

Después de probar el código, podemos afirmar que sí es posible diseñar, implementar, entrenar y utilizar una Red Neuronal Artificial para hacer clasificaciones y predicciones de autoría de textos con conjuntos de datos limitados y heterogéneos utilizando las herramientas gratuitas ofrecidas por *Google Colaboratory*. Aunque una red con estas características presentará limitaciones tanto por el conjunto de datos como por las condiciones del *hardware*, la red es funcional y puede utilizarse con las respectivas precauciones que su uso conlleva.

Las compuertas LSTM resultaron de gran utilidad y pueden ser entrenadas de manera eficaz. Esto es una gran ventaja si tomamos en cuenta que estas compuertas presentan requerimientos técnicos menos demandantes, pues no es necesario contar con *hardware* de marcas privativas, como es el caso de los *transformers* y demás modelos preentrenados. Pese a que su uso se centra principalmente en la generación automática de texto, los resultados demuestran que su implementación en la atribución de autoría de textos fue un acierto, sobre todo si consideramos que estamos trabajando en equipos cuyo único requerimiento técnico es una conexión estable a internet y una cuenta gratuita a las herramientas de *Google*.

El conjunto de datos utilizado en este trabajo tuvo la finalidad de emular condiciones reales de trabajo, pues se pretende que el modelo pueda implementarse en la atribución de textos clásicos, donde se encuentran textos muy fragmentados e incompletos. Además, el corpus literario clásico, aunque es mucho mayor que nuestro conjunto de datos, tanto en la cantidad de textos como en la cantidad de autores, presenta características muy semejantes en cuanto al desbalance de las clases: hay autores que tienen una gran cantidad de textos (como podría ser Ovidio) y otros de los que apenas se tienen algunos fragmentos (como el caso de Estacio).

Limitaciones del modelo

Es importante considerar las limitaciones del modelo. Hasta el momento, la red siempre buscará ajustar los datos a una de sus tres categorías (autores), incluso si el texto no pertenece a ninguno de ellos, lo que limita la fiabilidad del modelo. Además, la limpieza de las palabras estructurales implica una modificación directa de los textos, y el vocabulario formado por el modelo limita también el uso de palabras poco frecuentes. Dichas limitaciones, si bien no impiden que la Red Neuronal Artificial opere, sí deben tenerse en cuenta si se busca trabajar con este modelo en condiciones reales y no solo con datos de entrenamiento.

Propuestas de trabajo futuro

A partir de las dificultades detectadas durante la implementación, se identifican varias líneas de mejora para trabajos futuros:

Balanceo de datos y manejo de longitudes dispares

La disparidad en las extensiones de los cuentos representa un obstáculo que debe resolverse. Sería necesario buscar una manera de balancear los datos de cada clase (autor) de manera que los bloques de texto tengan dimensiones semejantes. Con ello se resolverían dos problemas: primero, se reduciría el ruido generado por el *padding*; segundo, se evitaría desechar los textos demasiado largos.

Ampliación del vocabulario

Es necesario ampliar el vocabulario para evitar dejar fuera vocablos que, si bien pueden ser de uso poco frecuente, podrían constituir una marca estilística de un autor en particular. Al tener las clases mejor balanceadas, también podría incrementarse el tamaño del vocabulario sin riesgo de sobreajuste.

Manejo de palabras estructurales

Si bien eliminar las palabras estructurales reduce considerablemente el tiempo de compilación (lo que se traduce en un menor costo computacional), también es cierto que se está dejando de lado una posible marca estilística relacionada directamente con dichas palabras. Una posible solución sería reducir el vocabulario estructural a palabras sin flexión sintáctica como preposiciones, conjunciones y algunos adverbios, o bien conservar los cuentos íntegros y utilizar equipos de cómputo con mayores recursos para trabajar

con el conjunto de datos sin necesidad de limpieza de palabras estructurales. El costo computacional se incrementará, pero también podría incrementarse la eficacia del modelo al momento de clasificar adecuadamente.

Mejoras en el código y la clasificación

En relación con el código propuesto, es importante implementar un intervalo de confianza para las predicciones del modelo. La implementación actual calcula las probabilidades de que un texto pertenezca a alguno de los tres autores y atribuye el texto al autor con la probabilidad más alta, sin importar que la diferencia sea mínima. Por ejemplo, si el resultado es Borges = 0.34, Benedetti = 0.33 y Cortázar = 0.33, el modelo arrojará que el texto es de Borges, aunque en realidad la diferencia es mínima y no hay elementos suficientes para asegurarlo. Sería conveniente implementar una solución para que el modelo atribuya el texto únicamente cuando la probabilidad supere un umbral, por ejemplo, cuando $P \geqslant 0.75$.

Además, sería importante establecer una nueva clase que permita clasificar textos que no pertenecen a ninguno de los autores utilizados para el entrenamiento. Para ello, sería necesario tomar un conjunto amplio de datos de autores diversos e integrarlos en esa nueva categoría, de manera que el modelo pueda predecir si el texto es de Borges, Benedetti o Cortázar, o si, por el contrario, no puede atribuirse a ninguno de ellos.

Uso de embeddings preentrenados

Finalmente, un trabajo futuro podría utilizar *embeddings* preentrenados y contextualizados para mejorar los resultados en las predicciones. El uso de estos *embeddings* está restringido por ciertas características del equipo, pues la mayoría requieren tarjetas de vídeo de la marca *Nvidia* específicamente, y no todas las tarjetas gráficas de esta marca tienen el soporte necesario para dichos modelos de Procesamiento del Lenguaje Natural. Sin embargo, utilizar estos modelos preentrenados podría significar una mejora importante en el desempeño de nuestro modelo.

Conclusión final

En conclusión, una Red Neuronal Artificial con compuertas LSTM es una herramienta práctica y muy útil para apoyar trabajos en filología y estudios literarios. Es necesario refinar el modelo con las mejoras propuestas, pero su uso en condiciones reales de trabajo es factible y puede reducir el trabajo manual y el tiempo de investigación en la atribución de autoría de textos.

Bibliografía

Abbas, A., & Naser Alzubaidi, A. M. (2023). Ancient Textual Restoration Using Deep Neural Networks: A Literature Review. *Al-Sadiq International Conference on Comuncation and Information Tecnology*.

Aggarwal, C. C. (2018a). *Machine learning for text*. Springer.

Aggarwal, C. C. (2018b). *Neural Networks and Deep Learning*. Springer.

Aggarwal, C. C. (2020). *Linear Algebra and Optimization for Machine Learning*. Springer.

Assael, Y., Sommerschield, T., Shillingford, B., Bordbar, M., Pavlopolus, J., Chatzipanagiotou, M., Androutsopolus, I., Prag, J., & de Freitas, N. (2022). Restorin and attriuting acient texts using deep neural networks. *Nature*, *603*.

Bonomi, G. (s/a). *Deep Learning Techniques for Authorship Attribution of Literary Texts*. Linnaeus University.

Gagala, L. (2018). Authorship attribution with neural networks. *CEURS-WS*, *2125*.

Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT Press.

Hilera González, J. R., & Martínez Hernando, V. J. (1995). *Redes Neuronales Artificiales. Fundamentos, Modelos y Aplicaciones*. Addison-Wesley Iberoamericana.

Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. *Neural Computation*, *9*.

Huertas-Tato, J., A., M., & Camacho, D. (2022). PART: Pre-trained Authorship Representation Transformer. https://doi.org/10.48550/arXiv.2209.15373

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2023). *An Introduction to Statitical Learning*. Springer.

Jurafsky, D., & Martin, J. (2020). *Speech and Language Processing*. Pearson.

Kovács, G., Alonso, P., & Saini, R. (2021). Challenges of Hate Speech Detection in Social Media. *SN Computer Science*, *2*.

López Nieves, L. (1996). *Ciudad Seva. Casa digital del escritor Luis López Nieves*. Consultado el 7 de febrero de 2024, desde https://ciudadseva.com

Martín del Brío, B., & Sanz Molina, A. (2001). *Redes Neuronales y Sistemas Difusos*. Alfaomega.

Martín López, M. (2024). *¿Qué es la entropía cruzada en Deep Learning*. Consultado el 3 de abril de 2026, desde https://keepcoding.io/blog/entropia-cruzada-deep-learning

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems*, *26*.

Raschka, S., & Mirjalili, V. (2019). *Python Machine Learning*. Packt.

Saladin, K. S. (2012). *Anatomía y fisiología: la unidad entre forma y función*. McGraw-Hill.

Silaparasetty, N. (2020). *Machine Learning Concepts with Python and Jupyter Notebook Environment*. Apress.

Simic, M. (2025). *Diference Between the Cost, Loss, and the Objective Function*. Consultado el 3 de marzo de 2026, desde https://www.baeldung.com/cs/cost-vs-loss-vs-objective-function

Tang, X. (2024). Autor identification of literary works based on text analysis and deep learning. *Heliyon*, *10*.

Uquillas Trujillo, G. B., & Moposita Lasso, R. M. (2025). Arquitecturas neuronales para la clasificación de sentimientos: una evaluación empírica de LSTM, BERT y CNN usando PyTorch. *Technology Rain Journal*, *4*. https://doi.org/10.55204/trj.v4i2.e100

Zhao, L., Shi, J., Zhang, C., & Liu, Z. (2025). Authorship Detection on Classical Chinese Text Using Deep Learning. *applied sciences*, *15*.

Zhou, V. (2019). *Machine Learning for Beginners: An Introduction to Neural Networks*. Consultado el 3 de marzo de 2024, desde https://victorzhou.com/blog/intro-to-neural-networks/